Machine Learning Homework 6 Technical Report

tags: `Course1082` `IOC5191`

> 網工所 0856523 許尹睿

Since there are a lot of GIFs to be showed, I place all images in directory Result instead of directly putting all of them in the report.

Naming Principle for each GIF explained in **Appendix**, which is for TA's convenience.

Kernel K-means

Extract data & Normalization

```
# read the input images
def read_image(path):
    return cv2.imread(path).reshape((10000, 3))

image1 = read_image('./image1.png')
image2 = read_image('./image2.png')
```

read_image function transforms the input image into a 2-D numpy array. The shape of this numpy array is (10000, 3).

Initialize the classification

```
def initialization(image, k, method):
    if method == 'random':
        classification = np.random.randint(k, size=10000)
        return classification
    elif method == 'equal-divide':
        classification = []
        num_in_clusters = 10000 / k
        for data in range(10000):
            classification.append(data // num_in_clusters)
        return np.array(classification, dtype = np.int)
```

initialization function initializes the classification of data points. The data points are classified into k clusters.
In this function, I use two different method for initializing the classification, which are random and equal-divide. Random method is randomly classifying the data points into different clusters; Equal-Divide method is classifying the data points into clusters with equal size.

Calculate the kernel matrix

```
# the implementation of new kernel which is basically multiplying two RBF
kernels
def make_kernel(image, gamma_s, gamma_c):
    color_similarity = squareform(pdist(image, 'sqeuclidean'))

    coordinate = []
    for row in range(100):
        for col in range(100):
            coordinate.append((row, col))
    coordinate = np.array(coordinate)
    spatial_similarity = squareform(pdist(coordinate, 'sqeuclidean'))

    kernel = np.exp(-gamma_s * spatial_similarity) * np.exp(-gamma_c *
color_similarity)

    return kernel

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

make_kernel function uses the input image, gamma_s and gamma_c to calculate the kernel matrix. The gamma_s here is **1/10000**, and gamma_c is **1/65025**. The shape of kernel matrix will be (10000, 10000).
The formula of calculating kernel is as follow, which is basically multiplying two RBF kernels:

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

Classification

```
def classify(kernel, classification, k):
    cluster_sum = np.zeros(k, dtype=np.int)
    all_term = np.zeros((10000, k), dtype=np.float32)

    for i in range(10000):
        cluster_sum[classification[i]] += 1

    for cluster in range(k):
        all_term[:, cluster] += (1 / cluster_sum[cluster]) ** 2 * np.sum(
            kernel[classification == cluster][:, classification == cluster])
        all_term[:, cluster] -= (2 / cluster_sum[cluster]) * np.sum(kernel[:,
classification == cluster], axis=1)

    new_classification = np.argmin(all_term, axis=1)

    return new_classification
```

classify function first calculates the distance from data points to cluster centers and store it into all_term numpy array. The shape of all_term numpy array is (10000, cluster_num).
This function then uses the distance from data points to cluster centers to calculate the new classification and store it into new_classification numpy array. The shape of new_classification numpy array is (10000,).

Plot the result

```
def plot_result(result, images):
    result = result.reshape((100, 100))
    image = plt.imshow(result)
    images.append([image])
    return images
```

plot_result function uses the classification result to plot the result image with size (100, 100).
This function then append the result image into images list in order to plot the result GIF.

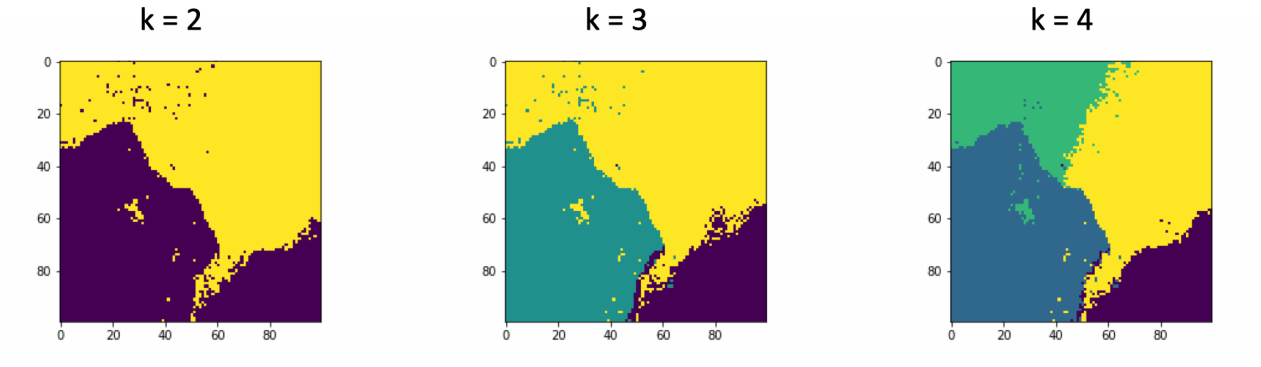
Implement the Kernel K-means algorithm

```
def kernel_k_means(image, k, name, gamma_s, gamma_c):
    methods = ['random', 'equal-divide']
    fig = plt.figure()
    for method in methods:
        images = []
        classification = initialization(k, method)
        kernel = make_kernel(image, gamma_s, gamma_c)

        while True:
            prev_classification = classification
            images = plot_result(classification, images)
            classification = classify(kernel, prev_classification, k)
            if np.array_equal(prev_classification, classification):
                break

        images = plot_result(classification, images)
        animate = animation.ArtistAnimation(fig, images, interval=500,
repeat_delay=1000)

animate.save(os.path.join(f'./{method}_{name}_{k}_kernel_k_means.gif'),
writer=PillowWriter(fps=20))

clusters = [2, 3, 4]
for cluster in clusters:
    kernel_k_means(image1, cluster, 'image1', gamma_s, gamma_c)
    kernel_k_means(image2, cluster, 'image2', gamma_s, gamma_c)
```

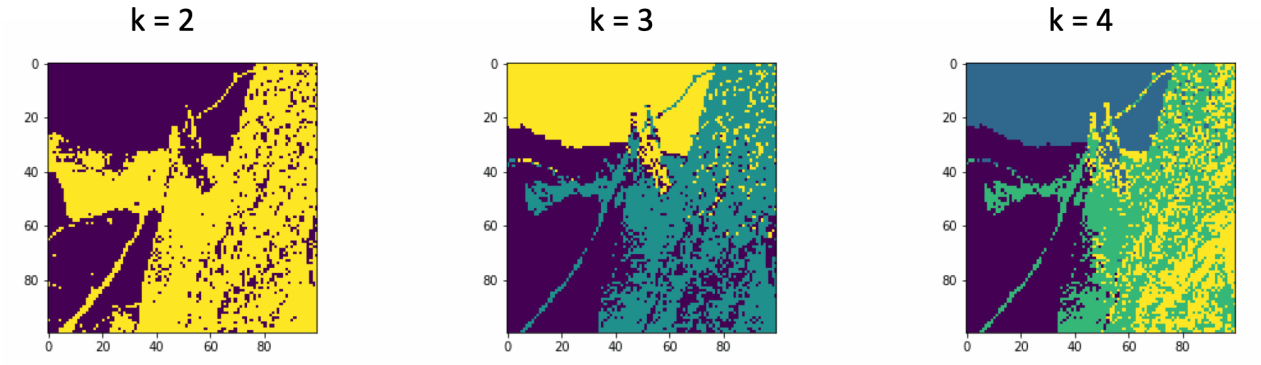
kernel_k_means function implements the kernel k-means algorithm with different cluster number. It will visualize the result by using matplotlib.animation package.

Result & Discussion

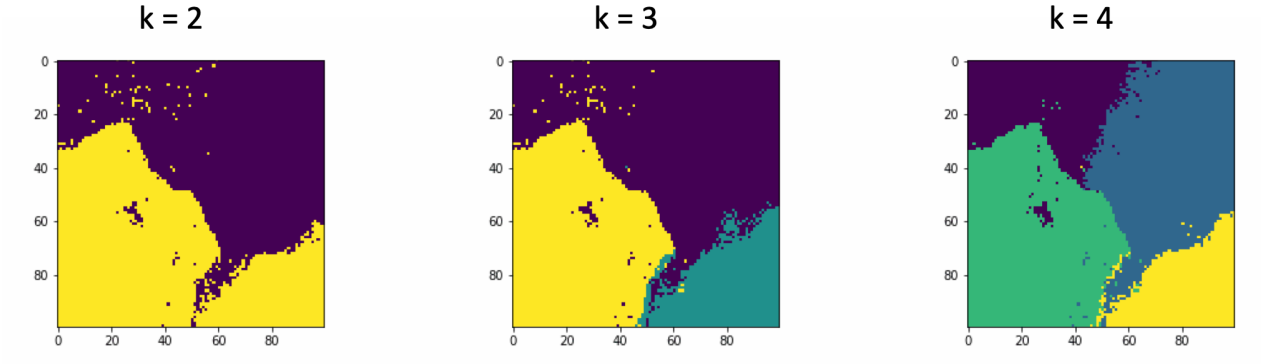
The classification result of image1 using random initialization method with cluster number k equals to 2, 3, and 4.



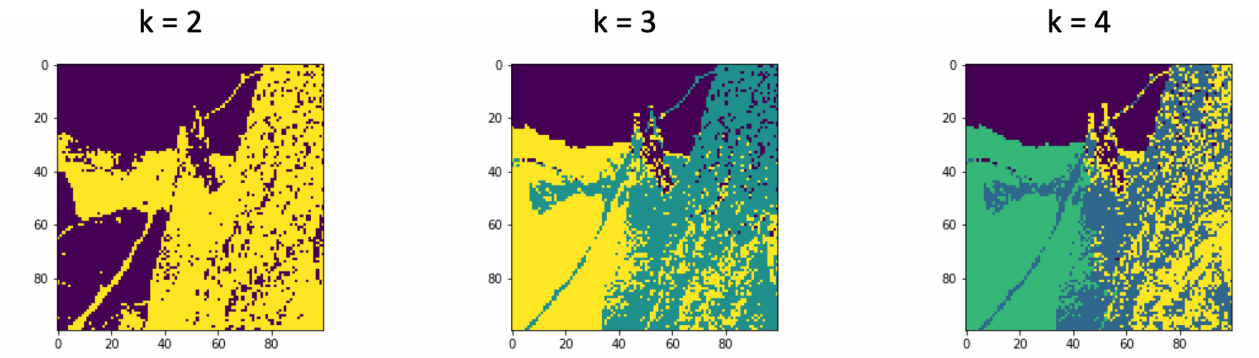
The classification result of image2 using random initialization method with cluster number k equals to 2, 3, and 4.



The classification result of image1 using equal-divide initialization method with cluster number k equals to 2, 3, and 4.



The classification result of image2 using equal-divide initialization method with cluster number k equals to 2, 3, and 4.



- Number of Clusters: I tried 2, 3, 4 cluster numbers. For **image1**, cluster number equals to 3 or 4 are better, and for **image2**, the result of cluster number equals to 4 is best among all. And is consistent with the original images.
- Initialization Method and Converge Speed: Training with Random mothod is faster than with Equal-Divide method. This phenomenon is especially obvious with cluster number equal to 3 or 4.

Spectral Clustering

Extract data & Normalization

```
# read the input images
def read_image(path):
    return cv2.imread(path).reshape((10000, 3))
```

read_image function transforms the input image into a 2-D numpy array. The shape of this numpy array is (10000, 3).

Initialize the classification

```
def nearest(data_point, cluster_centers):
    min_dist = 1e100
    cluster_num = np.shape(cluster_centers)[0]
    for i in range(cluster_num):
        dist = np.linalg.norm(data_point - cluster_centers[i, :])
        # choose the shortest distance
        if min_dist > dist:
            min_dist = dist
    return min_dist

def initialization(Laplacian, k, method):
    C = np.array([np.random.randint(100, size=k) for _ in range(k)], dtype =
np.float32)
    if method == 'random':
        classification = np.random.randint(k, size=10000)
```

```
        return C, classification
    elif method == 'k-means++':
        classification = np.random.randint(k, size=10000)
        data_num, dimension = np.shape(Laplacian)
        first_cluster = np.random.randint(data_num, size=1, dtype=np.int)
        C[0, :] = Laplacian[first_cluster, :]
        for i in range(1, k):
            distance = np.zeros(data_num, dtype=np.float32)
            for j in range(data_num):
                distance[j] = nearest(Laplacian[j, :], C[0:i, :])
            distance = distance / distance.sum()
            cluster = np.random.choice(data_num, size=1, p=distance)
            C[i, :] = Laplacian[cluster, :]
        return C, classification
```

initialization function initializes the classification of data points and cluster centers. The data points are classified into k clusters.
In this function, I use two different method for initializing the classification and cluster centers, which are random and k-means++. Random method initializes the cluster centers randomly and classifies the data points into these clusters; K-Means++ method Initializes the cluster centers far away from each other.

nearest function calculates the distance from data points to cluster centers.

Calculate the kernel matrix

```
# the implementation of new kernel which is basically multiplying two RBF
kernels
def make_kernel(image, gamma_s, gamma_c):
    color_similarity = squareform(pdist(image, 'sqeuclidean'))

    coordinate = []
    for row in range(100):
        for col in range(100):
            coordinate.append((row, col))
    coordinate = np.array(coordinate)
    spatial_similarity = squareform(pdist(coordinate, 'sqeuclidean'))

    kernel = np.exp(-gamma_s * spatial_similarity) * np.exp(-gamma_c *
color_similarity)

    return kernel

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

make_kernel function uses the input image, gamma_s and gamma_c to calculate the kernel matrix. The gamma_s here is **1/10000**, and gamma_c is **1/65025**. The shape of kernel matrix will be (10000, 10000).
The formula of calculating kernel is as follow, which is basically multiplying two RBF kernels:

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

Classification

```
def classify(Laplacian, C, k):
    new_classification = np.zeros(10000, dtype=np.int)
    distance = np.zeros(k, dtype=np.float)

    for data in range(10000):
        for cluster in range(k):
            delta = abs(np.subtract(Laplacian[data, :], C[cluster, :]))
            distance[cluster] = np.square(delta).sum(axis=0)
        new_classification[data] = np.argmin(distance)

    return new_classification
```

classify function first calculates the distance from data points to cluster centers and store it into distance numpy array.
This function then uses the distance from data points to cluster centers to calculate the new classification and store it into new_classification numpy array. The shape of new_classification numpy array is (10000,).

Update the cluster centers

```
def update_C(Laplacian, classification, k):
    new_C = np.zeros((k, k), dtype=np.float32)
    cluster_sum = np.zeros((k, k), dtype=np.int)
    for data in range(10000):
        cluster_sum[classification[data]] += np.ones(k, dtype=np.int)
        new_C[classification[data]] += Laplacian[data]
    for cluster in range(k):
        if cluster_sum[cluster][0] == 0:
            cluster_sum[cluster] += np.ones(k, dtype=np.int)
    new_C = np.true_divide(new_C, cluster_sum)

    return new_C
```

update_C function updates the cluster centers coordinate by using new classification result and store it into new_C numpy array. The shape of new_C numpy array is (k, k).

Plot the result

```
def plot_result(result, images):
    result = result.reshape((100, 100))
    image = plt.imshow(result)
    images.append([image])

    return images
```

plot_result function uses the classification result to plot the result image with size (100, 100). This function then append the result image into images list in order to plot the result GIF.

Plot the Eigenspace

```
def plot_eigenspace(Laplacian, classification, k, method, way, name):
    colors = ['red', 'green', 'blue']
    plt.clf()
    plt.title(f'{method}_{name}_{k}_{way}_eigenspace')
    if k == 2:
        for cluster in range(k):
            for data in range(10000):
                if classification[data] == cluster:
                    plt.scatter(Laplacian[data][0], Laplacian[data][1], s=8,
c=colors[cluster])
    elif k == 3:
        ax = plt.figure().add_subplot(111, projection='3d')
        for cluster in range(k):
            for data in range(10000):
                if classification[data] == cluster:
                    ax.scatter(Laplacian[data][0], Laplacian[data][1],
Laplacian[data][2], c=colors[cluster], marker='o')
    else:
        return
    plt.savefig(os.path.join(f'./{method}_{name}_{k}_{way}_eigenspace.png'))
```

plot_eigenspace function uses the classification result to plot the data points onto the eigenspace. Because the matplotlib package only can plot to at most 3-D image, I don't plot the data points onto the eigenspace with four dimension.

Implement the K-means algorithm

```
def k_means(Laplacian, image, k, name, way):
    methods = ['random', 'k-means++']
    images = []
    fig = plt.figure()
    for method in methods:
        C, classification = initialization(Laplacian, k, method)
        iteration = 0
```

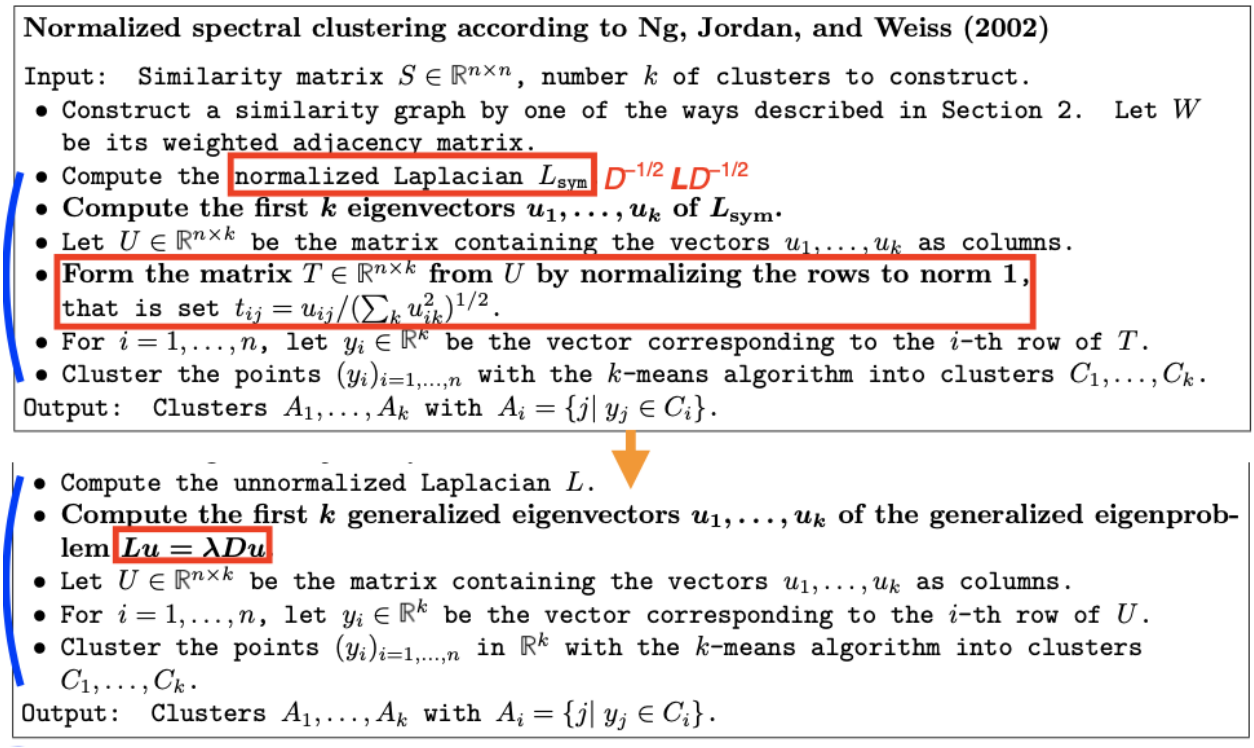
```
        while True:
            prev_classification = classification
            iteration += 1
            images = plot_result(classification, images)
            classification = classify(Laplacian, C, k)
            if np.array_equal(prev_classification, classification) or
iteration >= 100:
                break

            C = update_C(Laplacian, classification, k)

        images = plot_result(classification, images)
        animate = animation.ArtistAnimation(fig, images, interval=500,
repeat_delay=1000)
        animate.save(os.path.join(f'./{method}_{name}_{k}_{way}.gif'),
writer=PillowWriter(fps=20))

        plot_eigenspace(Laplacian, classification, k, method, way, name)
```

k_means function implements the k-means algorithm with different cluster number and initialization method. It will visualize the result by using matplotlib.animation package.

Implement the Spectral Clustering with Normalize Cut

```
def normalize_cut(image, k, name):
    weight = make_kernel(image, gamma_s, gamma_c)
    degree = np.diag(np.power(np.sum(weight, axis=1), -0.5))
    L_sym = np.eye(10000) - np.dot(degree, np.dot(weight, degree))
    eigen_values, eigen_vectors = np.linalg.eig(L_sym)
    U = eigen_vectors[:, np.argsort(eigen_values)[1: k +
1]].real.astype(np.float32)

    # normalization
    T = U.copy()
    for row in range(10000):
        sum = np.sum(np.power(U[row], 2)) ** 0.5
        if sum != 0:
            T[row, :] /= sum

    k_means(T, image, k, name, 'normalize_cut')

clusters = [2, 3, 4]
for cluster in clusters:
    normalize_cut(image1, cluster, 'image1')
    normalize_cut(image2, cluster, 'image2')
```

normalize_cut function implements the spectral clustering with normalize cut algorithm by using following pseudo code:



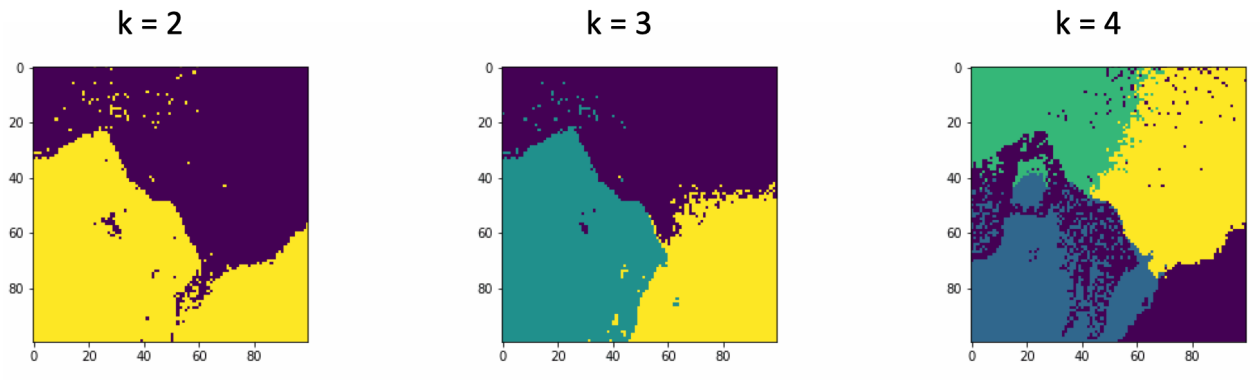
Implement the Spectral Clustering with Radio Cut

```
def radio_cut(image, k, name):
    weight = make_kernel(image, gamma_s, gamma_c)
    degree = np.diag(np.sum(weight, axis=1))
    L = degree - weight
    eigen_values, eigen_vectors = np.linalg.eig(L)
    U = eigen_vectors[:, np.argsort(eigen_values)[1: k +
1]].real.astype(np.float32)

    k_means(U, image, k, name, 'radio_cut')

clusters = [2, 3, 4]
for cluster in clusters:
    radio_cut(image1, cluster, 'image1')
    radio_cut(image2, cluster, 'image2')
```

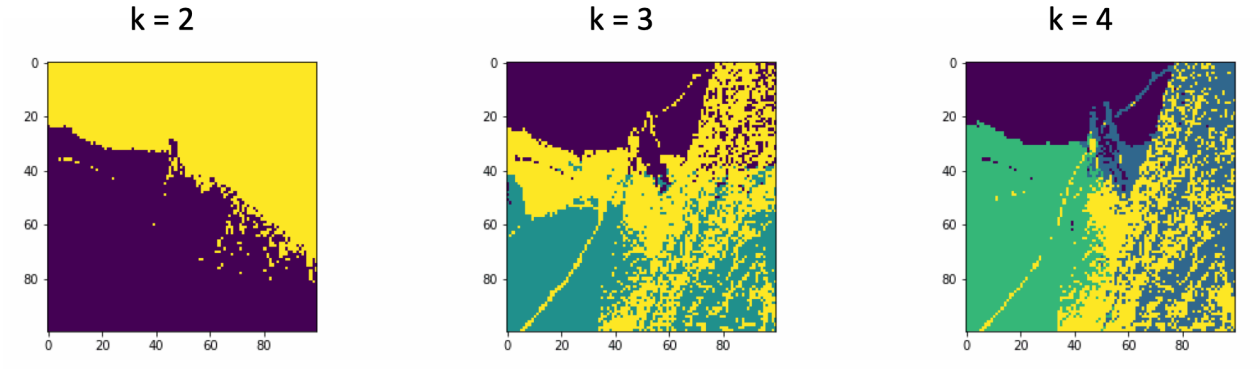
radio_cut function implements the spectral clustering with radio cut algorithm.

Result & Discussion

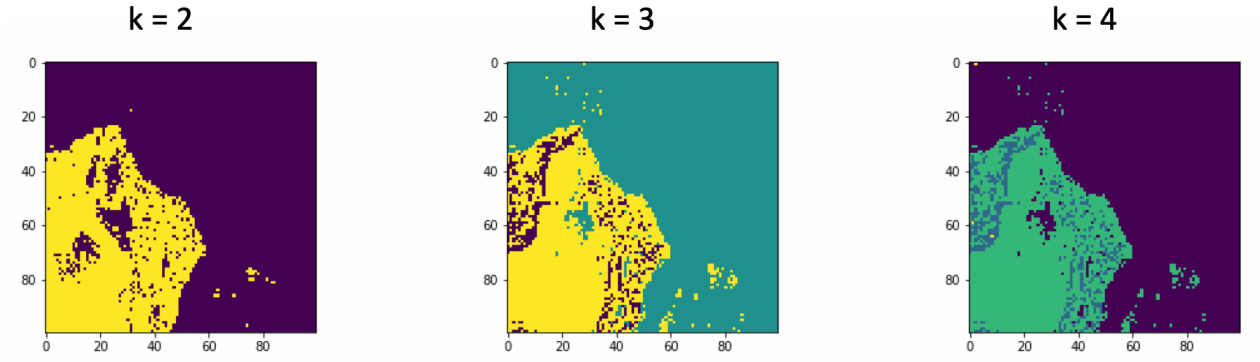
The classification result of image1 using random initialization method and normalize cut with cluster number k equals to 2, 3, and 4.



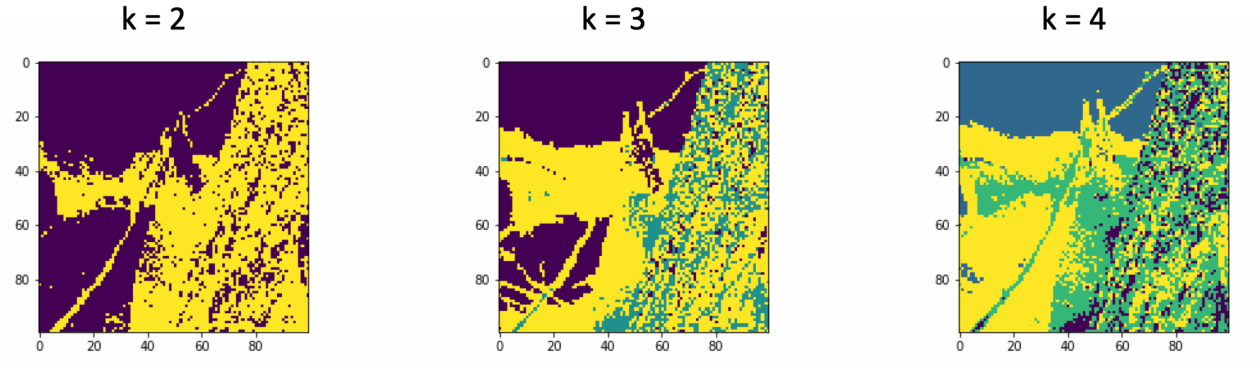
The classification result of image2 using random initialization method and normalize cut with cluster number k equals to 2, 3, and 4.



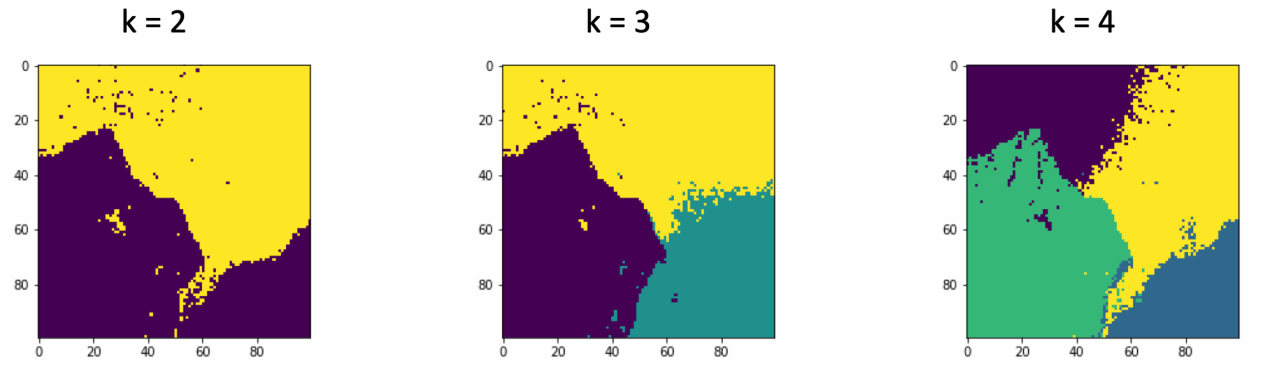
The classification result of image1 using random initialization method and radio cut with cluster number k equals to 2, 3, and 4.



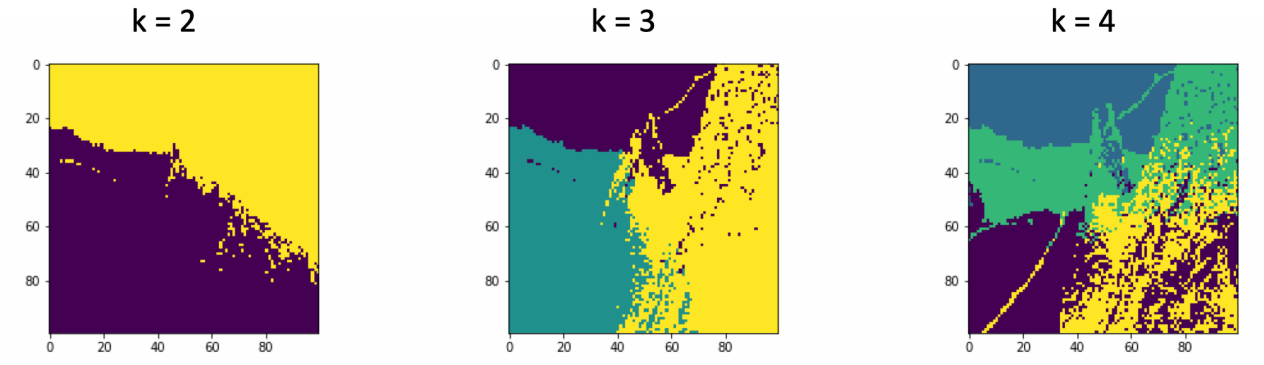
The classification result of image2 using random initialization method and radio cut with cluster number k equals to 2, 3, and 4.



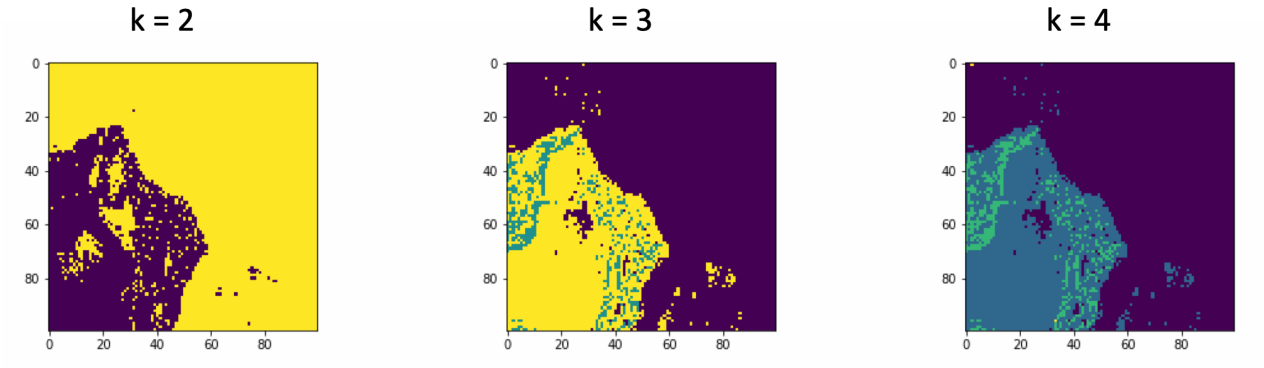
The classification result of image1 using k-means++ initialization method and normalize cut with cluster number k equals to 2, 3, and 4.



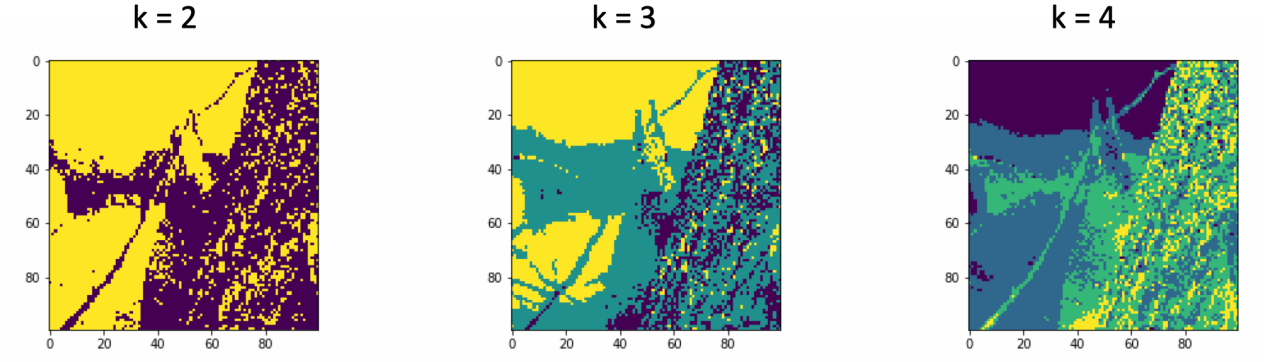
The classification result of image2 using k-means++ initialization method and normalize cut with cluster number k equals to 2, 3, and 4.



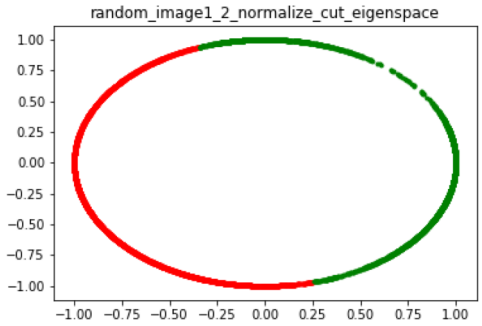
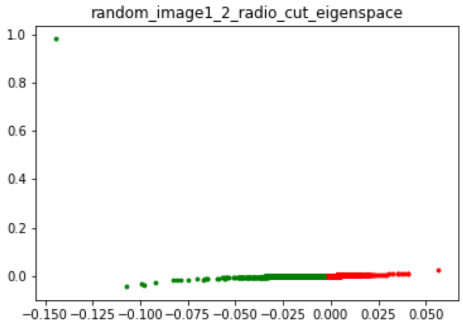
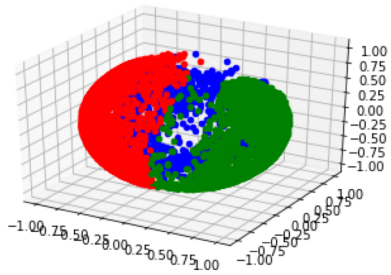
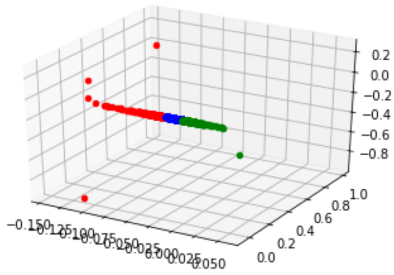
The classification result of image1 using k-means++ initialization method and radio cut with cluster number k equals to 2, 3, and 4.



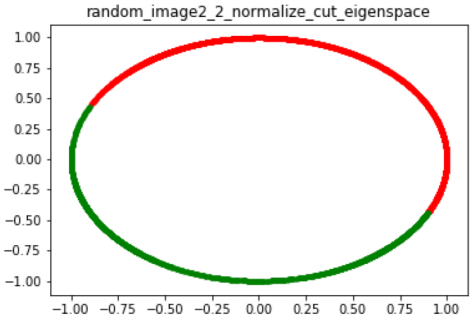
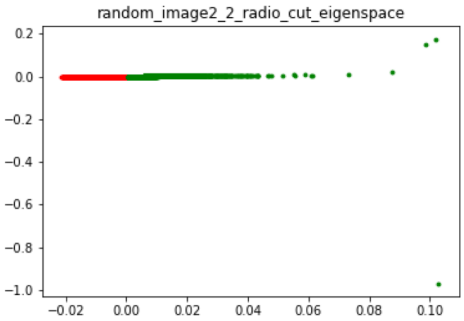
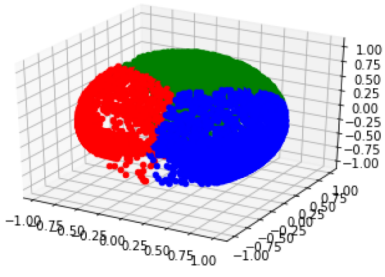
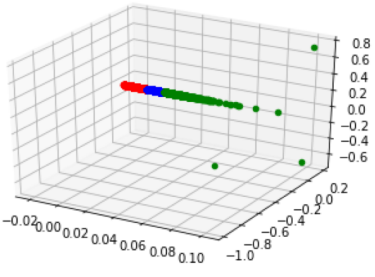
The classification result of image2 using k-means++ initialization method and radio cut with cluster number k equals to 2, 3, and 4.



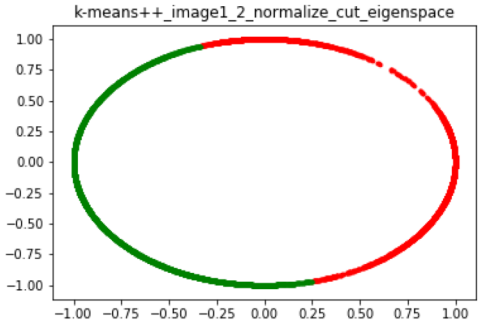
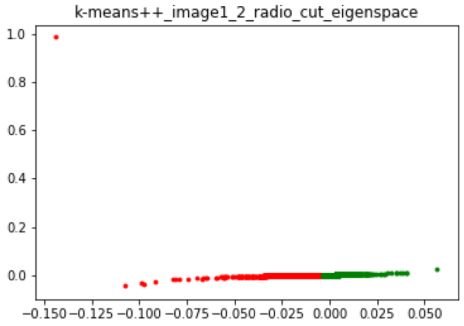
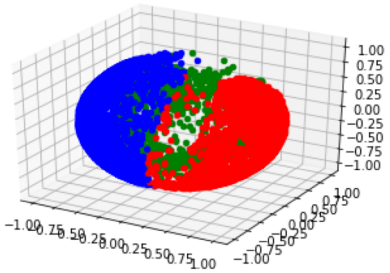
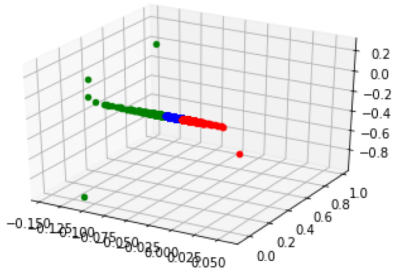
The data points of image1 on the eigenspace using random initialization method.

	Normalize Cut	Radio Cut
k = 2		
k = 3		

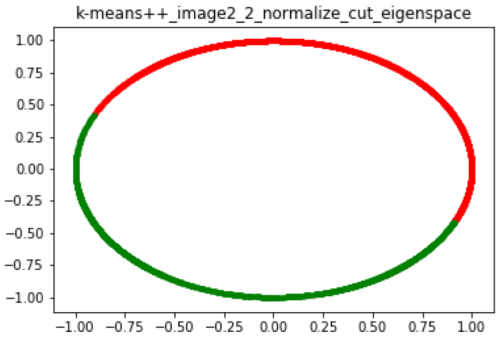
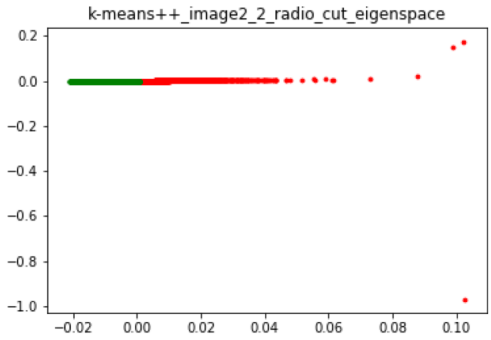
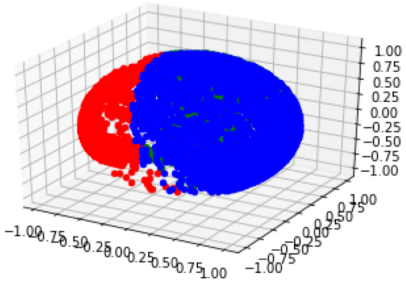
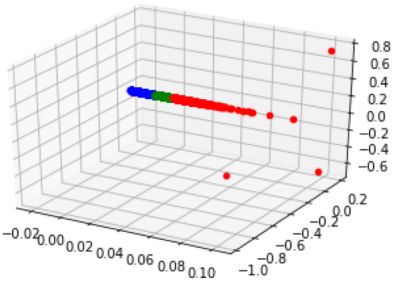
The data points of image2 on the eigenspace using random initialization method.

	Normalize Cut	Radio Cut
k = 2		
k = 3		

The data points of image1 on the eigenspace using k-means++ initialization method.

	Normalize Cut	Radio Cut
k = 2		
k = 3		

The data points of image2 on the eigenspace using k-means++ initialization method.

	Normalize Cut	Radio Cut
k = 2		
k = 3		

Observing data points on eigenspace, Normalized Cut looks more cubic than that of Ratio Cut. As a result, Normalized Cut takes more time to converge.

Appendix

In this part I will explain my GIF naming principle and directory structure.
The directory structure should look like as following:

- Result
  - Kernel_K_means
    - Random
      - GIFs in format of random*{Image_Number}* {Cluster_Number}_kernel_k_means.gif
    - Equal_Divide
      - GIFs in format of equal_divide*{Image_Number}* {Cluster_Number}_kernel_k_means.gif
  - Spectral_Clustering
    - Random
      - Normalize_Cut
        - GIFs in format of random*{Image_Number}* {Cluster_Number}_normalize_cut.gif
      - Radio_Cut
        - GIFs in format of random*{Image_Number}*{Cluster_Number}_radio_cut.gif
      - Eigenspace
        - Images in format of random*{Image_Number}* {Cluster_Number}_radio_cut_eigenspace.png
    - K_means++
      - Normalize_Cut
        - GIFs in format of k_means_plus*{Image_Number}* {Cluster_Number}_normalize_cut.gif
      - Radio_Cut
        - GIFs in format of k_means_plus*{Image_Number}* {Cluster_Number}_radio_cut.gif
      - Eigenspace
        - Images in format of k_means_plus*{Image_Number}* {Cluster_Number}_radio_cut_eigenspace.png