# Machine Learning HW6

## Kernel K-means and Spectral Clustering

GMBA 310712009 楊家碩

# a. code with detailed explanations

# I. Kernel K-means

## Implementation

I implement kernel k-means in python. The essential steps in this section are

1. Generate a Gram Matrix using the kernel function as shown below, and
2. Complete the K-means method.
3. Display gif in the visualize function.

I set gamma_C = 1e-5 and gamma_S = 1e-5. At the same time, I also set k list "K_lis = [2,3,4,5,6,8,10,12,15]", to find out the result with different k and how the image will look like.

```
imagename = 'image1'
K_lis = [2,3,4,5,6,8,10,12,15]
mode_lis = [0,1]
for i in K_lis:
    for j in mode_lis:
        gamma_C = 1e-5
        gamma_S = 1e-5

        filename = img1
        k = i
        mode = j

        dataC, dataS, image_size = read_input(filename)
        Gram_K = compute_kernel(gamma_S, gamma_C, dataS, dataC)

        datapoint_his = k_means(Gram_K, k, mode)
        visualplot(datapoint_his, image_size, storename1, k, mode, imagename)
```

### The read_input function:

```
def read_input(filename):
    image = Image.open(filename)
    data = np.array(image)
    # color data: RGB for each pixel (10000, 3)
    dataC = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
    # spatial data: coordinate for each pixel
    dataS = np.array([(i,j) for i in range(data.shape[0]) for j in range(data.shape[1])])
    return dataC, dataS, image.size
```

First, we need to load the image files.

### The compute_kernel function:

```python
def compute_kernel(gammaS, gammaC, S, C):
    result_k = np.exp(-gammaS*cdist(S, S, 'sqeuclidean'))
    result_k *= np.exp(-gammaC*cdist(C, C, 'sqeuclidean'))
    return result_k
```

Formula: $k(x, x') = e^{-\gamma_S \|S(x) - S(x')\|^2} * e^{-\gamma_C \|C(x) - C(x')\|^2}$

I set gamma_C = 1e-5 and gamma_S = 1e-5. Then, to calculate the norm by using Euclidean distance.

### The initial_data function:

```python
def initial_data(Gram_K, k, mode):
    mean_k = np.zeros((k, Gram_K.shape[1]), dtype=Gram_K.dtype)

    # normal k-means -> random center
    if mode == 0:
        center =  np.array(random.sample(range(0, 10000), k))
        mean_k = Gram_K[center,:]

    # k-means++
    elif mode == 1:
        mean_k[0] = Gram_K[np.random.randint(Gram_K.shape[0], size=1), :]
        for cluste_id in range(1, k):
            temp_dist = np.zeros((len(Gram_K), cluste_id))
            for i in range(len(Gram_K)):
                for j in range(cluste_id):
                    temp_dist[i][j] = np.linalg.norm(Gram_K[i]-mean[j])
            dist = np.min(temp_dist, axis=1)
            sum = np.sum(dist) * np.random.rand()
            for i in range(len(Gram_K)):
                sum -= dist[i]
                if sum <= 0:
                    mean_k[cluste_id] = Gram_K[i]
                    break
    return mean_k
```

Depending on the input mode, this procedure generates k-means as the beginning points.

- I set “0” as normal k-means, which gives random k-means
- then, “1” is for k-means++, which tries to let each mean be "far" enough.

### The k_means function:

```
def k_means(Gram_K, k, mode):
    datapoint_his = []

    mean = initial_data(Gram_K, k, mode)
    old_mean = np.zeros(mean.shape, dtype=Gram_K.dtype)
    while np.linalg.norm(mean - old_mean) > 1e-10:
        # E-step: classify all samples
        clusters = np.zeros(Gram_K.shape[0], dtype=int)
        for i in range(Gram_K.shape[0]):
            J = []
            for j in range(k):
                J.append(np.linalg.norm(Gram_K[i] - mean[j]))
            clusters[i] = np.argmin(J)
        datapoint_his.append(clusters)

        # M-step: Update center mean
        old_mean = mean
        mean = np.zeros(mean.shape, dtype=Gram_K.dtype)
        counters = np.zeros(k)
        for i in range(Gram_K.shape[0]):
            mean[clusters[i]] += Gram_K[i]
            counters[clusters[i]] += 1
        for i in range(k):
            if counters[i] == 0:
                counters[i] = 1
            mean[i] /= counters[i]
    print("Total No. of iteration(s):", len(datapoint_his))
    return datapoint_his
```

The primary purpose of performing k-means is this. Each iteration's clusters are stored in the list history. Each k-means iteration can be divided into an E-step and an M-step.

- Sort all data points in E-step by the mean of the nearest data center.
- Update the new data center in M-step in accordance with the E-step outcome.
- Up until the means are covered, use k-means.

### The visualplot function:

```
def visualplot(datapoint_his, image_size, storename, k, mode, imagename):
    gif = []
    color = [ImageColor.getrgb('darkorange'), ImageColor.getrgb('navy'), ImageColor.getrgb('Brown'), ImageColor.getrgb('greenyellow'),
             ImageColor.getrgb('purple'), ImageColor.getrgb('silver'), ImageColor.getrgb('gold'), ImageColor.getrgb('MediumAquamarine'),
             ImageColor.getrgb('black'), ImageColor.getrgb('magenta'), ImageColor.getrgb('peru'), ImageColor.getrgb('green'),
             ImageColor.getrgb('yellow'), ImageColor.getrgb('pink'), ImageColor.getrgb('red') #cyan, dodgerblue, cornflowerblue
             ]

    iteration = len(datapoint_his)
    for i in range(iteration):
        gif.append(Image.new("RGB", image_size))
        for y in range(image_size[0]):
            for x in range(image_size[1]):
                gif[i].putpixel((x, y), color[datapoint_his[i][y*image_size[0]+x]])

    gif[0].save(storename + f"k_means_gif_{imagename}_mode{mode}_k{k}.gif",
                format='GIF',
                save_all=True,
                append_images=gif[1:],
                duration=400, loop=0)
    gif[-1].save(storename + f"k_means_pic_{imagename}_mode{mode}_K{k}.jpg", format='JPEG')
```

I utilize historical data points as input to build the plot and hand-pick those colors to show the data in a better way for us to compare. I then use PIL to create a picture and save the resulting gif file.

# II. Spectral clustering

This section's primary process is to:

1. We will generate a gram matrix using the compute_kernel function as the k-means portion and then
2. Produce Graph Laplacian L based on the chosen cut.
3. To obtain the U matrix, calculate the eigenvalue and eigenvector. And for normalized cut, we must create the matrix T from U by bringing the rows up to norm 1
4. Plot the image to show the result

I continue to use the read_input function, initial_data function and k_means function from the Kernel K-means. We need to add two other functions in order to do Spectral clustering.

## Unnormalized Laplacian (ratio cut)

### The ratio_cut function:

```
def ratio_cut(pixel, coord):
    weight = compute_kernel(pixel, coord) #W
    degree = np.diag(np.sum(weight, axis=1)) #D
    L = degree - weight # L = D-W

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

    return U
```

Formula: $L = D - W$

Take the degree matrix and the similarity matrix apart. And to express the similarity matrix, we utilize the kernel matrix. The formula and how it is used are shown.

# Normalized Laplacian (normalized cut)

## The normalized_cut function:

```
def normalized_cut(pixel, coord):
    weight = compute_kernel(pixel, coord) #W
    degree = np.diag(np.sum(weight, axis=1)) #D

    degree_square = np.diag(np.power(np.diag(degree), -0.5))
    L_sym = np.eye(weight.shape[0]) - degree_square @ weight @ degree_square #L
    eigen_values, eigen_vectors = np.linalg.eig(L_sym)
    idx = np.argsort(eigen_values)[1: K+1]
    U = eigen_vectors[:, idx].real.astype(np.float32)

    # normalized
    sum_over_row = (np.sum(np.power(U, 2), axis=1) ** 0.5).reshape(-1, 1)
    T = U.copy()
    for i in range(sum_over_row.shape[0]):
        if sum_over_row[i][0] == 0:
            sum_over_row[i][0] = 1
        T[i][0] /= sum_over_row[i][0]
        T[i][1] /= sum_over_row[i][0]

    return T
```

Formula: $L = I - D^{-1} W D^{-1}$

The formula can be used to create a Laplacian matrix, and the illustration that results shows how it is done.

1. First, Calculate the first k eigenvectors of L, where k is the number of clusters,
2. Second, use those k eigenvectors to build the matrix U. Here, np.linalg.eig is used to assist in computing the outcome.
3. Third, for normalized cut, we must create the matrix T from U by bringing the rows up to norm 1. We can use the formula below to implement the matrix T.

$$T \in \mathbb{R}^{n*k} \quad where \;\; t_{ij} = \frac{u_{ij}}{(\sum_k u_{ik}^2)^{(1/2)}}$$

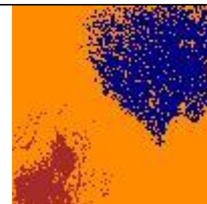
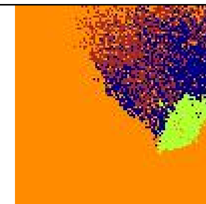
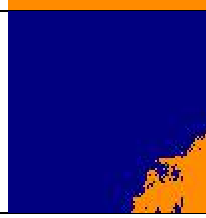
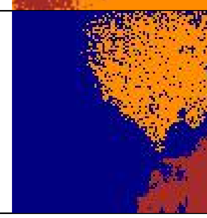
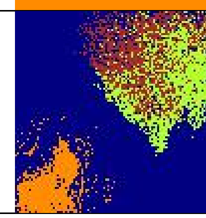
Then we can do the visualization with our visualplot function below.

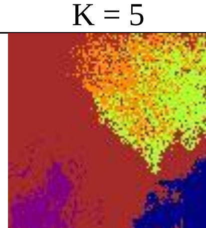
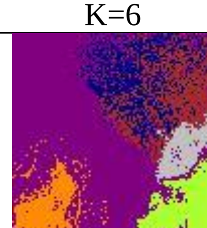
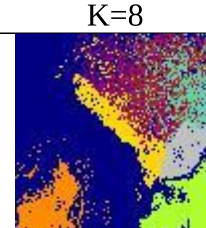
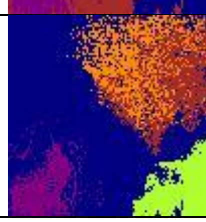
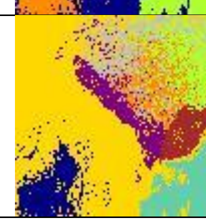
```
def visualplot(filename, storename, iteration, classification, initial_method):
    img = Image.open(filename)
    width, height = img.size
    pixel = img.load()
    color = [ImageColor.getrgb('darkorange'), ImageColor.getrgb('navy'), ImageColor.getrgb('Brown'), ImageColor.getrgb('greenyellow'),
             ImageColor.getrgb('purple'), ImageColor.getrgb('silver'), ImageColor.getrgb('gold'), ImageColor.getrgb('MediumAquamarine'),
             ImageColor.getrgb('black'), ImageColor.getrgb('magenta'), ImageColor.getrgb('peru'), ImageColor.getrgb('green'),
             ImageColor.getrgb('yellow'), ImageColor.getrgb('pink'), ImageColor.getrgb('red')
             ]
    for i in range(img.size[0]):
        for j in range(img.size[1]):
            pixel[j, i] = color[classification[i * num + j]]
    img.save(storename + '_' + initial_method + '_' + str(gamma_c) + '_' + str(gamma_s) + '_' + str(iteration) + '_'+ str(K) + '.png')
```

# b. experiments settings and results

## I. Result of Kernel K-means

image1.png

| | K = 2 | K=3 | K=4 |
|---|---|---|---|
| k-means |  |  |  |
| k-means++ |  |  |  |

| | K = 5 | K=6 | K=8 |
|---|---|---|---|
| k-means |  |  |  |
| k-means++ |  |  |  |

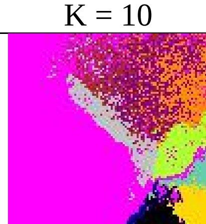
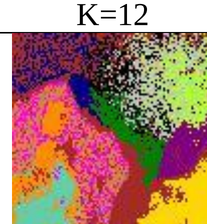
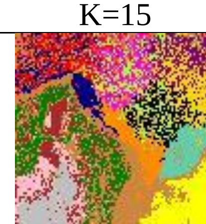
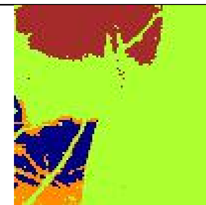
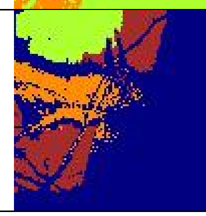
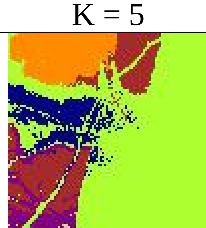
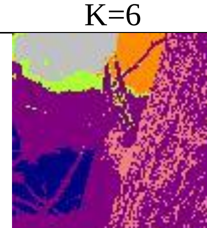
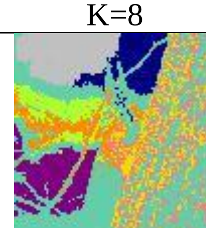
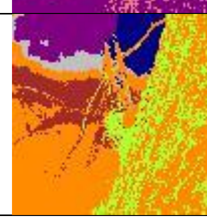
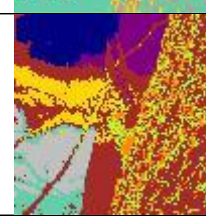
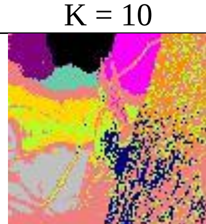
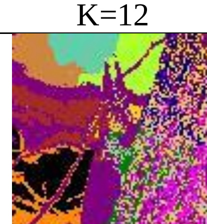
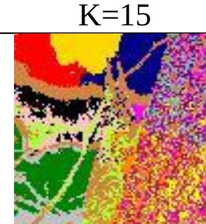
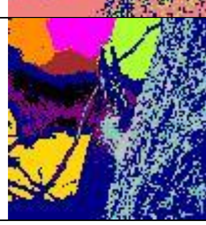
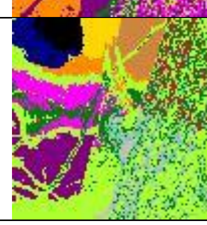
| | K = 10 | K=12 | K=15 |
|---|---|---|---|
| k-means |  |  |  |
| k-means++ |  |  |  |

Image2.png

| | K = 2 | K=3 | K=4 |
|---|---|---|---|
| k-means |  |  |  |
| k-means++ |  |  |  |

| | K = 5 | K=6 | K=8 |
|---|---|---|---|
| k-means |  |  |  |
| k-means++ |  |  |  |

| | K = 10 | K=12 | K=15 |
|---|---|---|---|
| k-means |  |  |  |
| k-means++ |  |  |  |

**Observation:**

Although k-means and k-means++ occasionally provide the same results, k-means++ has a higher level of stability because it begins with random means. K-means suffered from noise at the top of the image for k=4 in image1. K-means and K-means++ produce different results for k=3 in image1. Both of them, k-means for land and sea and k-means++ for dark and light sea, seems reasonable to me.

The fascinating thing is that if k=2, the clusters in image 1 will appear to resemble land and water to human sight. After experimenting with various initial strategies, however, it appears that the distinction between a dark and a light sea is greater than that between land and water. It might be a result of the kernel function we employ, which multiplies two PBF kernels with respect to color and spatial information. It forces k-means to take into account both the color information and the location of the color.

I also try some more clusters (K = 5, 6, 8, 10, 12, 15). As you can see, it can catch some more details but does not guarantee a better result in higher clusters. We still need to depend on the images to decide.

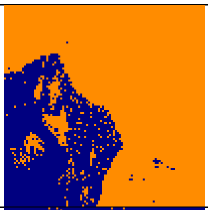
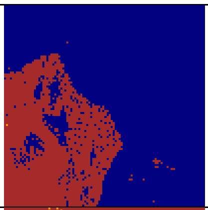
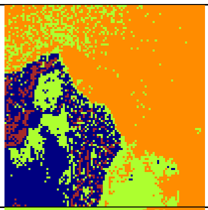
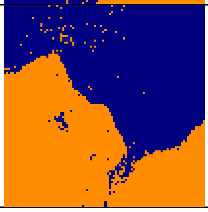
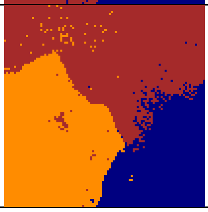
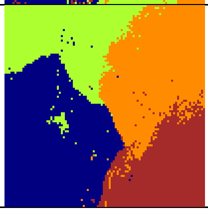
Kernel K-means took less time than Spectral clustering to come out with result.

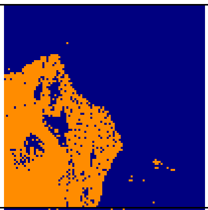
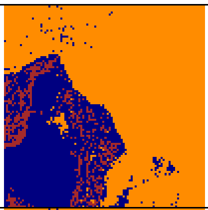
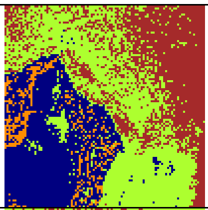
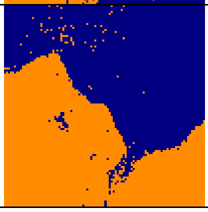
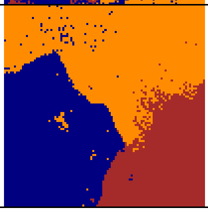
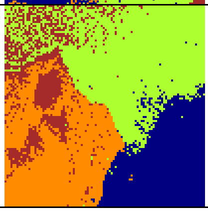
<u>k-means++ v.s. Random initial</u>

- Both initials may produce the same outcome. However, there is a greater likelihood that a random beginning may produce poor clustering.
- The program's initial random initial may have very close centers. As a result, it takes longer than k-means++ to converge to the final clustering.
- K-mean++ promises to locate distant initial centers. It is crucial to have centers situated apart from one another. Given this assignment's small amount of input data, it might not make a significant difference. Even with a worse beginning, the cost is still acceptable. However, having a solid start would make a significant impact in more complex computational or data applications. As a result, having k-mean++ rather than random is advantageous.

# II. Result of Spectral clustering

image1.png

| **random** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |  |  |  |
| normalized cut |  |  |  |

| **Random from data** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |  |  |  |
| normalized cut |  |  |  |

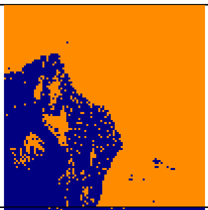
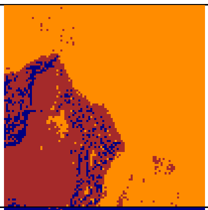
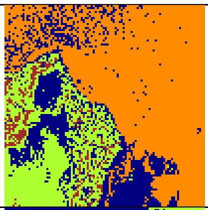
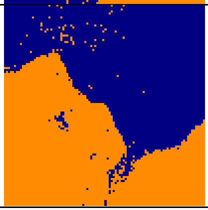
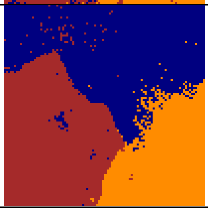
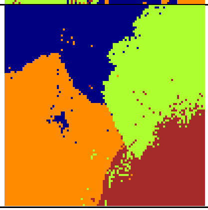
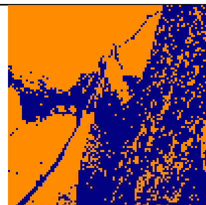
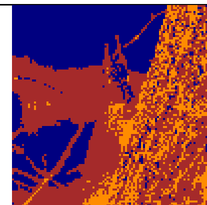
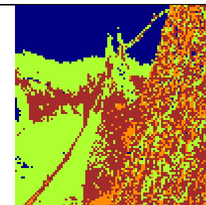
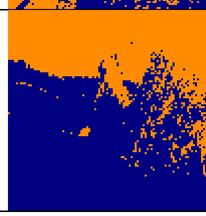
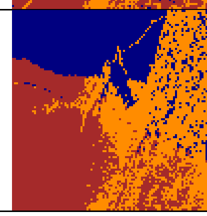
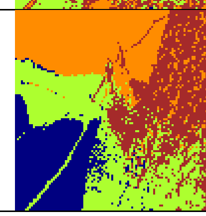
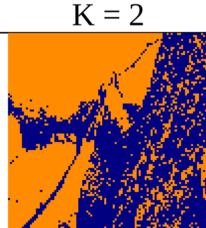
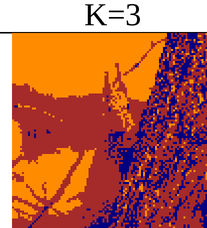
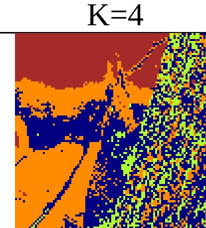
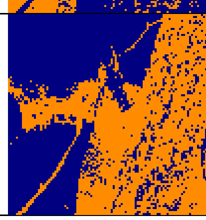
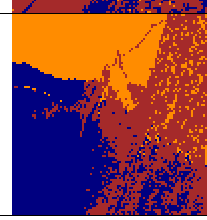
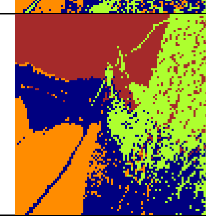
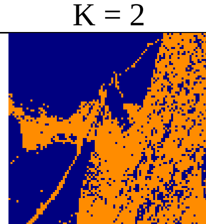
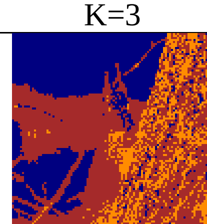
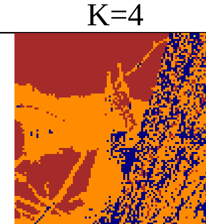
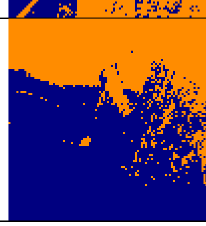
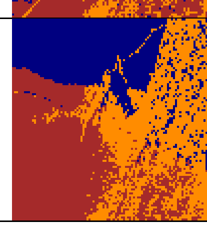
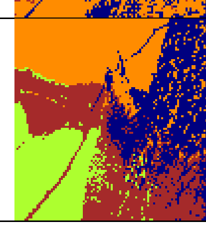
| **k-means++** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |  |  |  |
| normalized cut |  |  |  |

Image2.png

| **random** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |  |  |  |
| normalized cut |  |  |  |

| **Random from data** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |  |  |  |
| normalized cut |  |  |  |

| **k-means++** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |  |  |  |
| normalized cut |  |  |  |

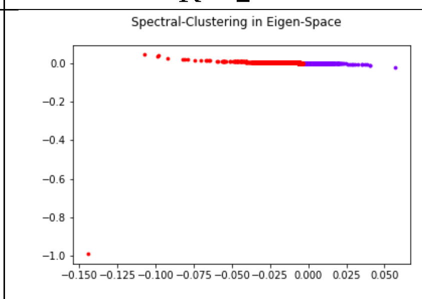
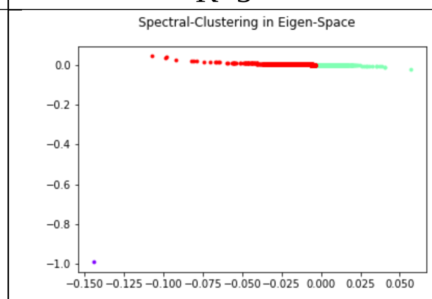
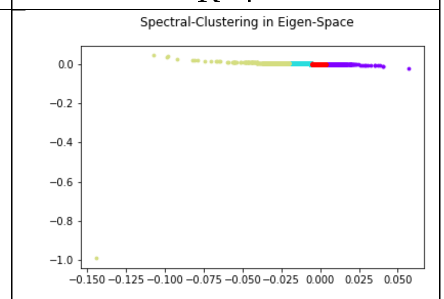
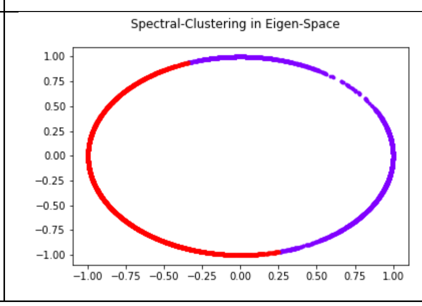
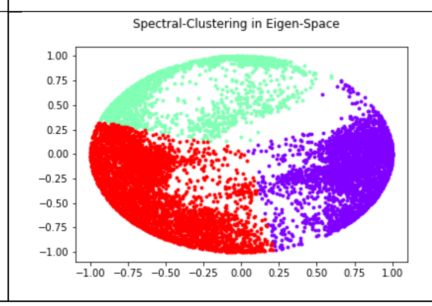
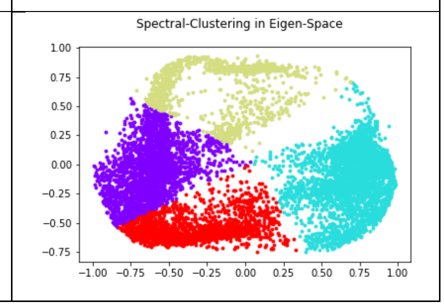
**Observation:**

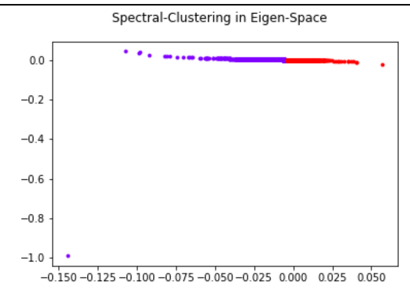
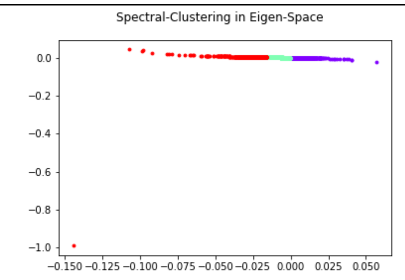
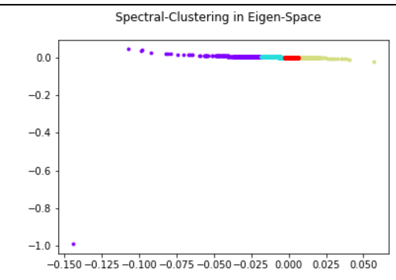
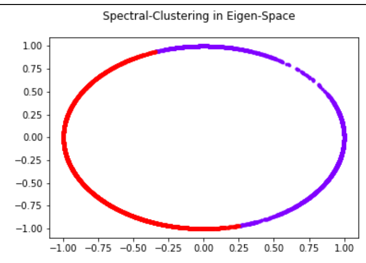
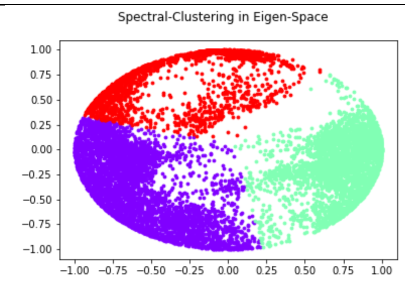
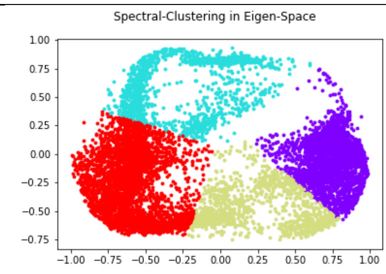
- The difference between a ratio cut and a normalized cut is less noticeable when the image is straightforward and has fewer boundaries.
- The difference between the ratio cut and the normalized cut increases with the number of clusters. Images 1 and 2 both show this to us.
- Similar to what we discovered in the k-means section, k-means is more likely to yield poor results because it begins with random means.
- Spectral clustering takes longer time than Kernel K-means but it can better catch the details by comparing the result on those graphs.

# Part4: coordinates in the eigenspace

image1.png

- 2-cluster (1st row is 1st-iteration, 2nd row is final result)

| **random** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |  |  |  |
| normalized cut |  |  |  |

| **Random from data** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |   |   |   |
| normalized cut |   |   |   |

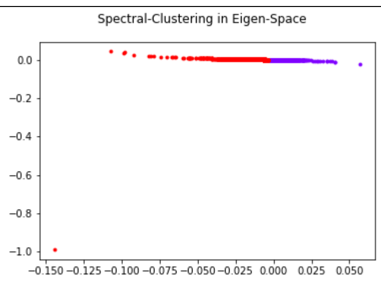
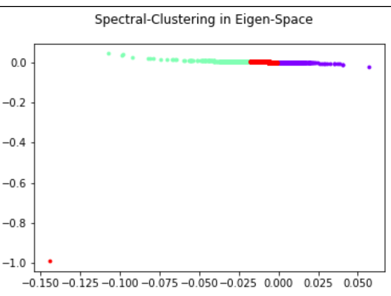
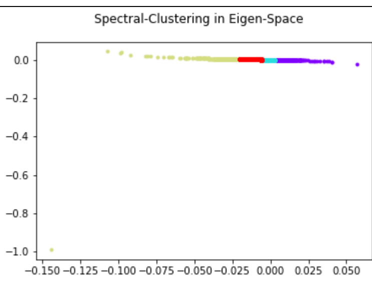
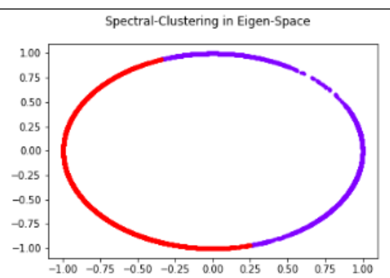
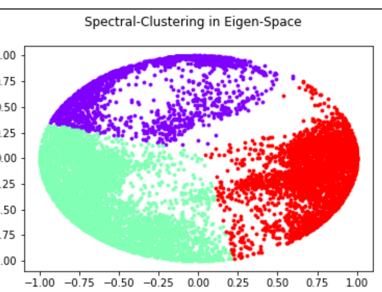
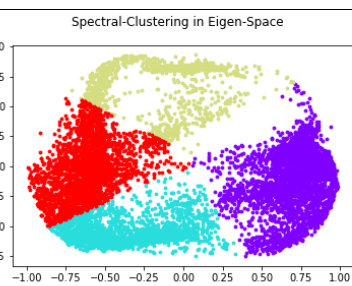
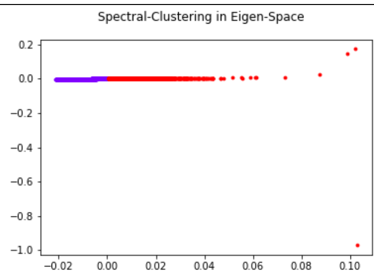
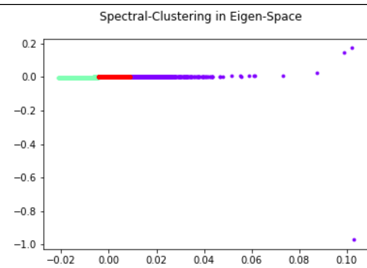
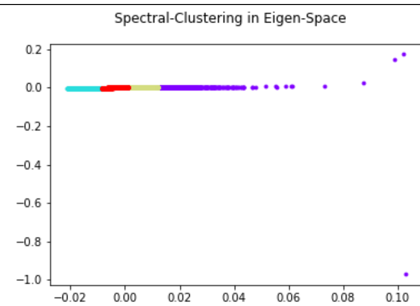
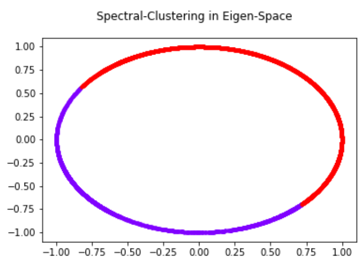
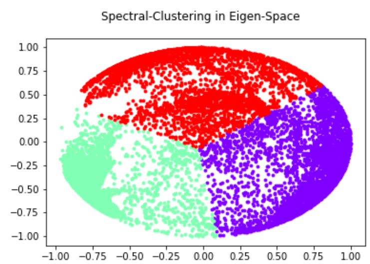
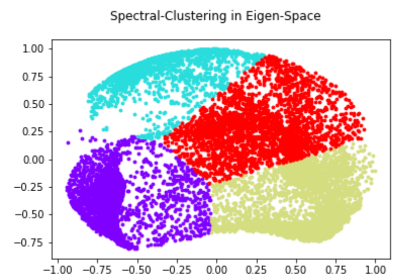
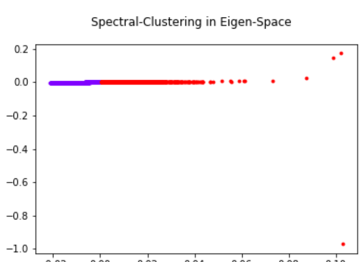
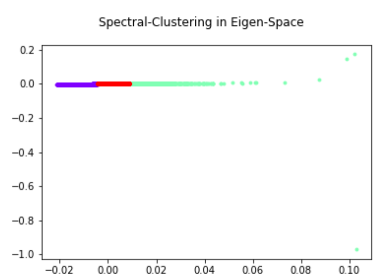
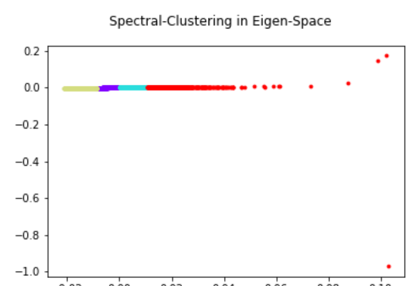
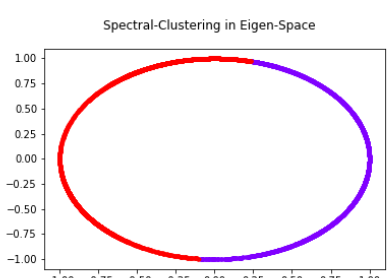
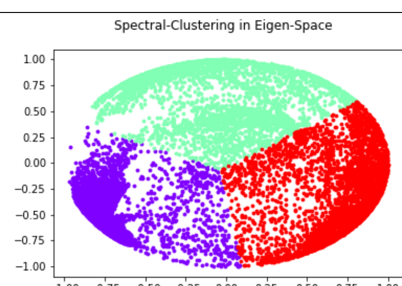
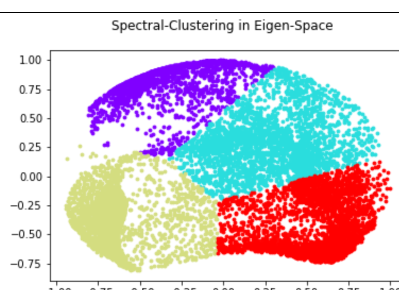
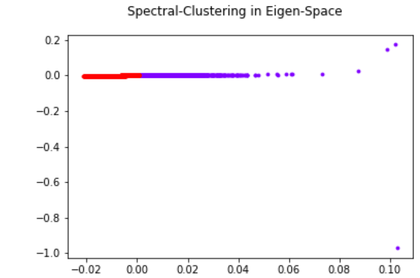
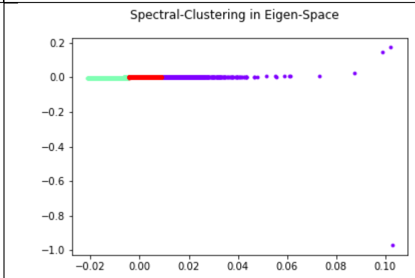
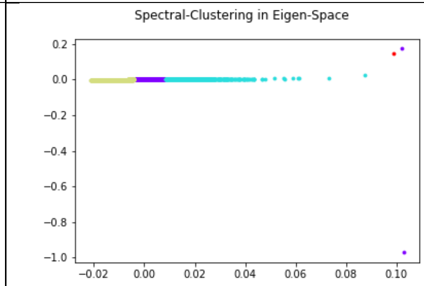
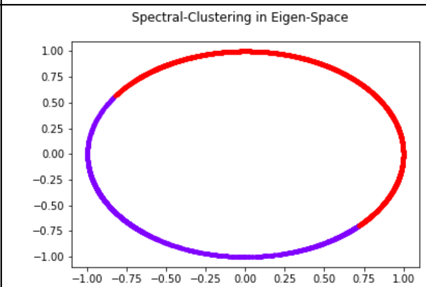
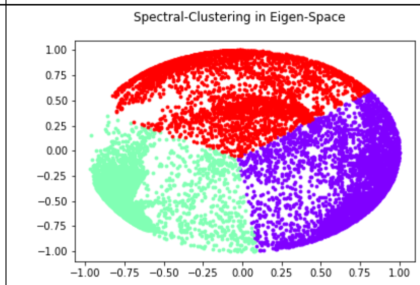
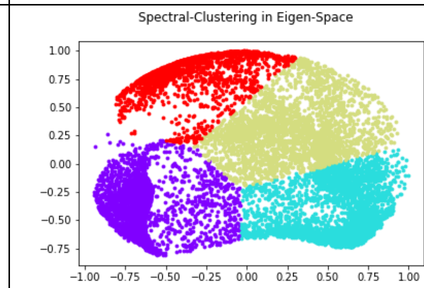
| **k-means++** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |   |   |   |
| normalized cut |   |   |   |

Image2.png

| **random** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |   |   |   |
| normalized cut |   |   |   |

| **Random from data** | K = 2 | K=3 | K=4 |
|---|---|---|---|
| ratio cut |   |   |   |
| normalized cut |   |   |   |

<table>
<tr><th><u>k-means++</u></th><th>K = 2</th><th>K=3</th><th>K=4</th></tr>
<tr><td>ratio cut</td><td>
</td><td>
</td><td>
</td></tr>
<tr><td>normalized cut</td><td>
</td><td>
</td><td>
</td></tr>
</table>

**<u>Observation:</u>**

- The eigenspace of the data in the same cluster is near. Their coordinates, however, are different.
- The charts for the ratio cut and the normalized cut are very dissimilar.
- I also test k-means graphs. The graphs produced by k-means are all comparable to those produced by k-means++ since their beginning means differ. Only the cluster outcomes are impacted; the coordinates of the data in eigenspace remain unaffected.