Machine Learning
HW6-Kernel K-means & Spectral Clustering

Yu-Ju Tsai, 311652010

May 30, 2023

1 Code with detailed explanations

First I load dataset

```python
def load_png(input):
    img = Image.open('data/' + input)
    img = np.array(img.getdata()) #(10000,3)
    return img
```

Figure 1: Load Data

- Two RBF kernels:
  I used two RBF kernels function according to the definition.

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

  where S(x) is the coordinate of the pixel of data x, and C(x) is the color information and the initial value of $\gamma_s = 0.001$, $\gamma_s = 0.01$ and X is the input image

```python
def RBF_kernel(X, gamma_s, gamma_c):  # img, spatial, color

    dist_c = cdist(X, X, 'sqeuclidean')  # (10000, 10000)
    X_spatial = np.array([[i, j] for i in range(100) for j in range(100)]) # (10000, 2)
    dist_s = cdist(X_spatial, X_spatial, 'sqeuclidean')
    RBF_s = np.exp(-gamma_s * dist_s)    # (10000,10000)
    RBF_c = np.exp(-gamma_c * dist_c)    # (10000,10000)
    kernel = np.multiply(RBF_s, RBF_c)   # (10000,10000)

    return kernel
```

Figure 2: two RBF kernels function

- Initial center:

  1. random:
     Randomly select k points from the data
  2. k-means++:
     The process is slightly different. In the k-means++ mode, you start by randomly selecting one data point. The objective is to find the data point that is farthest from the previous selection, as it will have the highest probability of being chosen next. This process is repeated until k initial data points have been found, each time selecting the farthest data point from the previous selections.

```python
def initial_center(k, kernel, mode="random"):

    if mode == "random":
        centers_idx = list(random.sample(range(0,10000), k))

    elif mode == "kmeans++":
        centers_idx = random.sample(range(kernel.shape[0]), 1)
        found = 1
        while found < k:

            dist = np.zeros(kernel.shape[0])

            for i in range(kernel.shape[0]):
                min_dist = np.Inf

                for f in range(found):
                    tmp = np.linalg.norm(kernel[i] - kernel[centers_idx[f]])

                    if tmp < min_dist:
                        min_dist = tmp

                dist[i] = min_dist

            dist = dist / np.sum(dist)
            idx = np.random.choice(np.arange(kernel.shape[0]), size=1, p=dist)
            centers_idx.append(idx[0])
            found += 1

    return centers_idx
```

Figure 3: initial center

- Kemans:
  Since the initial center function output provides the coordinates of the initial center, I begin by extracting the kernel of the initial center coordinate and placing it in the Mean array. The k-means algorithm consists of two key steps: the E-step and the M-step, which are repeated iteratively. In the E-step, all data points are assigned to the nearest data center, which is determined by calculating the distances to each mean. During the M-step, the new data centers are updated by calculating the means of the assigned data points, using the assignments made in the E-step. This iterative process of performing the E-step and M-step continues until the means converge. To determine the convergence, a threshold of 1e-12 is set, meaning that the algorithm stops when the change in means falls below this threshold.

```python
def kmeans(k, kernel, output_dir):

    iter = 0
    Mean = np.zeros((k, kernel.shape[1]))
    centers_idx = initial_center(k, kernel, mode="kmeans++")

    for i in range(len(centers_idx)):
        Mean[i] = kernel[centers_idx[i]].real

    cluster = np.zeros(len(kernel), dtype=np.uint8)
    diff = 1e9
    count = 1
    while diff > 1e-12 :
        # E-step
        for i in range(len(kernel)):
            dist = []
            for j in range(k):
                dist.append(np.sqrt(np.sum((kernel[i] - Mean[j])**2)))
            cluster[i] = np.argmin(dist)

        # M-step
        New_Mean = np.zeros(Mean.shape, dtype=np.float64)
        for i in range(k):
            belong = np.argwhere(cluster==i).reshape(-1)
            for j in belong:
                New_Mean[i] += kernel[j].real
            if len(belong) > 0:
                New_Mean[i] = New_Mean[i] / len(belong)

        diff = np.sum((New_Mean - Mean)**2)
        Mean = New_Mean
        save_png(k, cluster, iter, output_dir)
        iter += 1

    return cluster
```

Figure 4: kmeans

- Save png:
  The given code snippet defines a function named save png that is used to save a visualization of clustering results in the form of a PNG image file.

  The function save png takes four parameters as input:

  - k: The number of clusters.
  - cluster: An array containing the cluster assignments for each data point.
  - iter: The current iteration number.
  - output dir: The directory path where the PNG image will be saved.

  Map each point to its corresponding cluster and assign the corresponding index in the color array. For example, if the cluster is 0, then use the color [255, 0, 0]. Finally, reshape

the result to (100, 100, 3) and save the image to the specified output directory. The purpose of saving the image is to create a GIF file.

```python
def save_png(K, cluster, iter, output_dir):

    colors = np.array([[175,208,201], [145,161,186], [81,98,142], [24,32,68], [14,18,45]])
    result = np.zeros((100*100, 3))

    for k in range(K):
        mask = np.isin(cluster, [k])
        result[mask] = colors[cluster[mask]]

    img = result.reshape(100, 100, 3)
    img = Image.fromarray(np.uint8(img))
    img.save(os.path.join(output_dir, '%06d.png' % iter))
```

Figure 5: save

- Create gif:
  The create gif function generates a GIF animation by sequentially opening and appending image files from a file dir. The resulting animation is saved as a GIF file in the output dir.

```python
def create_gif(figure_dir, output_dir, filename):

    files = sorted(os.listdir(figure_dir))
    frames = []

    for file in files:
        file_path = os.path.join(figure_dir, file)

        image = Image.open(file_path)
        frames.append(image)

    output_file = os.path.join(output_dir, filename+'.gif')
    frames[0].save(output_file, format='GIF', append_images=frames[1:], save_all=True, duration=200, loop=0)
```

Figure 6: create gif

- Eigenspace visualize:
  The show eigen function is responsible for visualizing data in either a 2D or 3D plot, depending on the input data and cluster information. It employs a predefined list of colors to differentiate and represent distinct clusters. The function iterates through each K and assigns a color based on the corresponding cluster value (colors[k]).

```python
def show_eigen2D(K, data, cluster):
    colors = ['b', 'r']
    plt.clf()
    data = data.real
    for k in range(K):
        mask = np.isin(cluster, [k])
        plt.scatter(data[mask,0], data[mask,1], c=colors[k], alpha=0.1)
    plt.savefig("eigen_" + args.input_name + '_' + str(args.k)+ '_' + str(args.mode) + ".png")
```

(a) eigenspace visualize 2D

```python
def show_eigen3D(K, data, cluster, path):
    colors = ['b', 'r', 'g']
    plt.clf()
    data = data.real
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    for k in range(K):
        mask = np.isin(cluster, [k])
        ax.scatter(data[mask,0], data[mask,1], data[mask,2], c=colors[k], alpha=0.1)

    plt.savefig(path+"eigen_" + args.input_name + '_' + str(args.k)+ '_' + str(args.mode) + ".png")
```

(b) eigenspace visualize 3D

1. Kernel K-means
   In the main function, you can decide the size of K (number of clusters) and specify which image to use. You can use the aforementioned functions to divide the image into K clusters.

```python
import os
import numpy as np
import argparse
from util import *

parser = argparse.ArgumentParser()
parser.add_argument('--input', type=str, default='image1.png')
parser.add_argument('--input_name', type=str, default='image1')
parser.add_argument('--k', type=int, default=4)
parser.add_argument('--s', type=float, default=0.001)
parser.add_argument('--c', type=float, default=0.01)
args = parser.parse_args()

if __name__=='__main__':
    output_dir = os.path.join('Kmeans', args.input_name)
    if not os.path.exists(output_dir):
        os.mkdir(output_dir)

    X_color = load_png(args.input)
    kernel = RBF_kernel(X_color, args.s, args.c) #spatial, color
    cluster = kmeans(args.k, kernel, output_dir)

    # GIF
    GIF_dir = os.path.join('GIF')
    if not os.path.exists(GIF_dir):
        os.mkdir(GIF_dir)
    create_gif(output_dir, GIF_dir, args.input_name)
```

Figure 8: kmeans

2. Spectral clustering - ratio cut

   First, calculate W using the aforementioned function. Then, use W to obtain D and L. Next, use L to solve for the eigenvalues and eigenvectors. Save all the eigenvalues and eigenvectors as an npy file using np.save. This way, I only need to calculate the eigenvalues once per image, as solving for eigenvalues of a matrix of size 10000x10000 can be time-consuming. Then, create a "sort index" by sorting the eigenvalues in ascending order. Use a mask to obtain the indices of eigenvalues greater than 0 and select the top k eigenvalues and their corresponding eigenvectors. Save these eigenvectors as U, then apply the k-means algorithm to the rows of U (10000 × k) to obtain the final result.

```python
if args.mode == 1:  # unnormalized
    print("ratio spectral")
    output_dir = os.path.join('spectral_clustering_unnormalized', args.model, 'k_{}'.format(args.k), args.input_name)
    if not os.path.exists(output_dir):
        os.mkdir(output_dir)

    X_color = load_png(args.input)  # (10000, 3)

    ################ laplacian ################

    # W = RBF_kernel(X_color, args.s, args.c)
    # D = np.diag(np.sum(W, axis=1))
    # L = D - W

    # ***************** eigenvalue *****************

    # eigenvalue, eigenvector = np.linalg.eig(L) # (100,), (100, 100)
    # np.save(f"unnormalized_eigenvalue_{args.input_name}.npy", eigenvalue)
    # np.save(f"unnormalized_eigenvector.npy_{args.input_name}", eigenvector)

    eigenvalue = np.load(f"unnormalized_eigenvalue_{args.input_name}.npy")
    eigenvector = np.load(f"unnormalized_eigenvector_{args.input_name}.npy")

    # ***************** eigenvalue *****************

    sort_idx = np.argsort(eigenvalue)
    mask = eigenvalue[sort_idx] > 0
    sort_idx = sort_idx[mask]
    U = eigenvector[:, sort_idx[0:args.k]]
    cluster = kmeans(args.k, U, output_dir, args.model)

    # eigenspace
    eigen_path = os.path.join("spectral_clustering_normalized\\normalized_eigenspace\\")
    if args.model == 'kmeans++':
        if args.k == 2:
            show_eigen2D(args.k, U, cluster, eigen_path)
        elif args.k == 3:
            show_eigen3D(args.k, U, cluster, eigen_path)

    # GIF
    GIF_dir = os.path.join('spectral_clustering_unnormalized_GIF', args.model, 'k_{}'.format(args.k))
    if not os.path.exists(GIF_dir):
        os.mkdir(GIF_dir)
    create_gif(output_dir, GIF_dir, args.input_name)
```

(a) ratio cut

Unnormalized spectral clustering

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

- Construct a similarity graph by one of the ways described in Section 2. Let W be its weighted adjacency matrix.
- Compute the unnormalized Laplacian L.
- **Compute the first k eigenvectors $u_1, \ldots, u_k$ of L.**
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \ldots, u_k$ as columns.
- For $i = 1, \ldots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of U.
- Cluster the points $(y_i)_{i=1,\ldots,n}$ in $\mathbb{R}^k$ with the k-means algorithm into clusters $C_1, \ldots, C_k$.

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

(b) Alogrithm

3. Spectral clustering - normalized cut

   The main difference from the previous ratio cut approach lies in the construction of the L matrix. In this case, the L matrix needs to be normalized. Additionally, after obtaining the eigenvector matrix U, it is necessary to normalize the rows of U. Apart from these changes, all other steps remain the same as in the ratio cut approach.

```
elif args.mode == 2:
    print("Normalized Spectral Clustering")
    output_dir = os.path.join('spectral_clustering_normalized', args.model, 'k_{}'.format(args.k), args.input_name)
    if not os.path.exists(output_dir):
        os.mkdir(output_dir)
    X_color = load_png(args.input)  # (10000, 3)

    ############### laplacian #################

    # W = RBF_kernel(X_color, args.s, args.c)
    # D = np.diag(np.sum(W, axis=1))
    # L = D - W
    # Dsym = D_minus_half_square_root(D)
    # Lsym = Dsym.dot(L).dot(Dsym)

    # ***************** eigenvalue *****************

    # eigenvalue, eigenvector = np.linalg.eig(Lsym)
    # np.save(f"normalized_eigenvalue_{args.input_name}.npy", eigenvalue)
    # np.save(f"normalized_eigenvector_{args.input_name}.npy", eigenvector)
    eigenvalue = np.load(f"normalized_eigenvalue_{args.input_name}.npy")
    eigenvector = np.load(f"normalized_eigenvector_{args.input_name}.npy")

    # ***************** eigenvalue *****************

    sort_idx = np.argsort(eigenvalue)
    mask = eigenvalue[sort_idx] > 0
    sort_idx = sort_idx[mask]
    U = eigenvector[:, sort_idx[0:args.k]]
    T = normalize_rows(U)
    cluster = kmeans(args.k, T, output_dir, args.model)

    # eigenspace
    eigen_path = os.path.join("spectral_clustering_normalized\\normalized_eigenspace\\")
    if args.model == 'kmeans++':
        if args.k == 2:
            show_eigen2D(args.k, U, cluster, eigen_path)

        elif args.k == 3:
            show_eigen3D(args.k, U, cluster, eigen_path)

    # GIF
    GIF_dir = os.path.join('spectral_clustering_normalized_GIF', args.model, 'k_{}'.format(args.k))
    if not os.path.exists(GIF_dir):
        os.mkdir(GIF_dir)
    create_gif(output_dir, GIF_dir, args.input_name)
```

(a) normalized cut

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

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

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

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

(b) Alogrithm

2 Experiments settings and results

I utilized two different methods to initialize the center, as shown below. Additionally, I stored both the figure and the GIF file within a zip archive.

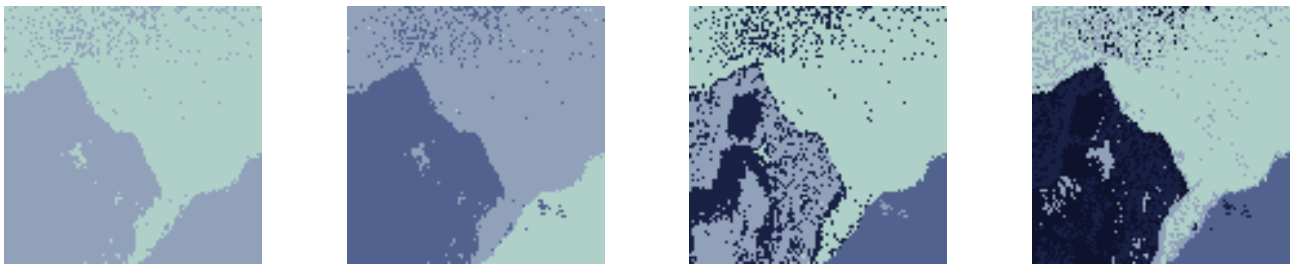
- $\gamma_s = 0.00001$
- $\gamma_c = 0.0001$
- Original figure:

(a) image1 (b) image2

1. Kernel K-means:

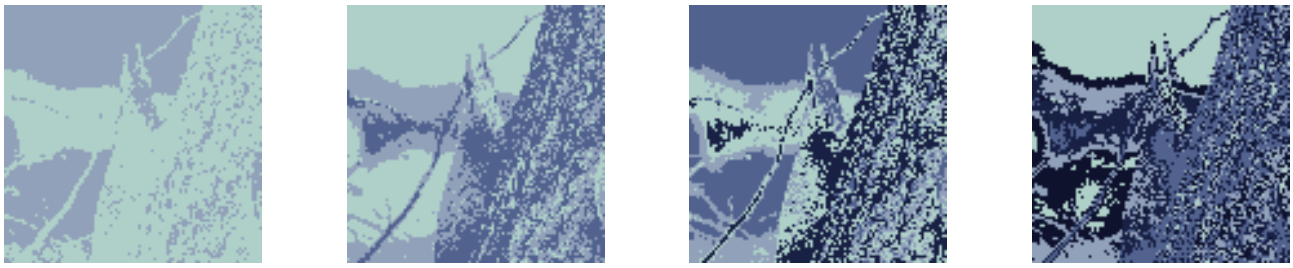
   In Figures 12 13 and 14 15, we employed two different methods, namely random initialization and k-means++. While there are cases where the random initialization produces the same result as k-means++, overall, k-means++ demonstrated superior performance. This can be attributed to the fact that k-means++ takes into account the distances between the initial centroid points, resulting in more effective clustering.
   In Figures 12 (d) and 14 (d), we can observe that these two methods can sometimes exhibit significant differences. However, it is important to note that both methods yield satisfactory results overall, the boundaries are well-defined in both cases.

   - random:



(a) K=2 (b) K=3 (c) K=4 (d) K=5

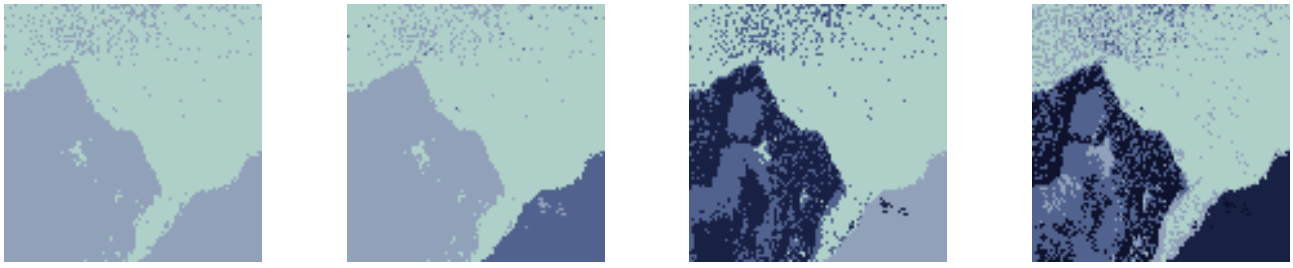
Figure 12: Kernel K-means of Image1



(a) K=2 (b) K=3 (c) K=4 (d) K=5

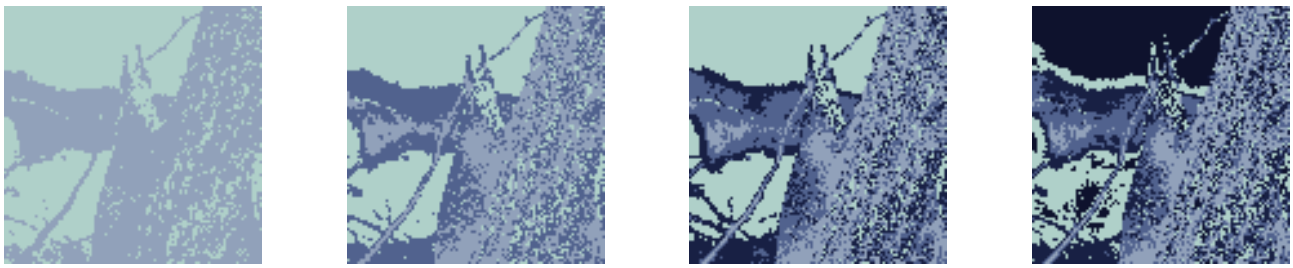
Figure 13: Kernel K-means of Image2

- Kmeans++:



(a) K=2 (b) K=3 (c) K=4 (d) K=5

Figure 14: Kernel K-means of Image1



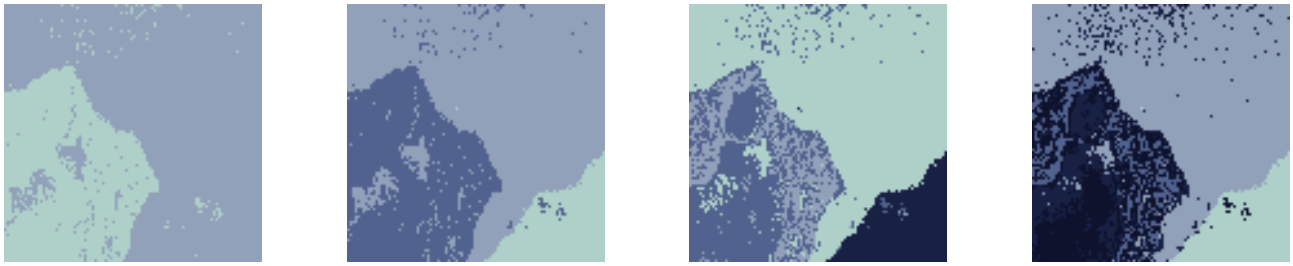
(a) K=2 (b) K=3 (c) K=4 (d) K=5

Figure 15: Kernel K-means of Image2

2. Ratio cut:

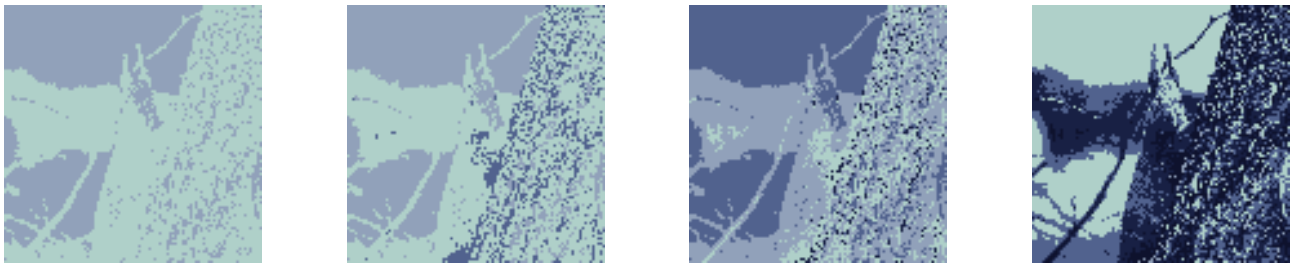
In the case of Ratio Cut and Normalized Cut, I noticed occasional noticeable differences in the results depending on the selection of initial conditions. This disparity is primarily attributed to the choice of initial centroids. Additionally, if the image has a simpler structure with fewer boundaries, the distinctions between Ratio Cut and Normalized Cut are less pronounced. However, it is worth noting that as the number of clusters increases, the disparity between Ratio Cut and Normalized Cut becomes more apparent. This observation holds true for both Image 1 and Image 2.

- random



(a) K=2 (b) K=3 (c) K=4 (d) K=5

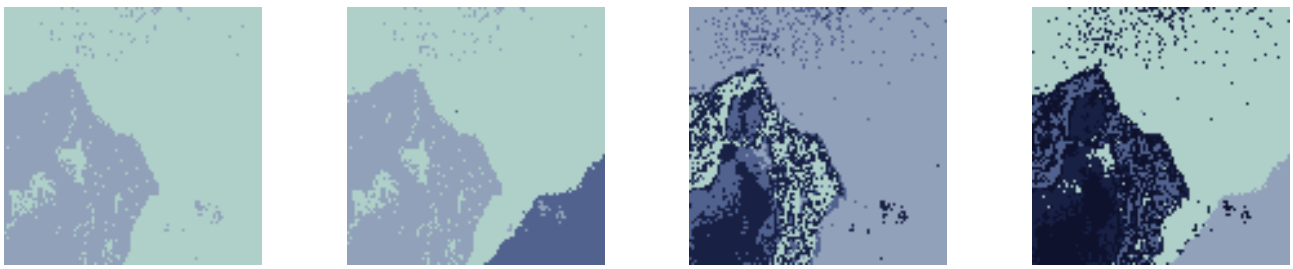
Figure 16: Ratio cut of Image1



(a) K=2 (b) K=3 (c) K=4 (d) K=5

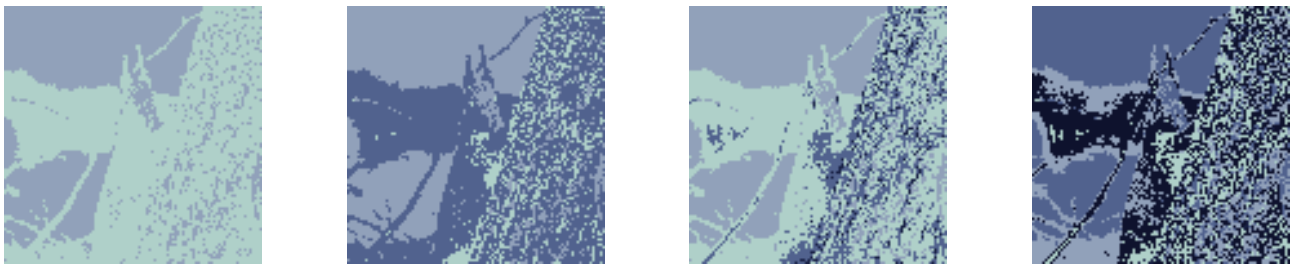
Figure 17: Ratio cut of Image2

- kmeans++



(a) K=2 (b) K=3 (c) K=4 (d) K=5

Figure 18: Ratio cut of Image1

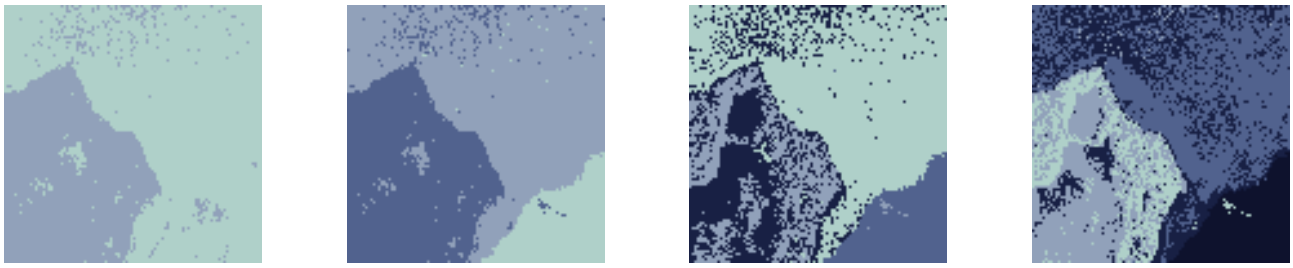


(a) K=2 (b) K=3 (c) K=4 (d) K=5

Figure 19: Ratio cut of Image2

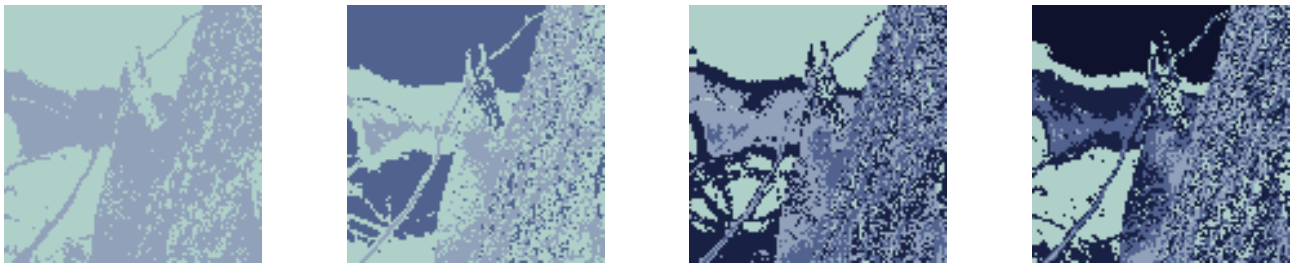
3. Normalize cut:

   - random



(a) K=2 (b) K=3 (c) K=4 (d) K=5

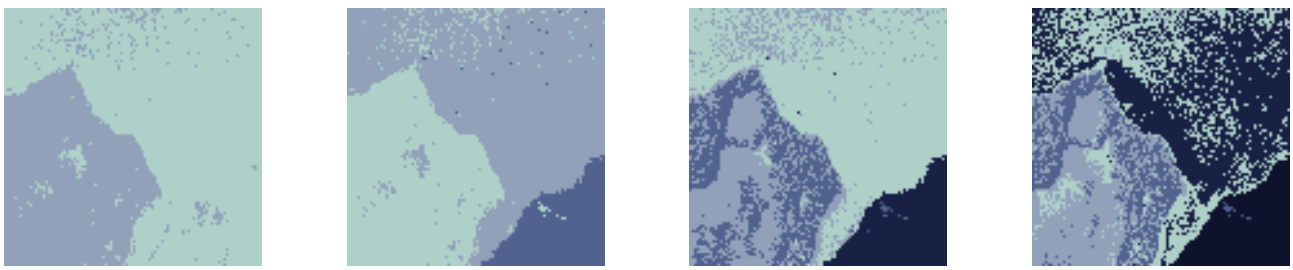
Figure 20: Normalize cut of Image1



(a) K=2 (b) K=3 (c) K=4 (d) K=5

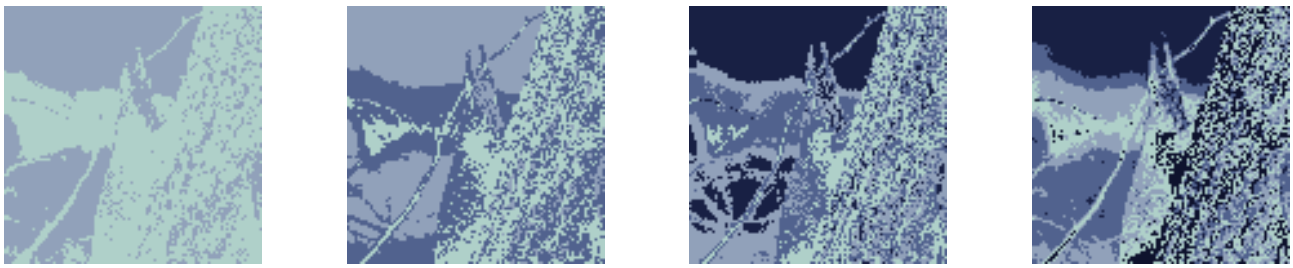
Figure 21: Normalize cut of Image2

   - kmeans++



(a) K=2 (b) K=3 (c) K=4 (d) K=5

Figure 22: Normalize cut of Image1



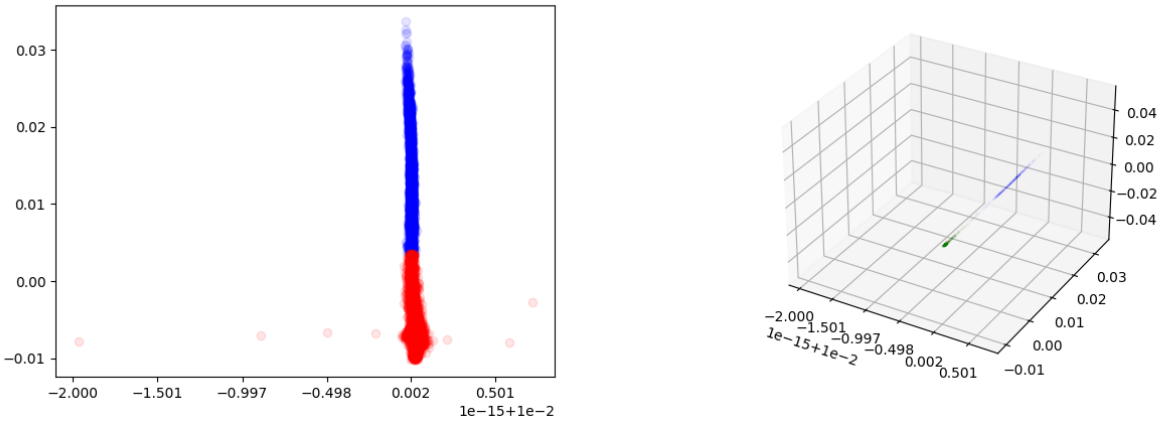
(a) K=2 (b) K=3 (c) K=4 (d) K=5

Figure 23: Normalize cut of Image2

The following plots illustrate the coordinates in eigenspace for different cuts, specifically for k=2 or k=3 using the k-means++ algorithm.

1. Ratio cut

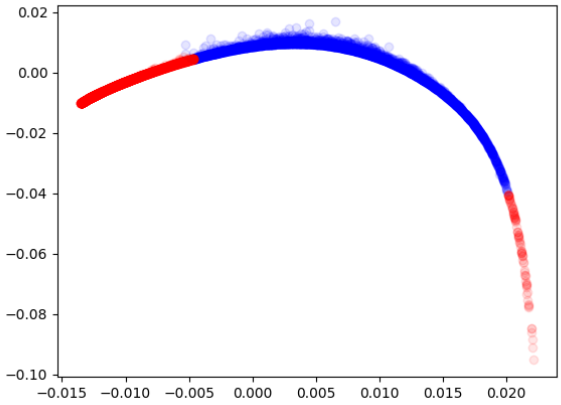
   - Image 1



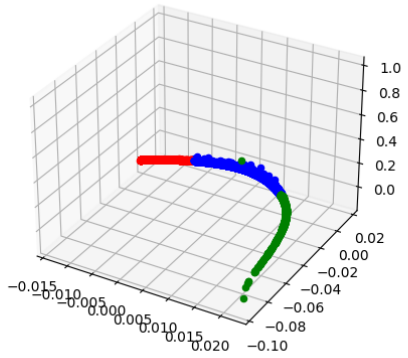
(a) K=2 (b) K=3

Figure 24: coordinates in the eigenspace of graph Laplacian

   - Image 2



(a) K=2

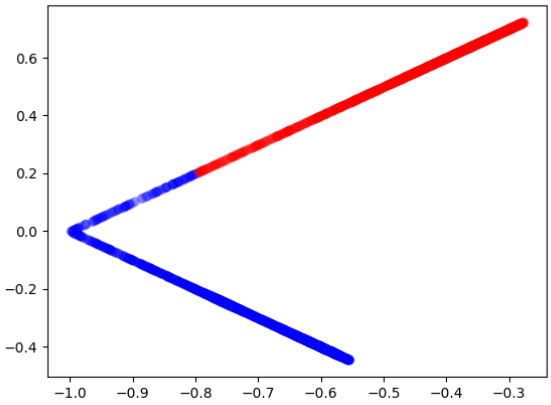


(b) K=3

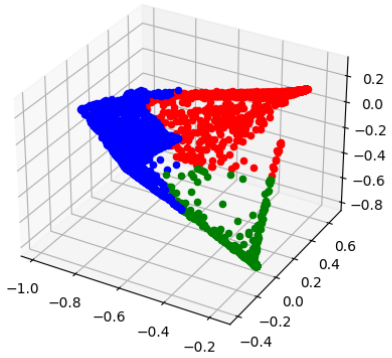
Figure 25: coordinates in the eigenspace of graph Laplacian

2. Normalize cut

- Image 1



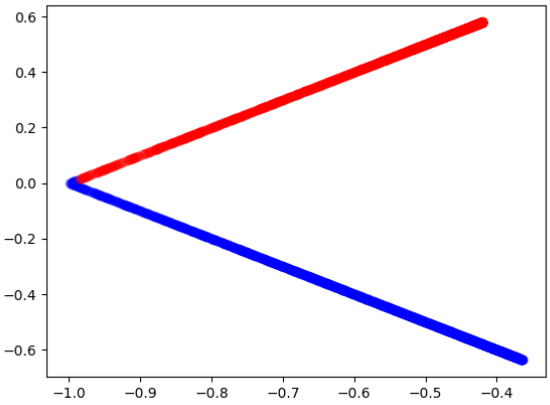
(a) K=2



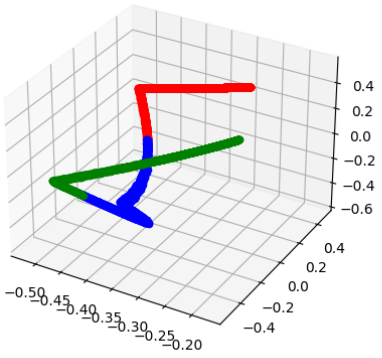
(b) K=3

Figure 26: coordinates in the eigenspace of graph Laplacian

- Image 2



(a) K=2



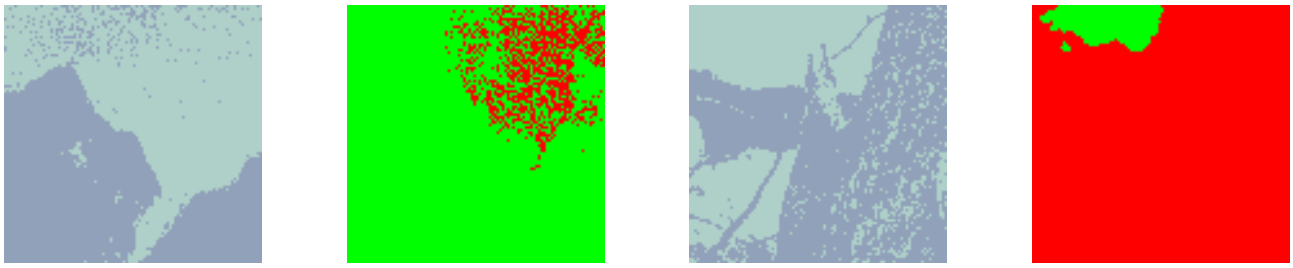
(b) K=3

Figure 27: coordinates in the eigenspace of graph Laplacian

- Not all dimensions are so useful for classifying the data see Figure 24 (a). The 1^{rd} dimension is almost useless .
- The resulting plots for ratio cut and normalized cut exhibit significant differences.
- In the eigenspace, data points belonging to the same cluster are in close proximity to each other. However, their coordinates in the original space may not be the same.

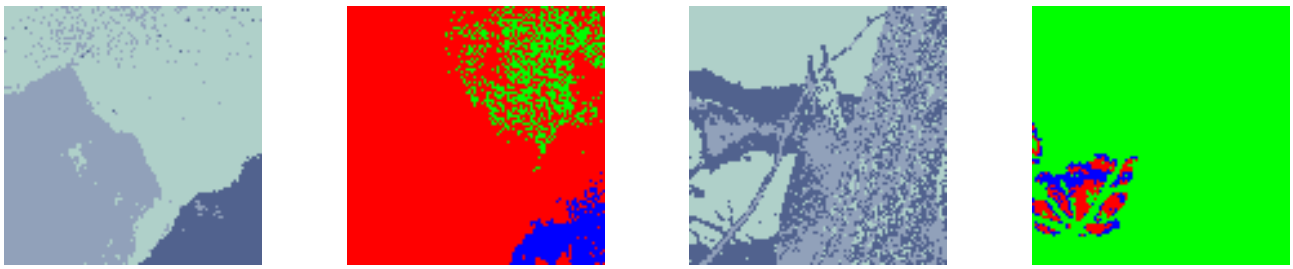
3 Observations and discussion

- In terms of execution time, K-means is significantly faster than spectral clustering. The main difference lies in the fact that spectral clustering requires the computation of eigenvalues and eigenvectors, and solving for eigenvalues of a 10000x10000 matrix can be extremely time-consuming.
- In normalized cut, there can be complex eigenvalues and eigenvectors. When comparing them, I choose to ignore the complex part since complex numbers cannot be directly compared in terms of magnitude.
- The values of γ_s and γ_c are very important. When I set γ_s to 0.001 and γ_c to 0.01, the performance is very poor. However, when I set γ_s to 0.00001 and γ_c to 0.0001, the performance improves significantly see figure 28 29, on the left is $\gamma_s = 0.00001$ and $\gamma_c = 0.0001$. On the right is $\gamma_s = 0.001$ and $\gamma_c = 0.01$.
- I'm thinking that different values of K may require different parameters. Looking at the graphs I mentioned, I feel that K=2 and K=3 classify well, but K=4 and K=5 seem just okay.



(a) image 1 (b) image 1 (c) image 2 (d) image 2

Figure 28: K=2



(a) image 1 (b) image 1 (c) image 2 (d) image 2

Figure 29: K=3