Machine Learning Homework 6

Kernel K-means & Spectral clustering

ChiaLiang Kuo 310552027

- HackMD link : https://hackmd.io/n5nRwNnkS82BQj1Kwa7rCQ
- Github link :

Code

main function

Before elucidating the main function, I would first introduce the input arguments first.

- `ctype` : The clustering type, 1 for kernel kmeans and 2 for spectral clustering
- `stype` : The type of spectral clustering, 1 for ratio-cut and 2 for normalized cut
- `itype` : The initialize method of the clustering, 1 for randomly initializing and 2 for kmeans++
- `gamma_c` : The parameter for the color similarity of the kernel function
- `gamma_s` : The parameter for the spacial similarity of the kernel function
- `k` : The number of clusters (min K = 2, max K = 10). This maximum limitation was due to the limited memory space of the eigen vectors used in spectral clustering. Because the graph Laplacian matrix was a 10000x10000 2D matrix, the precomputed eigen vectors would be about 1.6 GB in disk.
- `save_eigenvec` : A flag that if it was choosen, then the main function would save eigen vectors for computing the spectral clustering. This flag was needed because it would took about 15 minutes to compute the eigen vectors from scratch. As long as the eigen vectors were saved previously, the spectral clustering process would be fast by loading the precomputed eigen vectors in runtime.

```
if __name__=="__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--ctype', '-c', type=int, default=1, help='the
clustering type [1:kkmeans, 2:spectral]')
    parser.add_argument('--stype', '-s', type=int, default=1, help='type of
spectral clustering [1:ratio_cut, 2:normalized_cut]')
    parser.add_argument('--itype', '-i', type=int, default=1,
help='initialization methods [1:random, 2:kmeans++]')
    parser.add_argument('--gamma_c', '-gc', type=float, default=0.0001,
help='gamma c for kernel')
    parser.add_argument('--gamma_s', '-gs', type=float, default=0.0001,
help='gamma s for kernel')
    parser.add_argument('--k', '-k', type=int, default=3, help='k for
clustering')
    parser.add_argument('--save_eigenvec', '-seg', action='store_true',
default=False, help='save eigen vector only')
    args = parser.parse_args()
    main(args)
```

The procedure of the main function was :

- parse the input arguments mentioned previously
- if `save_eigenvec` was true, then it would save precomputed eigen vectors only, otherwise it would run the clustering process.
    - kernel kmeans would take the gram matrix of the input data and then ran the kmenas method.
    - spectral clustering would take the gram matrix as input and then computed the (normalized) graph Laplacian matrix for computing the first k eigen vectors of it, and finally ran the kmeans algorithm by passing the first k eigen vectors as input data.
- In order to plot the gif animations of the clustering results, `visualize_clusters` function would be run and then save them to files.

```
def main(args):
    paths = glob.glob(f'*.png')

    k = args.k
    assert k <= MAX_K and k >= 2, f'k should be in [0, {MAX_K}]'

    gamma_c = args.gamma_c
    gamma_s = args.gamma_s
    clustering_type = {1:'kkmeans', 2:'spectral'}[args.ctype]
    spectral_type = {1:'ratio-cut', 2:'normalized-cut'}[args.stype]
    init_type = {1:'random', 2:'kmeans++'}[args.itype]

    for path in paths:
        print(f'[processing kernel K-mean of image {path}]')

        # load image1 and image2
        img = cv2.imread(path)

        # get_gram_matrix img data with respect to color and space
        rgb_flatten = get_flattened_imrbg(img_resize)
        loc_flatten = get_flattened_imloc(img_resize.shape)
        gram_mat = get_gram_matrix(rgb_flatten, loc_flatten,
gamma_s=gamma_s, gamma_c=gamma_c)

        # for the saving directory name
        prefix = os.path.splitext(path)[0]

        # save eigen vectors for spectral clustering only
        if args.save_eigenvec :
            save_eigenvec(gram_mat, prefix, gamma_s=gamma_s,
gamma_c=gamma_c)
        # run the clustering process
        else :
            if clustering_type == 'kkmeans':
                # run kernel kmeans
                cluster_frames = kmeans(gram_mat, k, init_type)
                save_path =
f'output/{prefix}/gif/{clustering_type}_{init_type}_{k}_{gamma_s}_{gamma_c}
.gif'
```

```
            elif clustering_type == 'spectral':
                # run spectral clustering
                cluster_frames = spectral_clustering(gram_mat, k,
init_type, prefix, spectral_type, gamma_s, gamma_c)
                save_path =
f'output/{prefix}/gif/{spectral_type}_{init_type}_{k}_{gamma_s}_{gamma_c}.g
if'
            else :
                raise Exception(f'clustering type error :
{clustering_type}')

            visualize_clusters(save_path, cluster_frames, img_resize.shape,
palette)
```

init_clusters function

This function would run the initialize method for computing kmeans algorithm with the given `init_type` argument.

- `init_type = 'random'` : randomly choosed the k centers of the given data
- `init_type = 'kmeans++'` : assign the first centroid to the location of a randomly selected data point, and then choosing the subsequent centroids from the remaining data points based on a probability proportional to the squared distance away from a given point's nearest existing centroid. The effect is an attempt to push the centroids as far from one another as possible, covering as much of the occupied data space as they can from initialization

```
def init_clusters(data, k, init_type):
    '''
    @param data: ndarray with shape (n, dim), input data or input feature
    @param k: int, number of clusters for kmeans initialization
    @param initType: string, ['random','kmeans++']
    @return: ndarray with shape (k, dim),
    '''
    data_size = data.shape[0]
    data_dim = data.shape[1]

    centers = np.zeros((k, data_dim))
    if init_type == 'random':
        clusters = np.array([np.random.randint(0, k) for i in
range(data_size)])
        for j in range(k):
            cond = np.where(clusters == j)
            centers[j] = np.mean(data[cond], axis=0)

    elif init_type == 'kmeans++':
        # randomly choose one as first center
        first_center = data[np.random.randint(data_size)]
        centers[0] = first_center
        for j in range(1, k):
            dist = np.array([min([np.sum((x - c)**2) for c in centers]) for
```

```
x in data])
            dist /= np.sum(dist)
            cumulative_dist = np.cumsum(dist)
            r = np.random.rand()
            for jj, p in enumerate(cumulative_dist):
                if r < p:
                    i = jj
                    break

            centers[j] = data[i]

    return centers
```

get_gram_matrix function

In this assignment, we were asked to implement the kernel matrix by this function:

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

which is basically multiplying two RBF kernels in oreder to consider spatial similarity and color similarity at the same time. `S(x)` is the spatial information (i.e. the coordinate of the pixel) of data `x`, and `C(X)` is the color information (i.e. the RGB values) of data `x`, Both `gamma_s` amd `gamma_c` are hyper-parameters.

```
def get_gram_matrix(img_c, img_s, gamma_s, gamma_c):
    '''
    @param img_c: ndarray, the flattened RGB image
    @param img_s: ndarray, the flattened coordinate matrix of the RGB image
    @param gamma_s: float, for computing the spacial similarity kernel of
image data
    @param gamma_c: float, for computing the color similarity kernel of
image data
    @return: ndarray, the gram matrix of the image
    '''
    c_x = cdist(img_c, img_c) ** 2
    s_x = cdist(img_s, img_s) ** 2

    # gram matrix
    k = np.exp(-gamma_s * s_x) * np.exp(-gamma_c * c_x)

    return k
```

kmeans function

First, the initialized clusters would be computed by calling `init_clusters` function, then the while loop would run the E step and M step of kmeans method iteratively until the cluster result converged, and the

clustering results in each step would be returned.

- **E step** : kept k centers unchanged and updated the indicator matrix for k categories by assigning the data to j'th cluster which was the nearest center from the data to it.
- **M step** : kept the indicator matrix unchanged and updated k centers by averaging the data in j'th column of the indicator matrix.
- When the difference between the old clusters and new clusters was less than 10, the procedure would break the while loop.

```
def kmeans(data, k, init_type):
    '''
    @param data: ndarray with shape (n, dim), the input data or features
    @param k: int, number of clusters for kmeans initialization
    @param initType: string, ['random','kmeans++']
    @return: list of ndarray with shape (n, 1), a list of the clustering
result for each iteration
    '''
    data_size = data.shape[0]

    # for computing the distance to each center
    dist_map = np.zeros((data_size, k))
    # initial centers
    means = init_clusters(data, k, init_type)
    # for comparing with new clustering
    cluster_old = -np.ones(data_size) # initialize as -1
    # for recording cluster result for each step
    cluster_frames = []

    diff = np.inf
    iteration = 0
    max_iteration = 1000
    thresh = 10
    while diff > thresh and iteration < max_iteration:
        iteration += 1

        # E step -> update clusters with keeping centers unchanged
        for j in range(k):
            dist_map[:, j] = np.sum((data - means[j]) ** 2, axis=1)
        cluster = np.argmin(dist_map, axis=1)

        # M step -> update centers with keeping clusters unchanged
        for j in range(k):
            cond = np.where(cluster == j)
            means[j] = np.mean(data[cond], axis=0)

        # check if converge
        diff = np.sum(np.abs(cluster - cluster_old))
        cluster_old = cluster
        print(f'[kmeans iteration : {iteration}, diff : {diff}]')

        cluster_frames.append(cluster)
```

```
    return cluster_frames
```

spectral clustering function

This function would first compute the graph Laplacian matrix or the normalized graph Laplacian matrix based on the `spectral type` argument, and then computed the eigen vectors of the matrix and then sorted them by the ascending order of the eigen values. Finally, put first k eigen vectors as nx1 vectors to form a matrix U (size = nxk) and then normalized each row of U to form matrix T and then run the kmeans algorithm by passing T (size = nxk) as input data. If `k=3`, the eigen space would be visualized as figure. This function would return the clustering results for each step of the kmeans.

Actually, this function would load the precomputed eigen vectors and run the kmeans algorithm directory by input the T matrix mentioned above. It was because computing the eigen vectors from scratch would spend lots of time (about 15 minutes). The code below was just an example that running the spectral clustering process from scratch.

The graph Laplacian matrix was denoted as `L = D - W` and the normalized graph Laplacian matrix was denoted as `L_sym = D^-0.5 x L x D^-0.5 = I - D^-0.5 x W x D^-0.5`, where `D` is the degree of each data, or the sum for each row(column) in similarity matrix `W`, and `W` was actually the gram matrix by computing the kernel function in this assignmet. Beacuse each item instead of the diagonal term in `W` was promised to be greater than zero, the `W` could be seen as an adjacency matrix from a connected graph, so there exists only one eigen value of `L` to be 0.

```
def spectral_clustering(W, k, init_type, prefix, spectral_type, gamma_s,
gamma_c):
    '''
    @param W: ndarray, gram matrix of the image data
    @param k: int, number of clusters
    @param initType: string, ['random','kmeans++'] for kmeans
initialization
    @param prefix: string, for output file's directory
    @param spectral_type: string, ['ratio-cut','normalized-cut']
    @param gamma_s: float, for computing the spacial similarity kernel of
image data
    @param gamma_c: float, for computing the color similarity kernel of
image data
    @return: list of ndarray with shape (n, 1), a list of the clustering
result for each iteration
    '''

    data_size = W.shape[0]

    # setting Diagonal matrix
    D = np.zeros((data_size, data_size))
    for i in range(data_size):
        D[i, i] = np.sum(W[i])

    # graph laplacian matrix, where W is the kernel similarity matrix
```

```
    L = D - W

    if spectral_type == 'ratio-cut':
        # just need the eigen vectors of the original graph laplacian
matrix
        eigen_vals, eigen_vecs = np.linalg.eig(L)
    elif spectral_type == 'normalized-cut':
        # eigen vectors of the normalized graph laplacian matrix
        # L_sym = D^-0.5 x L x D^-0.5 = I - D^-0.5 x W x D^-0.5
        L_sym = np.linalg.inv(D ** 0.5) @ L @ np.linalg.inv(D ** 0.5)
        eigen_vals, eigen_vecs = np.linalg.eig(L_sym)

    # sort the eigen values as ascending order
    sorted_ind = np.argsort(eigen_vals)
    eigen_vecs = eigen_vecs[:, sorted_ind].real # just need the real part

    # load first k eigen vectors
    U = eigen_vecs[:, :k]
    # normalize each coordinate within 0->1
    sums=np.sqrt(np.sum(np.square(U),axis=1)).reshape(-1,1)
    T = U/sums

    # pass first k eigenspace of Laplacian matrix or normalized Laplacian
matrix
    # as input data of kmeans
    cluster_frames = kmeans(T, k)

    # for visualizing the eigenspace
    if k == 3:
        final_cluster = cluster_frames[-1]
        title = f'Eigenspace of {prefix} Using {spectral_type} (k=3,
gamma_s={gamma_s}, gamma_c={gamma_c})'
        save_path =
f'output/{prefix}/eigen_space/{spectral_type}_{init_type}_{gamma_s}_{gamma_
c}.png'
        visualize_eigenspace(final_cluster, T, title, save_path)

    return cluster_frames
```

save_eigenvec function

This function would save the eigen vectors of the (normalized) graph Laplacian matrix by the gram matrix `W` computed by `gamma_s`, `gamma_c` given and the output files will be in `.npz` format. Becuase of the limited storage space, only a subset of the eigen vectors (first `MAX_K` eigen vectors) would be saved into disk.

```
def save_eigenvec(W, prefix, gamma_s, gamma_c):
    '''
    @param W: ndarray, gram matrix of the image data
    @param prefix: string, for output file's directory
    @param gamma_s: float, for computing the spacial similarity kernel of
image data
```

```
    @param gamma_c: float, for computing the color similarity kernel of
image data
    @return: list of ndarray, a list of the clustering result for each
iteration
    '''
    data_size = W.shape[0]

    # setting Diagonal matrix
    D = np.zeros((data_size, data_size))
    for i in range(data_size):
        D[i, i] = np.sum(W[i])

    # graph laplacian matrix, where W is the kernel similarity matrix
    L = D - W

    # for ratio cut
    eigen_vals, eigen_vecs = np.linalg.eig(L)
    sorted_ind = np.argsort(eigen_vals)
    eigen_vals = eigen_vals[sorted_ind].real
    eigen_vecs = eigen_vecs[:, sorted_ind].real

    # save first (MAX_K=10) eigen vectors
    eigen_vals_k = eigen_vals[:MAX_K]
    eigen_vecs_k = eigen_vecs[:, :MAX_K]
    save_path = 'eigen_vec/{}_ratio-
cut_sorted_eg_{:.5f}_{:.5f}.npz'.format(prefix, gamma_s, gamma_c)
    with open(save_path, 'wb') as f:
        np.savez(f, eg_val=eigen_vals_k, eg_vec=eigen_vecs_k)
        print(f'write {save_path} done.')

    # for normalized cut
    # D^-0.5 x L x D^-0.5 = I - D^-0.5 x W x D^-0.5
    L_sym = np.linalg.inv(D ** 0.5) @ L @ np.linalg.inv(D ** 0.5)
    eigen_vals, eigen_vecs = np.linalg.eig(L_sym)
    sorted_ind = np.argsort(eigen_vals)
    eigen_vals = eigen_vals[sorted_ind].real
    eigen_vecs = eigen_vecs[:, sorted_ind].real

    # save first (MAX_K=10) eigen vectors
    eigen_vals_k = eigen_vals[:MAX_K]
    eigen_vecs_k = eigen_vecs[:, :MAX_K]
    save_path = 'eigen_vec/{}_normalized-
cut_sorted_eg_{:.5f}_{:.5f}.npz'.format(prefix, gamma_s, gamma_c)
    with open(save_path, 'wb') as f:
        np.savez(f, eg_val=eigen_vals_k, eg_vec=eigen_vecs_k)
        print(f'write {save_path} done.')
```

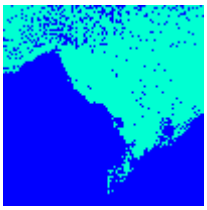
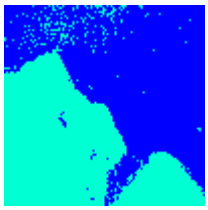
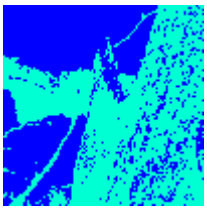
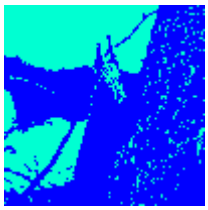
Experiments & Discussion

In the experiment section, I would visualize all the clustering results step by step into gif files.

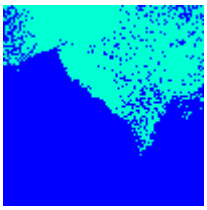
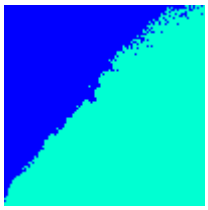
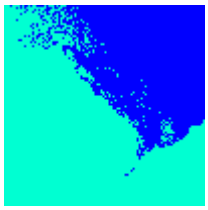
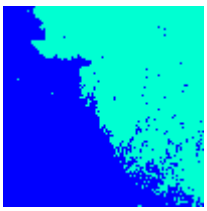
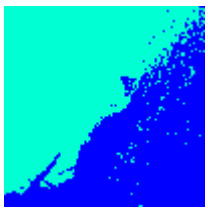
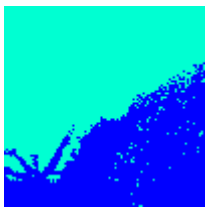
Part 1

This part I tried 4 different (gamma_s, gamma_c) pairs for the kernel function and ran 3 different clustering algorithms (kernel kmeans, spectral clustering using ratio cut and spectral clustering using normalized cut) to cluter 2 images respectively with k set to 2 and applying kmeans++ initialization method. The stop criteria in my implementation was the L1 distance between the current clustering result and the previous one smaller or equal than 10.

- k = 2
- gamma_s = 0.0001
- gamma_c = 0.0001
- initialization : kmeans++

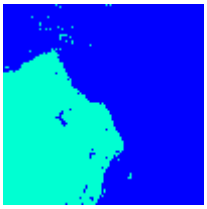
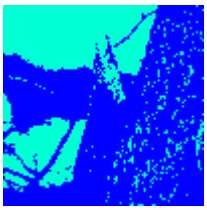
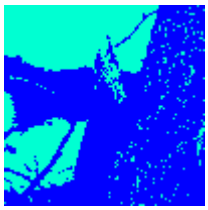
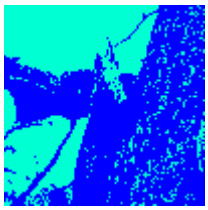
original image	kernel kmeans	ratio-cut	normalized-cut
			
			

- k = 2
- gamma_s = 0.0001
- gamma_c = 0.001
- initialization : kmeans++

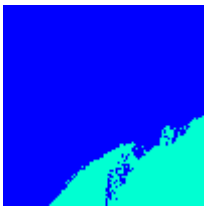
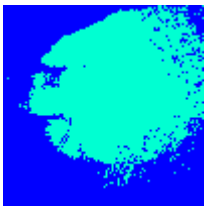
original image	kernel kmeans	ratio-cut	normalized-cut
			
			

- k = 2
- gamma_s = 0.001
- gamma_c = 0.0001
- initialization : kmeans++

original image	kernel kmeans	ratio-cut	normalized-cut

original image	kernel kmeans	ratio-cut	normalized-cut
			
			

- `k` = 2
- `gamma_s` = 0.001
- `gamma_c` = 0.001
- initialization : `kmeans++`

original image	kernel kmeans	ratio-cut	normalized-cut
			
			

To my observation, most of the iterations in each algorithm were less than 10, which means these method would lead to the clustring result converge in a quite short time. Besides, I found that kernel kmeans would averagely take a little more iterations to converge than spectral clustering algorithm.

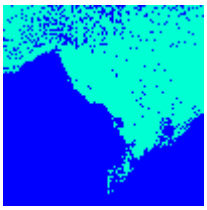
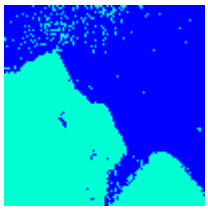
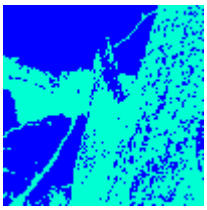
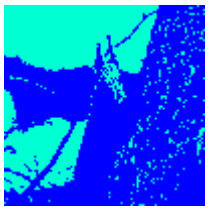
Further more, we could discover that smaller `gamma_c`, `gamma_s` tended to make the clustering results more exquisitely, especially `(gamma_s, gamma_c) = (0.0001, 0.0001)`. Moreover, kernel parameter would make the value of the kernel function larger, and thus we could found that color parameter (`gamma_c`) may affect the result more because the one using `(gamma_s, gamma_c) = (0.001, 0.0001)` would converge faster than the one using `(gamma_s, gamma_c) = (0.0001, 0.001)`.

Part 2

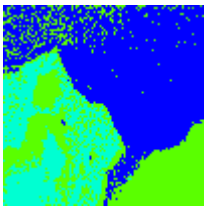
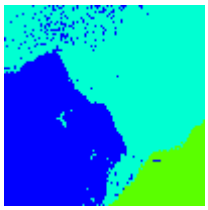
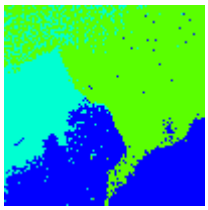
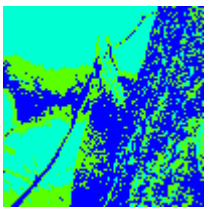
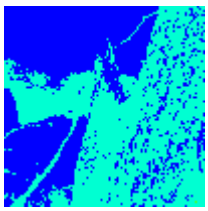
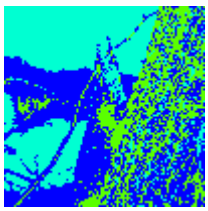
This part I would show the result that applied different `k` with the same kernel parameters pair `(gamma_s, gamma_c) = (0.0001, 0.0001)` and compare the three algorithms with kmeans++ initialization method. Why I chose this pair of kernel parameters was based on the discussion in Part1.

- `k` = 2

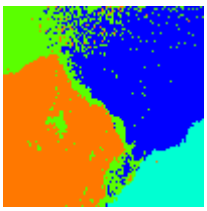
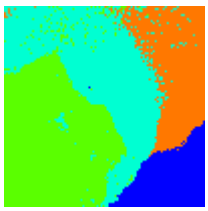
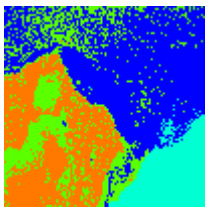
- gamma_s = 0.0001
- gamma_c = 0.0001
- initialization : kmeans++

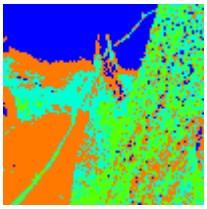
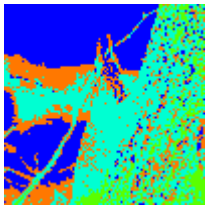
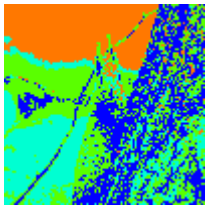
original image	kernel kmeans	ratio-cut	normalized-cut
			
			

- k = 3
- gamma_s = 0.0001
- gamma_c = 0.0001
- initialization : kmeans++

original image	kernel kmeans	ratio-cut	normalized-cut
			
			

- k = 4
- gamma_s = 0.0001
- gamma_c = 0.0001
- initialization : kmeans++

original image	kernel kmeans	ratio-cut	normalized-cut
			

original image	kernel kmeans	ratio-cut	normalized-cut
			

According to the results shown above, I could not assert that which `k` was the best. Instead, I would say it depends.

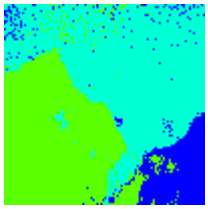
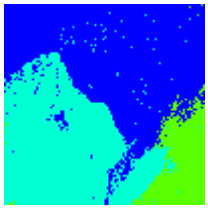
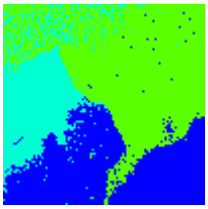
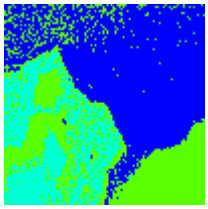
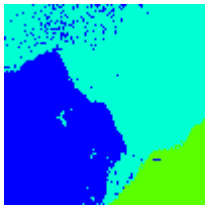
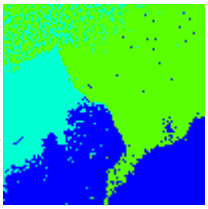
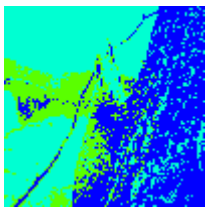
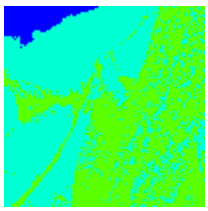
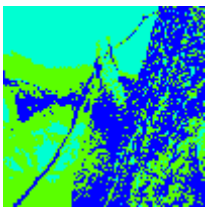
In image1, `k=2, 3` seemed to segment the image more accurate than `k=4`, and `k=2` would be a more suitable choise in image2 becuase the results of the three algorithms were similar.

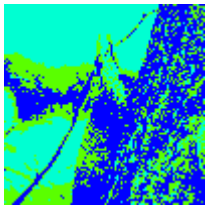
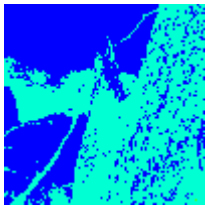
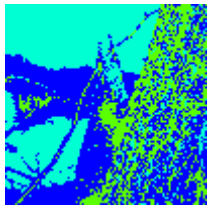
In summary, the two images may be more suitable for smaller `k` (less than or equal to 3). According to the experiments, when larger `k` applied, the result would become noisy.

Part 3

This section we would compare the results that applied different initialization methods. I tried two method, which was random assigned and kmeans++ method. I would set the cluster number `k` to be 3 and apply `(gamma_s, gamma_c) = (0.0001, 0.0001)` and compare three different algorithms.

- `k` = 3
- `gamma_s` = 0.0001
- `gamma_c` = 0.0001

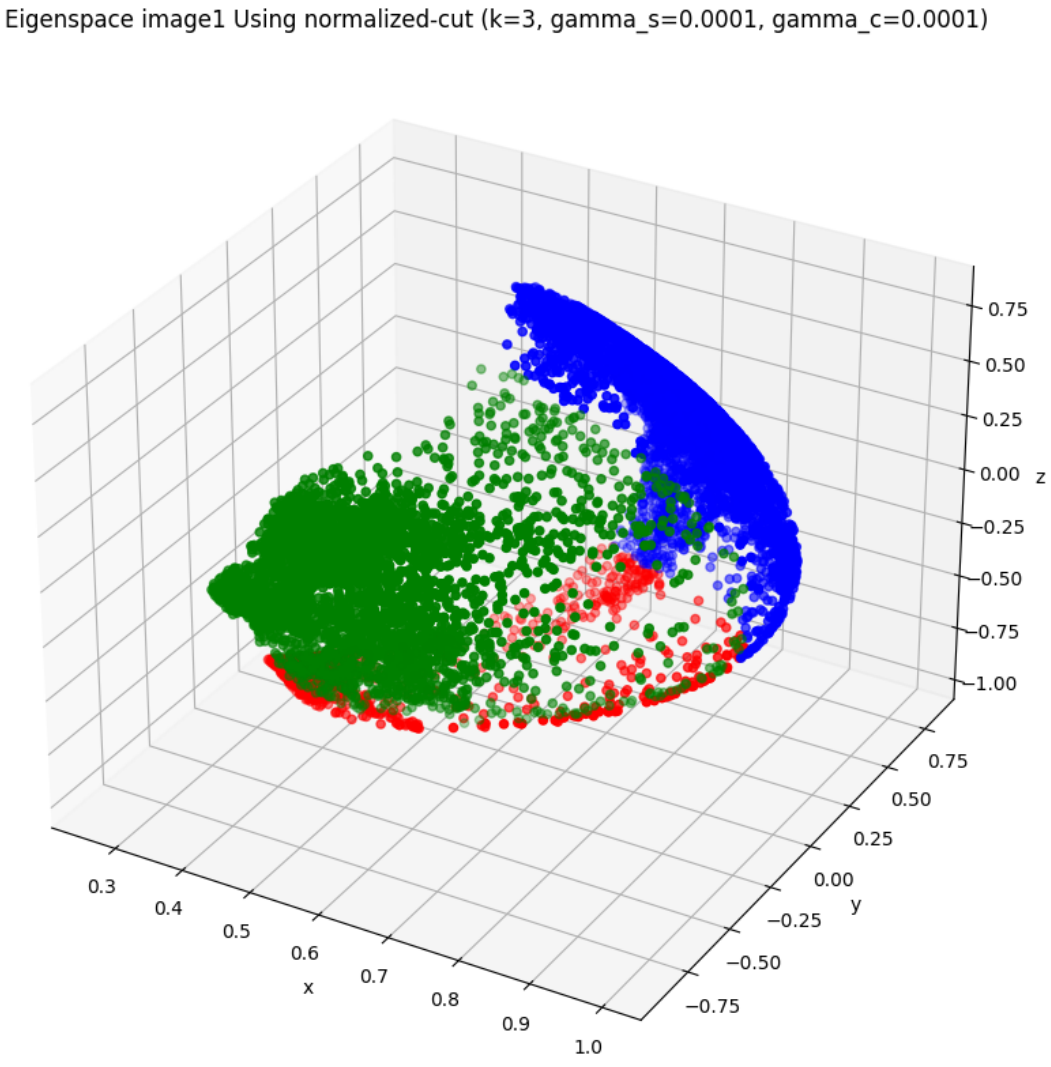
original image	kernel kmeans	ratio-cut	normalized-cut
			
original	kmeans++_init	kmeans++_init	kmeans++_init
			
	random_init	random_init	random_init
			
original	kmeans++_init	kmeans++_init	kmeans++_init

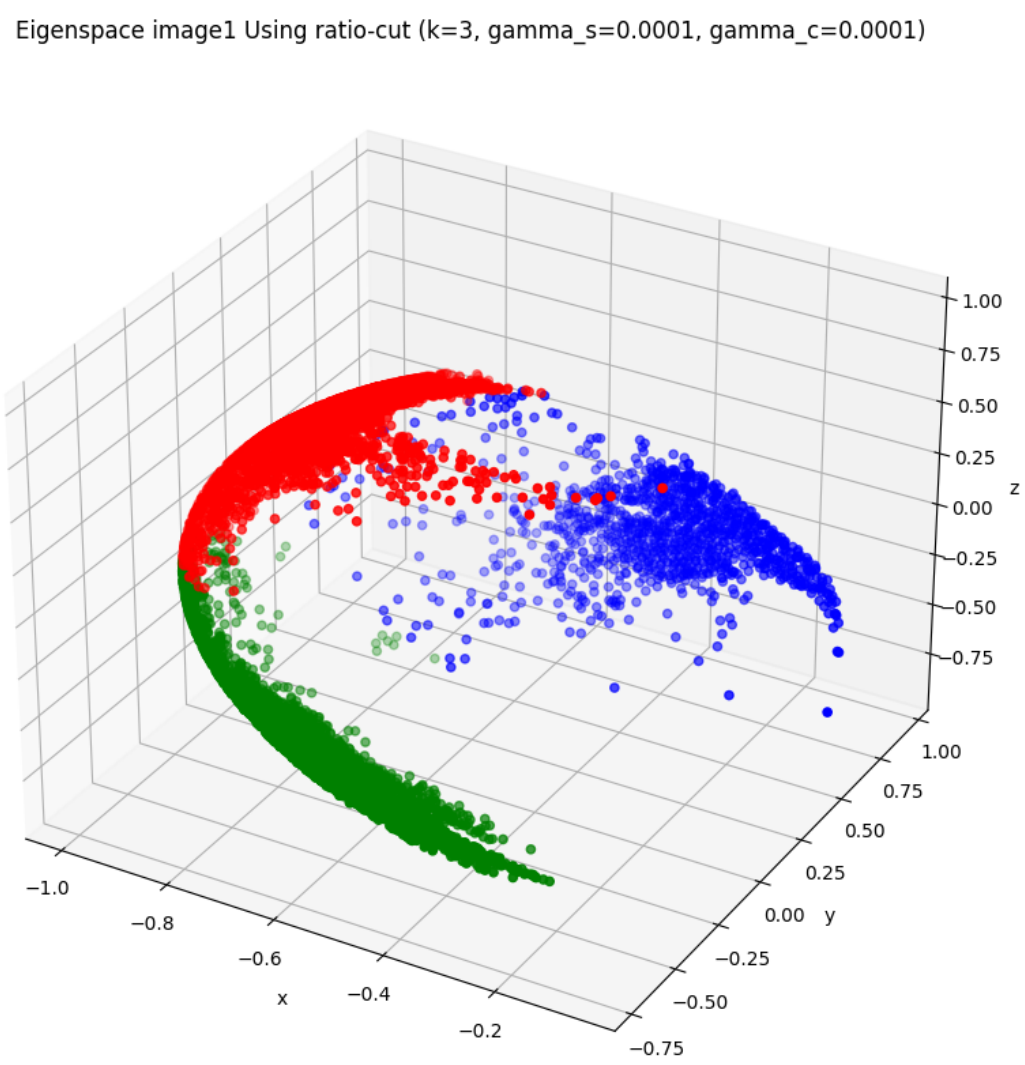
original image	kernel kmeans	ratio-cut	normalized-cut
			
	random_init	random_init	random_init

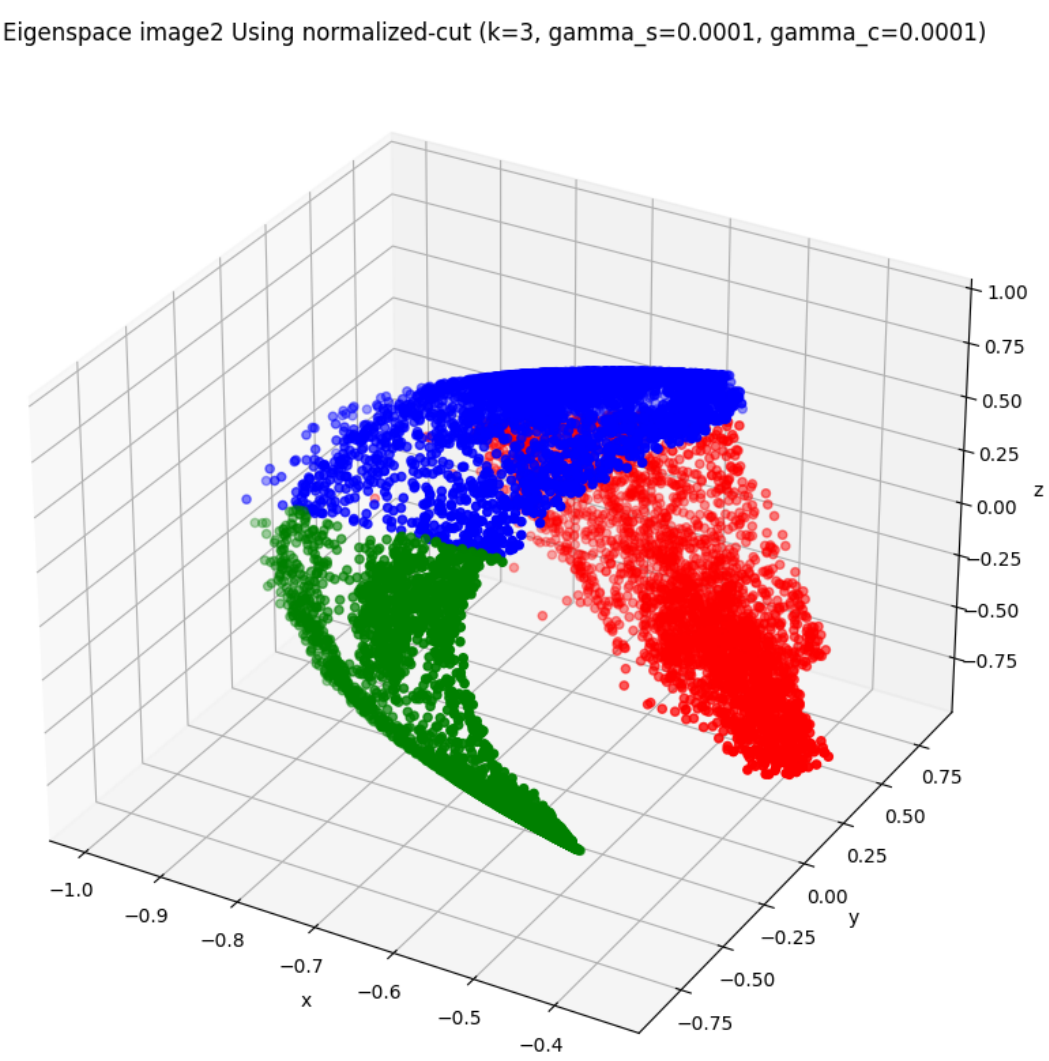
Averagely, the iteration number of kmeans++ were less than random assigned method.

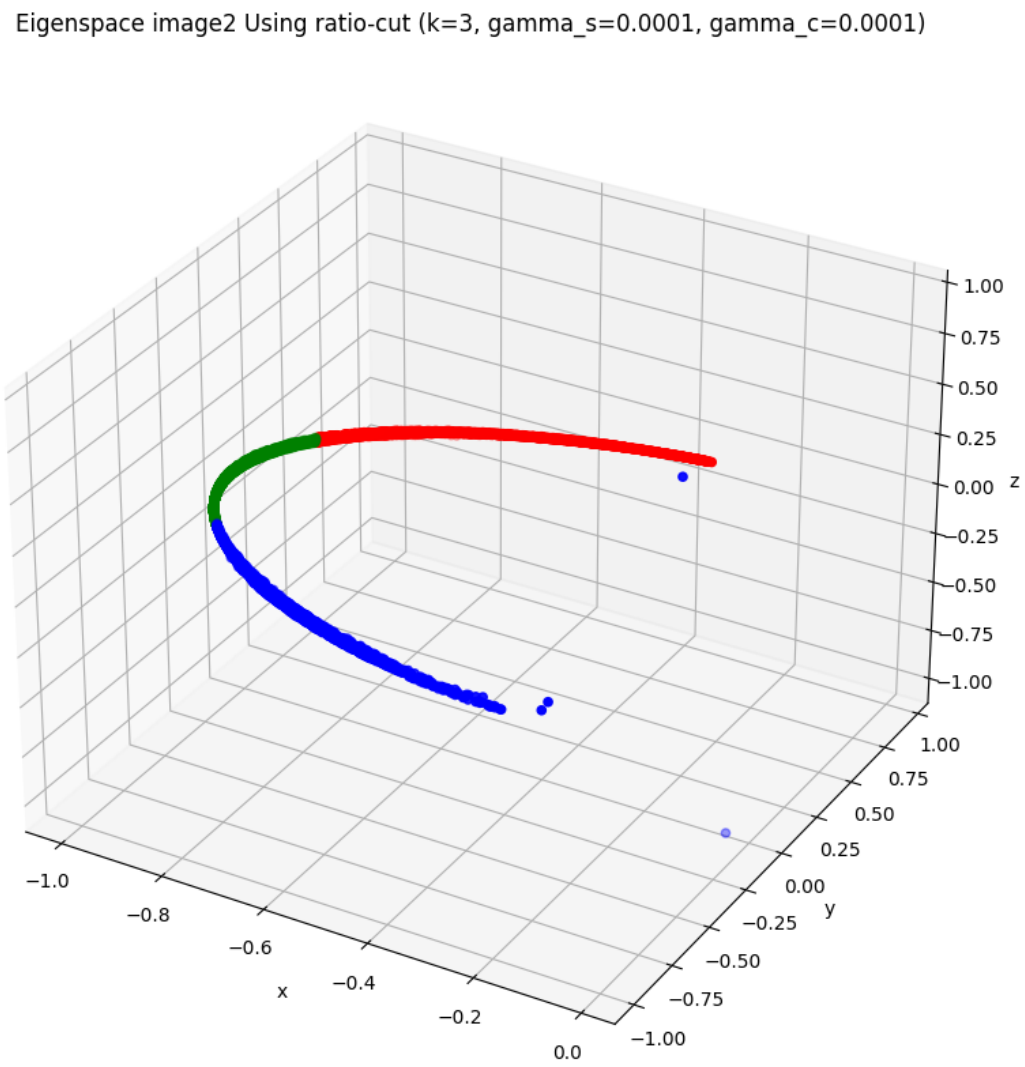
Part 4

This part we were going to plot the eigen space of the (normalized) graph Laplacian matrix and examine whether the data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian or not




Eigenspace image1 Using ratio-cut (k=3, gamma_s=0.0001, gamma_c=0.0001)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
z
−1.0
−0.8
−0.6
−0.4
−0.2
x
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
y


Eigenspace image2 Using normalized-cut (k=3, gamma_s=0.0001, gamma_c=0.0001)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
z
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
y
−1.0
−0.9
−0.8
−0.7
−0.6
−0.5
−0.4
x



Observations and Discussion

Different kernel kmeans implementation

Different eigenvectors

first k eigenvectors

second to the k'th eigenvectors

second to the (k+1)'th eigenvectors