ML Homework 6 Report

陳宥銓
313554013
Institute of Data Science and Engineering

Contents

1 Code

1.1 Kernel

We use the kernel defined below to compute the Gram matrix by multiplying two RBF kernels to account for both spatial similarity and color similarity

$$k(x, x') = e^{-\gamma_s \|S(x) - S(x')\|^2} \times e^{-\gamma_c \|C(x) - C(x')\|^2} \tag{1}$$

where $S(x)$ is the spatial information of data x, $C(x)$ is the color information of data x, γ_s controls the spatial similarity and γ_c controls the color similarity between data points.

```python
def spatial_color_RBFkernel(X, cols, rows, opt):
    gamma_s, gamma_c = opt["gamma_s"], opt["gamma_c"]
    coords = np.array([[i, j] for i in range(rows) for j in range(cols)])

    # Compute squared distances for spatial and color features
    spatial_dists = cdist(coords, coords, metric='sqeuclidean')  # 10000x10000
    color_dists = cdist(X, X, metric='sqeuclidean')
    # Compute the kernel matrix
    kernel_matrix = np.exp(-gamma_s * spatial_dists) * np.exp(-gamma_c * color_dists)
    return kernel_matrix
```

Figure 1: Kernel

1.2 Initialize the Means

To classify the data into K clusters, we use the following two methods to initialize the means.

1. **Random**:
   Randomly pick K points as the centers.

2. **Kmeans++**
   Randomly pick one point as the initial center, then iteratively select the farthest point from the centers already chosen as the next center. Repeat this process until K points are selected.

```python
""" Initialize  Mean """
def init_means(data, opt):
    METHOD = opt["init_method"]
    K = opt["num_clusters"]
    N = len(data)

    if METHOD == "random":
        # Randomly select K unique indices as initial cluster centers
        initial_indices = random.sample(range(N), K)
        initial_centers = data[initial_indices]
    if METHOD == "kmeans++":
        centers_indices = [random.randint(0, N - 1)]  # Random index as the first center
        initial_centers = [data[centers_indices[0]]]
        # Select remaining K-1 centers
        for _ in range(1, K):
            # Compute squared distances to the nearest selected center
            dist = np.min(cdist(data, np.array(initial_centers), metric='euclidean'), axis=1)

            # Normalize distances to get probabilities
            prob = dist / np.sum(dist)

            # Sample a new center index based on the probability
            new_center = np.random.choice(N, p=prob)
            centers_indices.append(new_center)
            initial_centers.append(data[new_center])

    # Compute distances from all data points to the initial centers
    distances = cdist(data, initial_centers, metric='euclidean')
    # Assign each point to the closest cluster
    initial_assignments = np.argmin(distances, axis=1)
    return np.array(initial_centers), np.array(initial_assignments)
```

Figure 2: Initialize the Means

1.3 Kernel K-means

Fig. 3 shows the main function of the kernel k-means. First, the image with dimensions $n \times m \times 3$ is reshaped into a matrix of size $mn \times 3$ then use **spatial_color_RBFkernel** (Fig.1) function to compute the Gram matrix. After computing the Gram matrix M_k, we apply **init_means** (Fig.2) function to find the initial centers. Next, the following kernel k-means algorithm is executed using the **kernel_kmeans** function (Fig. 4).

1. Classify all data points according to closet μ_k
2. Re-compute the mean μ_k of the points in cluster C_k

$$
\begin{aligned}
&\|\phi(x_j) - \mu_k^{\phi}\| \\
=&\|\phi(x_j) - \frac{1}{|C_k|}\sum_{n=1}^{N}\alpha_{kn}\phi(x_n)\| \\
=&k(x_j, x_j) - \frac{2}{|C_k|}\sum_{n}\alpha_{kn}k(x_j, x_n) + \frac{1}{|C_k|^2}\sum_{p}\sum_{q}\alpha_{kp}\alpha_{kq}k(x_p, x_q)
\end{aligned}
$$

   where if the data point x_n is assigned to the k-th cluster C_k, then $\alpha_{kn} = 1$

3. Repeat 1.and 2. until no change.

```python
def main(opt):
    # Load image and reshape its size into (rows * cols, 3)
    img = cv2.imread(opt["img_dir"])
    rows, cols, _ = img.shape # 100x100x3
    num_pixel = rows * cols # 10000
    data = img.reshape(num_pixel, 3)

    # Compute kernel
    kernel_mat = spatial_color_RBFkernel(data, cols, rows, opt)

    _, init_C = init_means(data, opt)

    star_time = time.time()
    C = kernel_kmeans(kernel_mat, init_C, cols, rows, opt)
    end_time = time.time()
    run_time = end_time - star_time
    make_vedio(opt, run_time)
```

Figure 3: Main Function of Kernel K-means

```python
""" Clustering """
def kernel_kmeans(kernel_mat, initial_cluster, cols, rows, opt):
    max_iter = 100
    iterate = 1
    K = opt["num_clusters"]
    N = kernel_mat.shape[0]

    C = initial_cluster.copy()
    save_procedure(0, C, cols, rows, opt)

    prev_C = np.zeros_like(C)
    while(iterate < max_iter):
        print('iter ', iterate)
        # Compute distances to each cluster
        dist = np.zeros((N, K))
        for k in range(K):
            idx = np.where(C==k)[0] # Get indices of points in cluster k
            if len(idx) == 0:
                continue
            C_size = len(idx) # Number of points in cluster k
            # k(x_j, x_j)
            dist[:, k] += kernel_mat.diagonal()
            # (- 2 / |C_k|) * Sum_n a_kn k(x_j, x_n)
            dist[:, k] += (-2 / C_size) * np.sum(kernel_mat[:, idx], axis=1)
            # (1 / |C_k|^2) * Sum_pq a_kp a_kq k(x_p, x_q)
            dist[:, k] += (1 / (C_size ** 2)) * np.sum([kernel_mat[i, j] for i in idx for j in idx])

        # Assign clusters
        C = np.argmin(dist, axis=1)
        if(norm(C - prev_C) < 1e-3):break # Ckeck convergence
        save_procedure(iterate, C, cols, rows, opt)
        prev_C = C.copy()
        iterate += 1
    return C
```

Figure 4: Function of Kernel K-means Clustering

1.4 Spectral K-means

1.4.1 Laplacian Matrix and Laplacian Embedding

Fig.5 shows the function for computing the Laplacian matrix and performing Laplacian embedding. In **laplacian** function, we first construct the degree matrix D, which is a diagonal matrix where each diagonal element represents the sum of the corresponding row in W, then compute the unnormalized Laplacian L using $L = D - W$ and normalized Laplacian L_{sym} using $L_{sym} = D^{-1/2}LD^{-1/2}$. In **embedding** function, we solve the eigenvalue problem for L, then sort the eigenvalues in ascending order and arrange the eigenvectors accordingly. Subsequently, we select the eigenvectors corresponding to the smallest K non-zero eigenvalues. These selected eigenvectors form the Laplacian embedding.

```python
def laplacian(W, opt):
    D = np.diag(np.sum(W, axis=1))
    L = D - W
    if opt["cut"] == 'ratio':
        return L
    elif opt["cut"] == 'normalized':
        D_inv_sqrt = np.diag(1.0 / np.sqrt(np.diag(D))) # D^-1/2
        L_sym = D_inv_sqrt @ L @ D_inv_sqrt # L_sym = D^-1/2 L D^-1/2
        return L_sym, D_inv_sqrt

def embedding(L, opt):
    K = opt["num_clusters"]
    npy_path = opt["save_path"]
    try:
        eigenvalues = np.load(os.path.join(npy_path, "eigenvalues.npy"))
        eigenvectors = np.load(os.path.join(npy_path, "eigenvectors.npy"))
    except:
        # Solve eigen problem
        eigenvalues, eigenvectors = np.linalg.eigh(L)
        np.save(os.path.join(npy_path, "eigenvalues.npy"), eigenvalues)
        np.save(os.path.join(npy_path, "eigenvectors.npy"), eigenvectors)

    idx = np.argsort(eigenvalues)
    eigenvectors = eigenvectors[:, idx]

    # Select the first k eigenvectors
    embedding = eigenvectors[:, 1:K+1].real
    return embedding
```

Figure 5: Laplacian Matrix and Laplacian Embedding

1.4.2 Main Function

Fig.6 shows the main function of the spectral clustering, we compute the kernel in the same way as in kernel k-means, then compute the Laplacian matrix using **laplacian** function and call **embedding** function to solve the eigenvalue problem for the Laplacian matrix and generate the low-dimensional embedding. This embedding is subsequently used to initialize the cluster centers (Fig.2) and perform k-means clustering(Fig.7).

```python
def main(opt):
    # Load image and reshape its size into (rows * cols, 3)
    img = cv2.imread(opt["img_dir"])
    rows, cols, _ = img.shape # 100x100x3
    num_pixel = rows * cols # 10000
    data = img.reshape(num_pixel, 3)

    # Compute kernel
    kernel_mat = spatial_color_RBFkernel(data, cols, rows, opt)

    if opt["cut"] == 'normalized':
        L_sym, D_inv_sqrt = laplacian(kernel_mat, opt)
        H = embedding(L_sym, opt)
        H = D_inv_sqrt @ H
        init_centers, init_clusters = init_means(H, opt)
    elif opt["cut"] == 'ratio':
        L = laplacian(kernel_mat, opt)
        H = embedding(L, opt)
        init_centers, init_clusters = init_means(H, opt)

    star_time = time.time()
    C = kmeans(H, init_centers, init_clusters, cols, rows, opt)
    end_time = time.time()
    run_time = end_time - star_time
    make_vedio(opt, run_time)

    plot_eigenspace(H, C, opt)
```

Figure 6: Main Function of Spectral K-means

```python
""" Clustering """
def kmeans(H, init_centers, init_clusters, cols, rows, opt):
    max_iter = 100
    iterate = 1
    K = opt["num_clusters"]

    centers = init_centers.copy()
    C = init_clusters.copy()
    save_procedure(0, C, cols, rows, opt)

    prev_C = np.zeros_like(C)
    while(iterate < max_iter):
        print('iter ', iterate)
        for k in range(K):
            idx = np.where(C==k)[0] # Get indices of points in cluster k
            if len(idx) == 0:
                continue
            # Compute new centers
            centers[k] = np.mean(H[idx], axis=0)
        # Compute distances to each centers
        dist = cdist(H, centers, metric='euclidean')
        # Assign clusters
        C = np.argmin(dist, axis=1)
        if(norm(C - prev_C) < 1e-3):break # Check convergence
        save_procedure(iterate, C, cols, rows, opt)
        prev_C = C.copy()
        iterate += 1

    return C
```

Figure 7: K-means

1.5 Visualization

To visualize the results, we save a figure for each step of the procedure using **save_procedure** function. We then call the **make_video** function to combine these saved images into a GIF, which animates the clustering process over multiple iterations. In spectral clustering, we plot the coordinates in the eigenspace of graph Laplacian using **plot_eigenspace** function.

```python
def make_video(opt, run_time):
    fig_path = opt["save_path"]
    fig_list = sorted([os.path.join(fig_path, f) for f in os.listdir(fig_path) if f.endswith(('.png'))])
    imgs = [Image.open(i) for i in fig_list]
    imgs[0].save(os.path.join(fig_path, f"output_{opt['fold_name']}_{run_time}.gif"),
                 save_all=True,
                 append_images=imgs[1:],
                 duration=500,
                 loop=0)

def save_procedure(iterate, C, cols, rows, opt):
    output_path = opt["save_path"]
    color_map = opt["color_map"]
    fig = np.zeros((len(C), 3), np.uint8)
    for i in range(len(C)):
        fig[i] = color_map[C[i]]

    fig = fig.reshape(cols, rows, 3)
    plt.imsave(os.path.join(output_path, f'{iterate}.png'), fig)

def plot_eigenspace(H, C, opt):
    output_path = opt["save_path"]
    color_map = np.array(opt["color_map"])
    color_map = color_map / 255
    if opt["num_clusters"] == 2:
        for i in range(2):
            x = H[np.where(C==i)[0], 0]
            y = H[np.where(C==i)[0], 1]
            plt.scatter(x, y, c=[color_map[i]], s=0.5)
        plt.savefig(os.path.join(output_path, f"output_{opt['fold_name']}_eigen.png"))
    elif opt["num_clusters"] == 3:
        fig = plt.figure()
        ax = fig.add_subplot(projection='3d')
        for i in range(3):
            x = H[np.where(C==i)[0], 0]
            y = H[np.where(C==i)[0], 1]
            z = H[np.where(C==i)[0], 2]
            ax.scatter(x, y, z, c=[color_map[i]], s=0.5)
        plt.savefig(os.path.join(output_path, f"output_{opt['fold_name']}_eigen.png"))
```

Figure 8: Visualization

2 Results

2.1 Kernel K-means

In kernel k-means, we set the hyperparameters $(\gamma_s,\ \gamma_c) = (10^{-4},\ 10^{-5})$. Fig.9 shows the results of classifying into K $(K = 2, 3, 4)$ clusters using kernel k-means, while Fig.10 shows the results of the initial and final clusters obtained when classifying the data into 2 clusters, comparing the random and k-means++ initialization methods. We found that, for both Image 1 and Image 2, the results of classifying into four clusters are better than those obtained with only two clusters. Additionally, we observed that using k-means++ for initialization generally leads to more stable and accurate clustering compared to the random initialization method since random initialization method might lead to poor selection of initial centers, while K-means++ ensures that the initial centers are more spread out, improving the overall clustering performance.

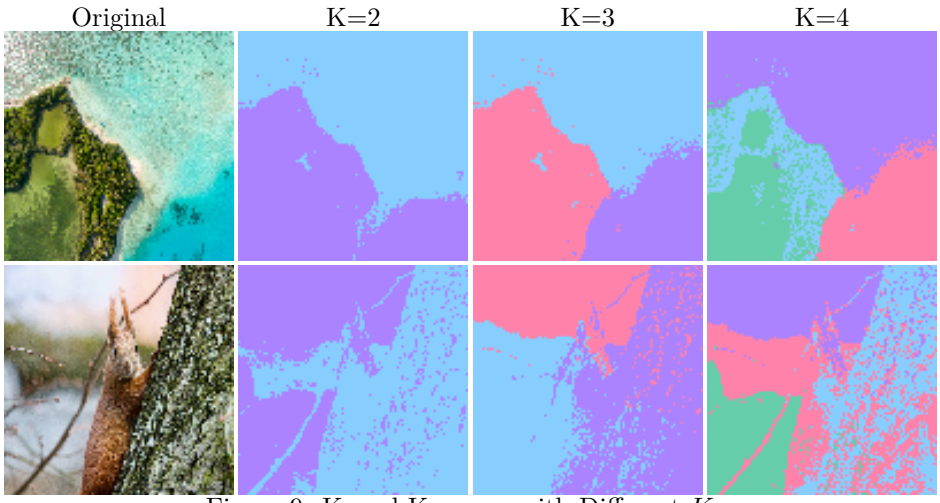


Figure 9: Kernel K-means with Different K

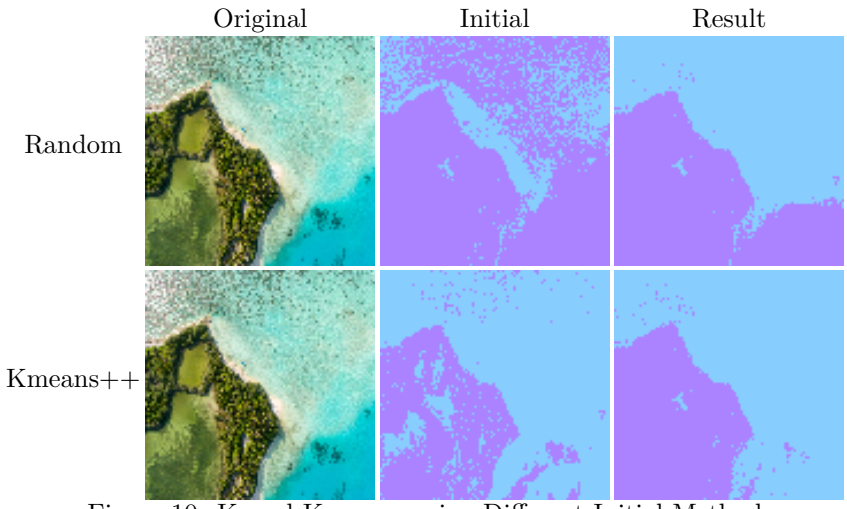


Figure 10: Kernel K-means using Different Initial Method

2.2 Spectral Clustering

In spectral clustering, we set $(\gamma_s,\ \gamma_c) = (10^{-4},\ 10^{-3})$. Fig.11 shows the results of classifying into K $(K = 2, 3)$ clusters using ratio cut and normalized cut of spectral clustering and plot their corresponding eigenspace, while Fig.12 shows the results of the initial and final clusters obtained when classifying the data into 2 clusters, comparing the random and k-means++ initialization methods. We found that, same as kernel k-means, the performance of k-means++ in selecting initial centers is better than random initialization for both normalized cut and ratio cut in spectral clustering. This demonstrates that k-means++ provides a more effective initialization, leading to improved clustering outcomes.

2.3 GIF Files

The GIF images of the output result will be named in the following way and stored in the **Results_GIF** folder.
[METHOD]-[initial method]-[numbers of clusters]-[γ_s]-[γ_c]_[execution time].gif
For example, ratio-random-3-1e-04-1e-03_0.01.gif means the result of ratio cut using random initial with $K = 3,\ \gamma_s = 1e-4,\ \gamma_c = 1e-3$ and the execution time is 0.01.

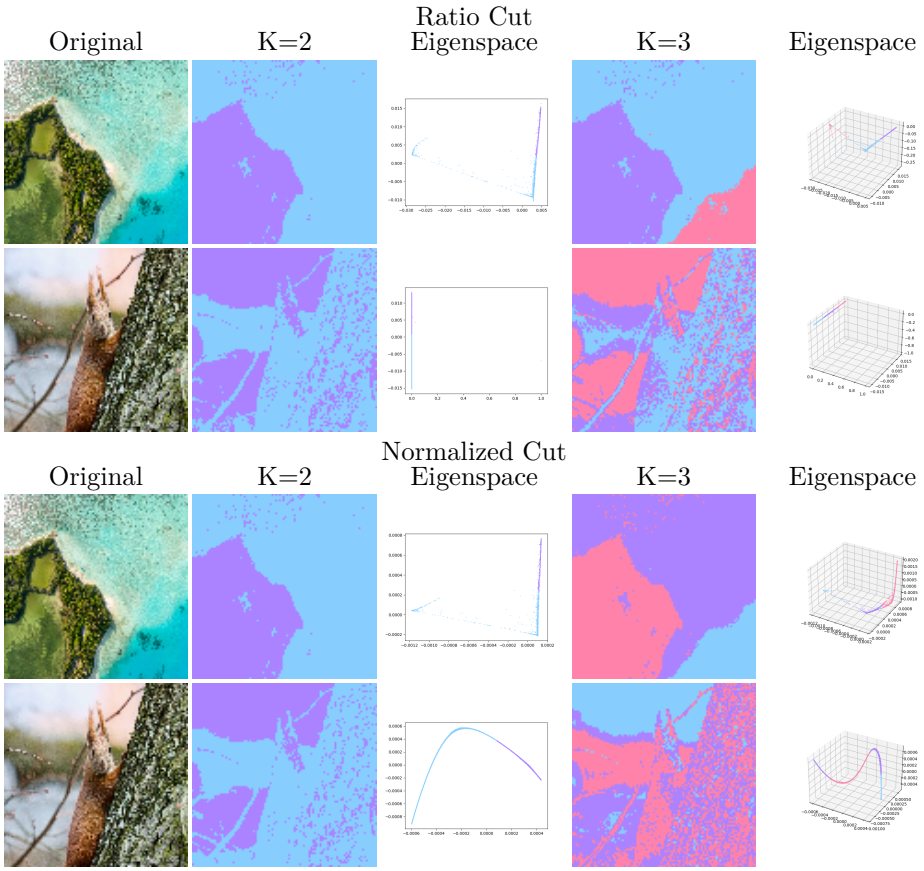


Figure 11: Spectral Clustering with Different K

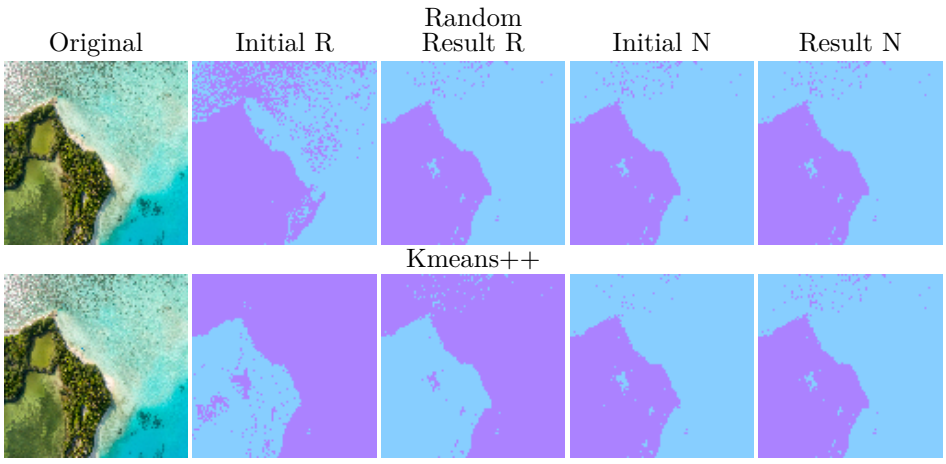
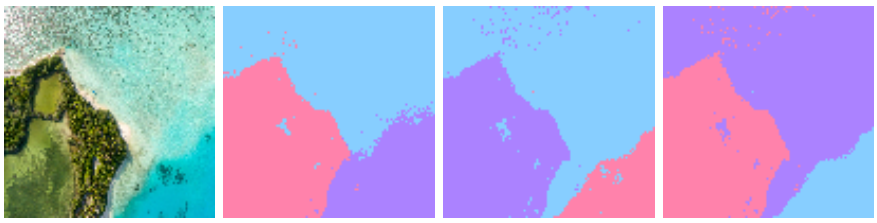


Figure 12: Spectral Clustering using Different Initial Method (R:ratio cut, N: normalized cut)

3 Observation

1. Fig.13 shows the results of different clustering methods. We found that spectral clustering performs better than kernel k-means since its ability to leverage the eigenspace of the graph Laplacian, which captures the underlying structure of the data more effectively.



(a) Original (b) Kernel K-means (c) Ratio Cut (d) Normalized Cut

Figure 13: Results of Different Clustering Method ($K = 3$)

2. Tab.1 shows the execution time of different methods for classify Image 1 with $K = 2$. According to the table, we observe that using k-means++ for initialization significantly reduces the execution time in kernel k-means compared to random initialization. However, in both ratio cut and normalized cut, the difference in execution time between random and k-means++ initialization is relatively small. Since we pre-calculated the eigenvectors of Laplacian matrix, this allows us to perform spectral clustering in significantly less time than kernel k-means.

	K-means	Ratio Cut	Normalized Cut
Random init	88.12	0.02	0.01
kmeans++	43.05	0.02	0.02

Table 1: Execution Time of Different Settings

3. The selection of RBF kernel hyperparameters (γ_c, γ_s) plays a crucial role in the performance of kernel k-means and spectral clustering. These parameters control the sensitivity of the kernel to spatial and color differences, respectively. Proper tuning of these hyperparameters is essential to balance spatial and color similarity, enabling the algorithm to cluster data accurately and effectively.