309551063 HW6 Report

a. code with detailed explanations

Part 1

the procedure of making gif image

```python
def make_gif_image(cluster_idx_history, input_image, k, final_center, gif_name='test'):
    """
    Use the center to colorize the same cluster, and use imageio to make gif file.
    :param cluster_idx_history: the cluster of each pixel in every iteration
    :param input_image: the input image (shape: (10000, 3))
    :param k: the number of clusters
    :param final_center: the center after trained
    :param gif_name: output gif file name
    :return: None
    """
    # use imageio package to generate gif
    images_frame = list()
    for idx in cluster_idx_history:
        image_frame = np.zeros(input_image.shape, dtype=np.uint8)
        for c_idx in range(k):
            # use the center to colorize the same cluster
            image_frame[np.where(idx == c_idx)[0], :] = final_center[c_idx]
        images_frame.append(image_frame.reshape(100, 100, 3))
    imageio.mimsave(f'gif/kernel-k-means/{gif_name}.gif', images_frame)
    # use *_tmp.gif file to show the final result in report
    imageio.mimsave(f'gif/kernel-k-means/{gif_name}_tmp.gif', [images_frame[-1], images_frame[-1]])
```

kernel k-means clustering procedure

```python
def kernel_function(spatial_a, spatial_b, color_a, color_b, theta=None):
    """
    Calculate the kernel noticed in the spec.
    $k(x,x')=e^{\gamma_s||S(x)-S(x')||^2}\times e^{\gamma_c||C(x)-C(x')||^2}$
    :param spatial_a: The spatial information of x
    :param spatial_b: The spatial information of x'
    :param color_a: The color information of x
    :param color_b: The color information of x'
    :param theta: The hyper-parameters used in kernel.
                  A list, format: [gamma_s, gamma_c]. (default: [1e-5, 1e-5])
    :return: the mix kernel
    """
    if theta is None:
        theta = np.array([1e-5, 1e-5]).reshape(-1, 1)
    spatial_information = cdist(spatial_a, spatial_b, 'sqeuclidean')
    color_information = cdist(color_a, color_b, 'sqeuclidean')
    return np.exp(- theta[0] * spatial_information) * np.exp(- theta[1] * color_information)
```

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

```python
def initialize_label(input_image, k, method='random'):
    """
    Initialize the label of each pixel.
    :param input_image: the input image (shape: (10000, 3))
    :param k: the number of clusters
    :param method: There are two initialized method can choose, {'random', 'k-means++'}.
    (Default: 'random')
    :return: return the initialized label of each pixel
    """
    np.random.seed(666)
    if method == 'random':
        # random to initialize the label
        return np.random.randint(k, size=len(input_image))
    elif method == 'k-means++':
        """
        k-means++ algorithm
        step1: sample a pixel randomly to be center
        step2: calculate the minimum distance D(x) between pixel and the nearest center
        step3: choose a new center based on the probability of $\frac{D(x)^2}{\sum D(x)^2}$
        repeat step2 and step 3 until get enough center
        """
        center = list()
        center.append(np.random.randint(len(input_image)))
        total_distance = kernel_k_means_square_distance_point_point()
        while len(center) != k:
            distance = np.min(total_distance[center], axis=0)
            prob = distance / sum(distance)
            center.append(np.random.choice(len(input_image), 1, p=list(prob))[0])
        # Find the the nearest center of each pixel to get label
        cluster_idx = np.argmin(total_distance[center], axis=0)
        return cluster_idx
    else:
        # Undefined method
        raise ValueError('Unknown initialize label method!')
```

```python
def kernel_k_means_square_distance_point_point():
    """
    Calculate the square distance based on the kernel.
    $||\phi(x_i)-\phi(x_j)||=
    \phi(x_i)\phi(x_i)-2\phi(x_i)\phi(x_j)+\phi(x_j)\phi(x_j)=k(x_i,x_i)-2k(x_i,x_j)+k(x_j,x_j)$
    :return: the square distance between point and point (shape: (10000, 10000))
    """
    tmp = total_kernel.diagonal().reshape(-1, 1)
    return tmp - 2 * total_kernel + tmp
```

$$||\phi(x_i) - \phi(x_j)|| = \phi(x_i)\phi(x_i) - 2\phi(x_i)\phi(x_j) + \phi(x_j)\phi(x_j) = k(x_i, x_i) - 2k(x_i, x_j) + k(x_j, x_j)$$

```python
def kernel_k_means_distance_point_center(target_idx):
    """
    Use the formula in slide P22 to calculate the square distance between point and the center
    $||\phi(x_j)-\mu^{\phi}_k||=
    ||\phi(x_j)-\frac{1}{|C_k|}\sum_{n=1}^{N}\alpha_{kn}\phi(x_n)||
    =k(x_j,x_j)-\frac{2}{|C_k|}\sum_n\alpha_{kn}k(x_j,x_n)+
    \frac{1}{|C_k|^2}\sum_p\sum_q\alpha_{kp}\alpha_{kq}k(x_p,x_q)$
    :param target_idx: the index of the specified center
    :return: the square distance between point and the center
    """
    first_term = total_kernel.diagonal()
    second_term = 2 / len(target_idx)
    alpha = np.zeros(total_kernel.shape[0])
    alpha[target_idx] = 1
    second_term = second_term * np.dot(total_kernel.T, alpha)
    third_term = 1 / (len(target_idx) ** 2)
    third_term = third_term * np.sum(total_kernel[np.ix_(target_idx, target_idx)])
    return first_term - second_term + third_term
```

$$||\phi(x_j)-\mu^{\phi}k|| = ||\phi(x_j)-\frac{1}{|C_k|}\sum_{n=1}^{N}\alpha_{kn}\phi(x_n)|| = k(x_j,x_j)-\frac{2}{|C_k|}\sum_n\alpha_{kn}k(x_j,x_n)+\frac{1}{|C_k|^2}\sum_p\sum_q\alpha_{kp}\alpha_{kq}k(x_p,$$

```python
def kernel_k_means(input_image, gif_name, k=2, max_iter=100, tol=1e-4, method='random'):
    """
    kernel k-means algorithm
    step1: use initialized method to get the cluster of each pixel
    step2: calculate the distance between pixel and center
    step3: update the cluster of each pixel based on the nearest center
    repeat step2 and step3 until convergence or reach the max iteration
    :param input_image: the input image (shape: (10000, 3))
    :param gif_name: the output gif file name
    :param k: the number of clusters (default: 2)
    :param max_iter: the maximum iteration (default: 100)
    :param tol: the tolerance to check convergence (default: 1e-4)
    :param method: the center initialized method (default: 'random')
    :return: the cluster of each pixel
    """
    old_cluster_distance = np.array(
```

```
        [0 for _ in range(k)] * len(input_image)).reshape(-1, len(input_image))
    cluster_idx = initialize_label(input_image, k, method)
    # record the cluster of each pixel in every iteration to output gif
    cluster_idx_history = [cluster_idx]
    final_center = list()
    final_iter_count = 0
    for iter_count in range(max_iter):
        cluster_distance = list()
        for c_idx in range(k):
            cluster_distance.append(
                kernel_k_means_distance_point_center(np.where(cluster_idx == c_idx)[0]))
        cluster_distance = np.array(cluster_distance)
        cluster_idx = np.argmin(cluster_distance, axis=0)
        cluster_idx_history.append(cluster_idx)
        if np.linalg.norm(old_cluster_distance - cluster_distance) < tol:
            final_iter_count = iter_count + 1
```

```
            break
        else:
            old_cluster_distance = cluster_distance
    # find the final center
    for c_idx in range(k):
        final_center.append(np.mean(input_image[np.where(cluster_idx == c_idx)[0], :], axis=0))
    make_gif_image(cluster_idx_history, input_image, k, final_center,
                   gif_name=f'{gif_name}_{final_iter_count}')
    return cluster_idx
```

```
if __name__ == '__main__':
    # use argparse to get all argument
    parser = argparse.ArgumentParser()
    parser.add_argument("-i", "--image", default='image1.png',
                        help="input image: image1.png or image2.png (Default: image1.png)")
    parser.add_argument("-m", "--method", default='k-means++',
                        help="initialize method: k-means++ or random (Default: k-means++)")
    parser.add_argument("--theta1", default=1e-5, type=float,
                        help="hyper-parameters of spatial information (Default: 1e-5)")
    parser.add_argument("--theta2", default=1e-5, type=float,
                        help="hyper-parameters of color information (Default: 1e-5)")
    parser.add_argument("-k", "--cluster", default=2, type=int,
                        help="the number of clusters (Default: 2)")
    args = parser.parse_args()
    initialize_method = args.method
    image_name = args.image
```

```
    cluster = args.cluster
    theta1 = args.theta1
    theta2 = args.theta2
    # read image
    image = cv2.imread(image_name).reshape(-1, 3)
    spatial = list()
    for x in range(0, 100):
        for y in range(0, 100):
            spatial.append([x, y])
    spatial = np.array(spatial)
    output_name = f'{image_name}_{cluster}_{initialize_method}_{theta1}_{theta2}'
    # use spatial and image information to get kernel
    total_kernel = kernel_function(spatial, spatial, image, image, theta=[theta1, theta2])
    kernel_k_means(image, gif_name=output_name, k=cluster, method=initialize_method)

```

Brief summary of kernel k-means clustering procedure:

- step1: use `argparse` package to get the parameters
- step2: use spatial information and color information to get kernel
- step3: use initialized method to get the cluster of each pixel
- step4: calculate the distance between pixel and center
- step5: update the cluster of each pixel based on the nearest center
  (repeat step4 and step5 until convergence or reach the max iteration)
- step6: generate gif image file

spectral clustering (both normalized cut and ratio cut) clustering procedure

```
def initialize_label(input_image, k, method='random'):
    """
    Initialize the label of each pixel.
    :param input_image: the input image (shape: (10000, 3))
    :param k: the number of clusters
    :param method: There are two initialized method can choose, {'random', 'k-means++'}.
    (Default: 'random')
    :return: return the initialized label of each pixel
    """
    np.random.seed(666)
    if method == 'random':
        return np.random.randint(k, size=len(input_image))
    elif method == 'k-means++':
        """
        k-means++ algorithm
        step1: sample a pixel randomly to be center
```

```
        step2: calculate the minimum distance D(x) between pixel and the nearest center
        step3: choose a new center based on the probability of $\frac{D(x)^2}{\sum D(x)^2}$
        repeat step2 and step 3 until get enough center
        """
        center = list()
        center.append(np.random.randint(len(input_image)))
        total_distance = cdist(input_image, input_image, 'sqeuclidean')
        while len(center) != k:
            distance = np.min(total_distance[center], axis=0)
            prob = distance / sum(distance)
            center.append(np.random.choice(len(input_image), 1, p=list(prob))[0])
        # Find the the nearest center of each pixel to get label
        cluster_idx = np.argmin(total_distance[center], axis=0)
        return cluster_idx
    else:
        raise ValueError('Unknown method!')
```

```
def get_laplacian(method='ratio_cut'):
    """
    The weighted adjacency matrix $W=(w_{ij})_{i,j=1,...,n}$
    The degree: $d_i=\sum_{j=1}^{n}w_{ij}$
    The degree matrix D: the diagonal matrix with the degrees $d_1,...,d_n$ on the diagonal.
    The unnormalized Laplacian matrix L: $D - W$
    The normalized Laplacian matrix L: $D^{-1/2}LD^{-1/2}$
    (Source: A Tutorial on Spectral Clustering)
    :param method: The cut method used in spectral clustering.
    There are two methods can choose: {'normalized_cut', 'ratio_cut'} (Default: 'ratio_cut')
    :return: The Laplacian matrix
    """
    # unknown cut method
    if method not in ['normalized_cut', 'ratio_cut']:
        raise ValueError('Unknown cut method!')
    degree = np.sum(total_kernel, axis=0)
```

The weighted adjacency matrix $W = (w_{ij})i, j = 1, ..., n$
The degree: $d_i = \sum_{j=1}^{n} w_{ij}$
The degree matrix D: the diagonal matrix with the degrees $d_1, ..., d_n$ on the diagonal.
The unnormalized Laplacian matrix $L : D - W$
The normalized Laplacian matrix $L : D^{-1/2}LD^{-1/2}$
(Source: A Tutorial on Spectral Clustering)

```python
def plot_eigen_space(u, cluster_idx, final_center, k, file_name='test'):
    """
    Plot the eigen space of graph Laplacian.
    If k is 2, output the 2D scatter plot.
    If k is 3, output the 3D scatter plot.
    If k is bigger than 3, output the 1D scatter plot of each dimension.
    :param u: the first k eigenvectors(the eigenvectors corresponding to the k smallest eigenvalues)
    :param cluster_idx: the cluster of each pixel
    :param final_center: the center after trained
    :param k: the number of clusters
    :param file_name: output jpg file name
    :return: None
    """
    if k == 2:
        for c_idx in range(k):
            idx = np.where(cluster_idx == c_idx)[0]
            plt.scatter(u[idx, 0], u[idx, 1], c=tuple(final_center[c_idx] / 255))
    elif k == 3:
        fig = plt.figure()
        ax = fig.gca(projection='3d')
        for c_idx in range(k):
            idx = np.where(cluster_idx == c_idx)[0]
            ax.scatter(u[idx, 0], u[idx, 1], u[idx, 2], c=tuple(final_center[c_idx] / 255))
    else:
        fig, axs = plt.subplots(k, figsize=(15, 15))
        count = 0
        for i in range(k):
            axs[i].set_title(f'eigenspace - the {i} dimension')
            for c_idx in range(k):
                idx = np.where(cluster_idx == c_idx)[0]
                axs[i].scatter(range(count, count+len(idx)), u[idx, i],
                               c=tuple(final_center[c_idx] / 255))
                count = count + len(idx)
    plt.tight_layout()
    plt.savefig(f'gif/spectral-clustering/{file_name}_eigen_space.jpg')
```

```python
def k_means(input_image, u, gif_name, k=2, max_iter=100, tol=1e-4, method='random'):
    """
    k-means algorithm
    step1: use initialized method to get the cluster of each pixel
    step2: calculate the distance between pixel and center
    step3: update the cluster of each pixel based on the nearest center
    repeat step2 and step3 until convergence or reach the max iteration
    :param input_image: the input image (shape: (10000, 3))
    :param u: the first k eigenvectors of Laplacian
    :param gif_name: the output gif file name
    :param k: the number of clusters (default: 2)
    :param max_iter: the maximum iteration (default: 100)
    :param tol: the tolerance to check convergence (default: 1e-4)
    :param method: the center initialized method (default: 'random')
    :return: the cluster of each pixel
    """
    old_cluster_distance = np.array([0 for _ in range(k)] * len(u)).reshape(-1, len(u))
    cluster_idx = initialize_label(u, k, method)
    center = list()
    for _ in range(k):
        center.append(u[np.random.randint(len(u))])
    cluster_idx_history = [cluster_idx]
    final_center = list()
    final_iter_count = 0
    for iter_count in range(max_iter):
        cluster_distance = list()
        for c_idx in range(k):
            cluster_distance.append(cdist(u, center[c_idx].reshape(1, -1), 'euclidean').flatten())
        cluster_distance = np.array(cluster_distance)
        cluster_idx = np.argmin(cluster_distance, axis=0)
        cluster_idx_history.append(cluster_idx)
        if np.linalg.norm(old_cluster_distance - cluster_distance) < tol:
            final_iter_count = iter_count + 1
            break
        else:
            old_cluster_distance = cluster_distance
        for c_idx in range(k):
            center[c_idx] = np.mean(u[np.where(cluster_idx == c_idx)[0], :], axis=0)
    for c_idx in range(k):
        final_center.append(np.mean(input_image[np.where(cluster_idx == c_idx)[0], :], axis=0))
    make_gif_image(cluster_idx_history, input_image, k, final_center,
                   gif_name=f'{gif_name}_{final_iter_count}')
    plot_eigen_space(u, cluster_idx, final_center, k, file_name=f'{gif_name}_{final_iter_count}')
    return cluster_idx
```

```python
def spectral_clustering(input_image, k, gif_name,
                        spectral_clustering_method='ratio_cut', initialize_method='random'):
    """
    :param input_image: the input image (shape: (10000, 3))
    :param k: the number of clusters
    :param gif_name: the output gif file name
    :param spectral_clustering_method: The cut method used in spectral clustering.
    There are two methods can choose: {'normalized_cut', 'ratio_cut'} (Default: 'ratio_cut')
    :param initialize_method: There are two initialized method can choose, {'random', 'k-means++'}.
    (Default: 'random')
    :return: None
    """
    laplacian = get_laplacian(spectral_clustering_method)
    # compute eigenvectors
    if os.path.exists(f'{args.image}_{args.cut_method}_{theta1}_{theta2}_a.npy'):
        a = np.load(f'{args.image}_{args.cut_method}_{theta1}_{theta2}_a.npy')
        b = np.load(f'{args.image}_{args.cut_method}_{theta1}_{theta2}_b.npy')
        print('npy file loaded')
    else:
        a, b = np.linalg.eigh(laplacian)
        np.save(f'{args.image}_{args.cut_method}_{theta1}_{theta2}_a', a)
        np.save(f'{args.image}_{args.cut_method}_{theta1}_{theta2}_b', b)
    # find the first k eigenvectors(the eigenvectors corresponding to the k smallest eigenvalues)
    # and drop the first column
    u = b[:, a.argsort()[1:k+1]]
    if spectral_clustering_method == 'normalized_cut':
        # if spectral clustering method is 'normalized_cut', we need to normalize the rows to norm 1
        u = u / cdist(u, np.array([0] * k).reshape(1, -1), 'euclidean')
    return k_means(input_image=input_image, u=u, gif_name=gif_name, k=k, method=initialize_method)
```

```python
if __name__ == '__main__':
    # use argparse to get all argument
    parser = argparse.ArgumentParser()
    parser.add_argument("-i", "--image", default='image1.png',
                        help="input image: image1.png or image2.png (Default: image1.png)")
    parser.add_argument("--initialize_method", default='k-means++',
                        help="initialize method: k-means++ or random (Default: k-means++)")
    parser.add_argument("--cut_method", default='normalized_cut',
                        help="clustering method: normalized_cut or ratio_cut (Default: normalized_cut)")
    parser.add_argument("--theta1", default=0.001, type=float,
                        help="hyper-parameters of spatial information (Default: 0.001)")
    parser.add_argument("--theta2", default=0.001, type=float,
                        help="hyper-parameters of color information (Default: 0.001)")
    parser.add_argument("-k", "--cluster", default=2, type=int,
                        help="the number of clusters (Default: 2)")
    args = parser.parse_args()
```

```python
image_name = args.image
cluster = args.cluster
theta1 = args.theta1
theta2 = args.theta2
image = cv2.imread(image_name).reshape(-1, 3)
spatial = list()
for x in range(0, 100):
    for y in range(0, 100):
        spatial.append([x, y])
spatial = np.array(spatial)
total_kernel = kernel_function(spatial, spatial, image, image, theta=[theta1, theta2])
output_name = f'{args.image}_{args.cut_method}_{args.initialize_method}_{cluster}_{theta1}_{theta2
spectral_clustering(image, cluster, output_name,
                    spectral_clustering_method=args.cut_method,
                    initialize_method=args.initialize_method)
```

Brief summary of spectral clustering procedure:

- step1: use `argparse` package to get the arguments
- step2: use spatial information and color information to get kernel
- step3: calculate Laplacian
  The unnormalized Laplacian matrix $L : D - W$
  The normalized Laplacian matrix $L : D^{-1/2}LD^{-1/2}$
- step4: compute the first k eigenvectors of L
- step5: if spectral clustering method is `normalized_cut`, we need to normalize the rows to norm 1
- step6: Use the eigenvectors as data points to do k-means algorithm
- step7: generate gif image file and plot the eigenspace of graph Laplacian

Part 2

If we want to try more clusters, such as 4 clusters, we can run the program with argument `-k`.

- Example:
  `$ python kernel-k-means.py -k 4`
  `$ python spectral_clustering.py -k 4`

Part 3

For the initialization, I wrote a function `initialize_label` in order to get the initialized label of each pixel(data point). I implemented two methods, `random` and `k-means++`.

- `random`
  - random initialized label of each pixel
- `k-means++`
  - step1: sample a pixel randomly to be center
  - step2: calculate the minimum distance $D(x)$ between pixel and the nearest center
  - step3: choose a new center based on the probability of $\frac{D(x)^2}{\sum D(x)^2}$
    repeat step2 and step 3 until get enough center

The main difference of `kernel k-means` and `spectral clustering` in `k-means++` initialization is the distance calculation.

- `kernel k-means` : use kernel to calculate distance
- `spectral clustering` : use eigenvectors to calculate distance

Part 4

In order to plot the eigenspace of graph Laplacian, I wrote a function `plot_eigen_space`.

- If k is 2, output the 2D scatter plot.
- If k is 3, output the 3D scatter plot.
- If k is bigger than 3, output the 1D scatter plot of each dimension.

b. experiments settings and results & discussion

Part 1 & Part 2 & Part 3 results

In kernel k-means, the hyper-parameters setting: theta1=1e-5, theta2=1e-5

(file: gif/kernel-k-means/report/{image_name}_{k}_{initialization_method}_{theta1}_{theta2}_{iteration_times}.gif)

- image1 with kernel k-means

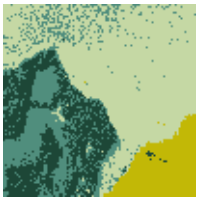
original

method: random (k=2, iteration: 9)

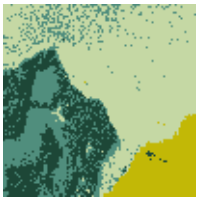
method: k-means++ (k=2, iteration: 9)

method: random (k=3, iteration: 11)

method: k-means++ (k=3, iteration: 10)



method: random (k=4, iteration: 23)



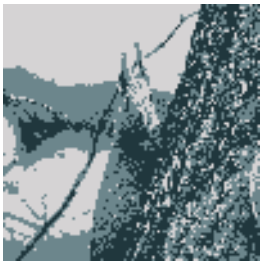
method: k-means++ (k=4, iteration: 24)

- image2 with kernel k-means

original

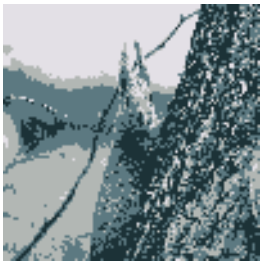
method: random (k=2, iteration: 13)

method: k-means++ (k=2, iteration: 12)

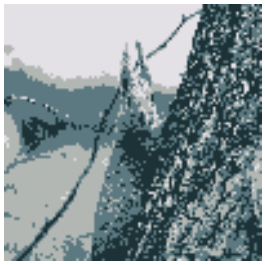


method: random (k=3, iteration: 20)

method: k-means++ (k=3, iteration: 17)



method: random (k=4, iteration: 60)



method: k-means++ (k=4, iteration: 20)

In spectral clustering, the hyper-parameters setting: theta1=0.001, theta2=0.001

(file: gif/spectral-clustering/report/{image_name}_{cut_method}_{initialization_method}_{k}_{theta1}_{theta2}_{iteration_times}.gif)

- image1 with spectral clustering(normalized cut)

original

method: random (k=2, iteration: 6)

method: k-means++ (k=2, iteration: 5)

method: random (k=3, iteration: 16)

method: k-means++ (k=3, iteration: 8)

method: random (k=4, iteration: 15)

method: k-means++ (k=4, iteration: 11)

- image1 with spectral clustering(ratio cut)

original

method: random (k=2, iteration: 6)

method: k-means++ (k=2, iteration: 6)

method: random (k=3, iteration: 6)

method: k-means++ (k=3, iteration: 6)

method: random (k=4, iteration: 28)

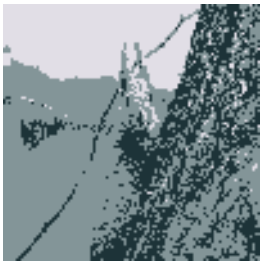
method: k-means++ (k=4, iteration: 16)

- image2 with spectral clustering(normalized cut)

original

method: random (k=2, iteration: 17)

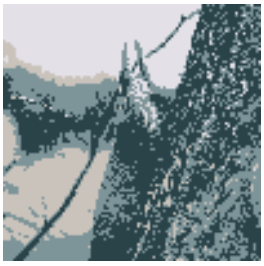
method: k-means++ (k=2, iteration: 6)



method: random (k=3, iteration: 16)

method: k-means++ (k=3, iteration: 16)

method: random (k=4, iteration: 18)



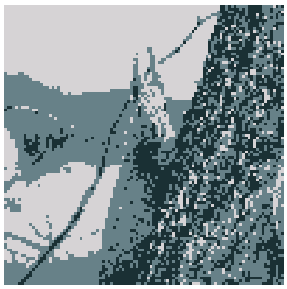
method: k-means++ (k=4, iteration: 27)

- image2 with spectral clustering(ratio cut)

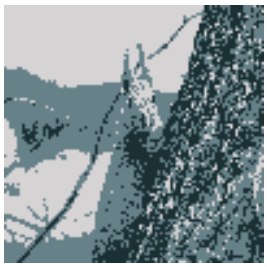
original

method: random (k=2, iteration: 6)

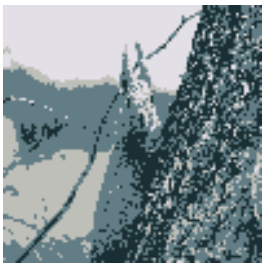
method: k-means++ (k=2, iteration: 10)



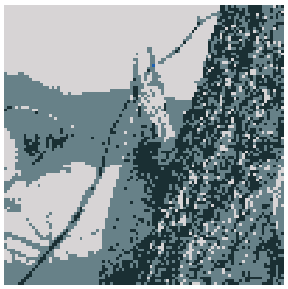
method: random (k=3, iteration: 13)



method: k-means++ (k=3, iteration: 11)



method: random (k=4, iteration: 13)

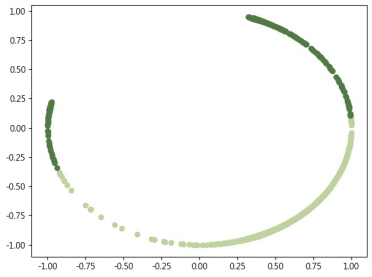


method: k-means++ (k=4, iteration: 15)

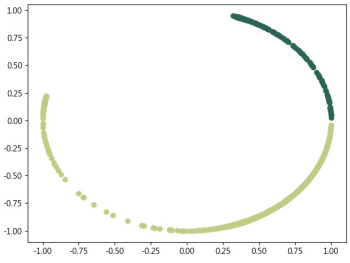
Part4 results

(file: gif/spectral-clustering/report/{image_name}_{cut_method}_{initialization_method}_{k}_{theta1}_{theta2}_{iteration_times}_eigen_spac

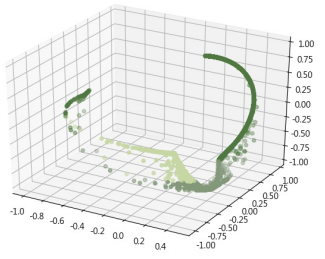
- image1 with spectral clustering(normalized cut)



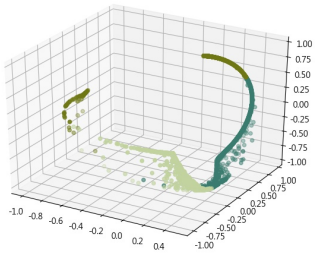
method: random (k=2, iteration: 6)



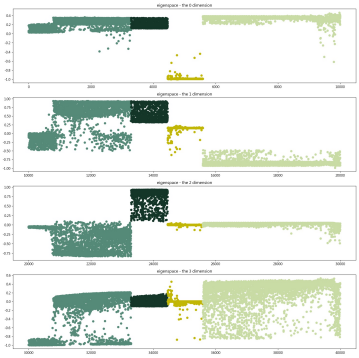
method: k-means++ (k=2, iteration: 5)



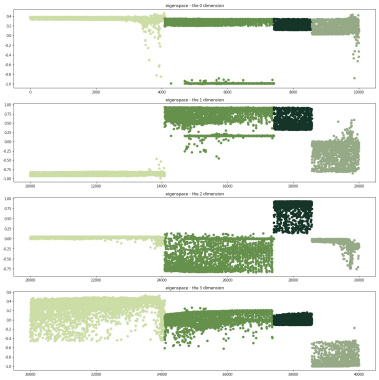
method: random (k=3, iteration: 16)



method: k-means++ (k=3, iteration: 8)

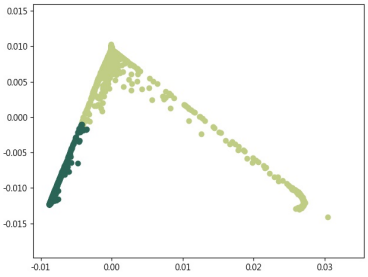


method: random (k=4, iteration: 15)

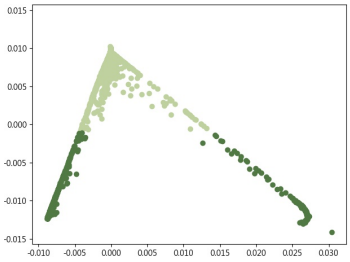


method: k-means++ (k=4, iteration: 11)

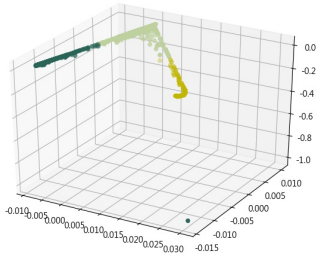
- image1 with spectral clustering(ratio cut)



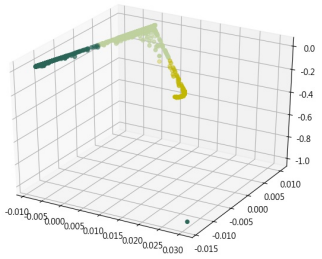
method: random (k=2, iteration: 6)



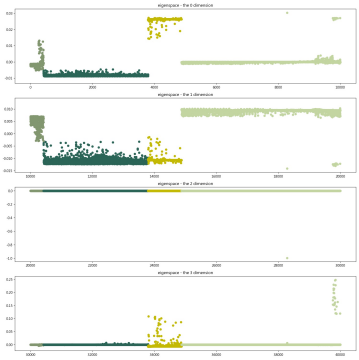
method: k-means++ (k=2, iteration: 6)



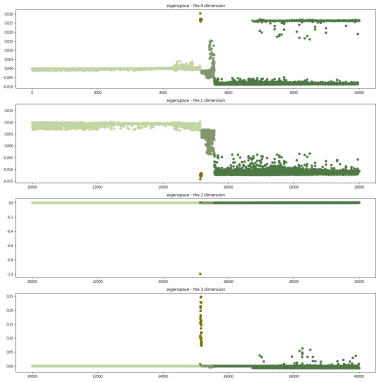
method: random (k=3, iteration: 6)



method: k-means++ (k=3, iteration: 6)

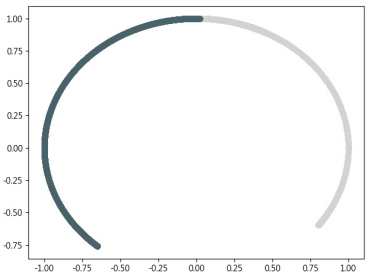


method: random (k=4, iteration: 28)

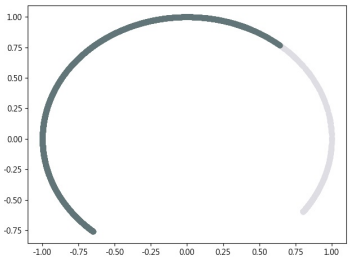


method: k-means++ (k=4, iteration: 16)

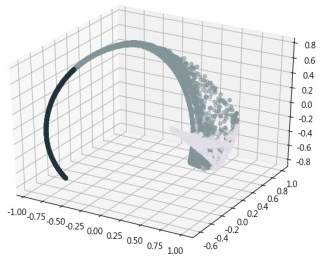
- image2 with spectral clustering(normalized cut)



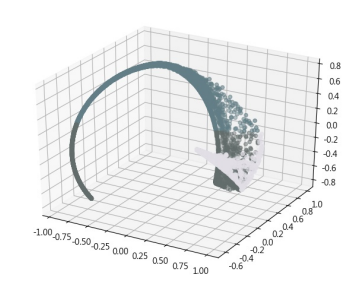
method: random (k=2, iteration: 17)



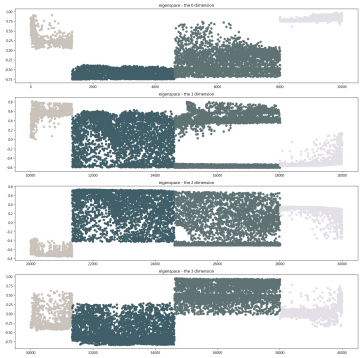
method: k-means++ (k=2, iteration: 6)



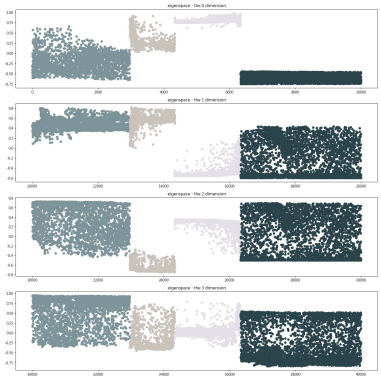
method: random (k=3, iteration: 16)



method: k-means++ (k=3, iteration: 16)

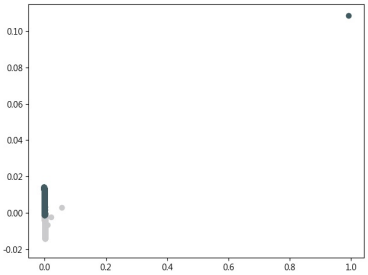


method: random (k=4, iteration: 18)

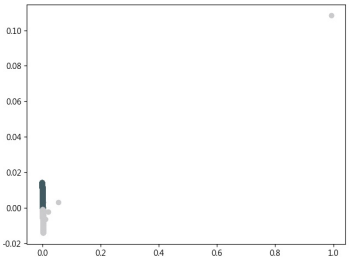


method: k-means++ (k=4, iteration: 27)

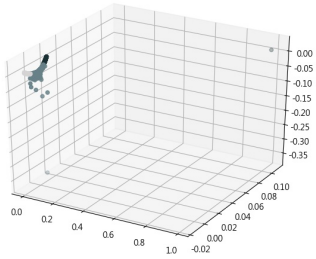
- image2 with spectral clustering(ratio cut)



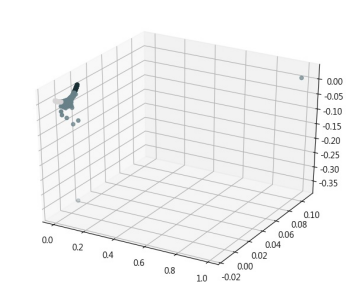
method: random (k=2, iteration: 6)



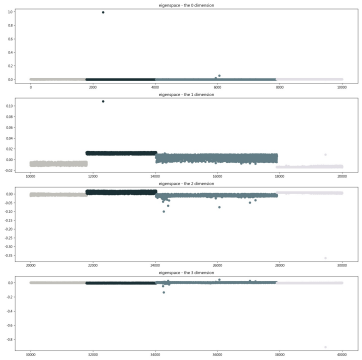
method: k-means++ (k=2, iteration: 10)



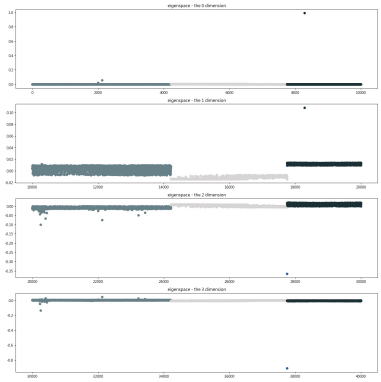
method: random (k=3, iteration: 13)



method: k-means++ (k=3, iteration: 11)



method: random (k=4, iteration: 13)



method: k-means++ (k=4, iteration: 15)

Part 1 discussion

When k=2, we can see that the results in image2 of all three method(kernel k-means, spectral clustering(normalized cut), and spectral clustering(ratio cut)) are quite similar. But the results in image1, there are some different in three method. In kernel k-means, we can see that the green dots in the sea are successfully classified into the same cluster with island. But in spectral clustering (both normalized cut and ratio cut), we can see that the results are quite different. The island and the sea are classified into the same cluster. I think the reason is that the hyper-parameters are different in kernel k-means and spectral clustering. When the hyper-parameters are setting to the same(theta1=1e-5, theta2=1e-5), the results are showed below.

normalized cut: method: random (k=2, iteration: 8

normalized cut: method: k-means++ (k=2, iteration: 11)

ratio cut: method: random (k=2, iteration: 6)

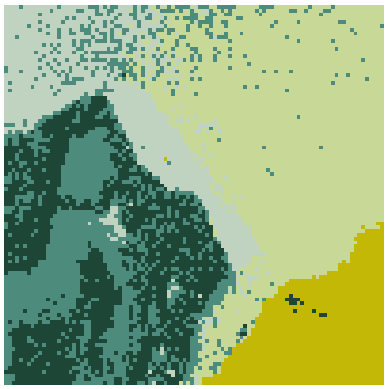
ratio cut: method: k-means++ (k=2, iteration: 15)

We can see that spectral clustering with normalized cut and with k-means++ initialization method can get the same clustering results with kernel k-means. But the result in spectral clustering with ratio cut are very strange, there are only two points clustered into the same cluster. So I think the hyper-parameters chosen are very important.

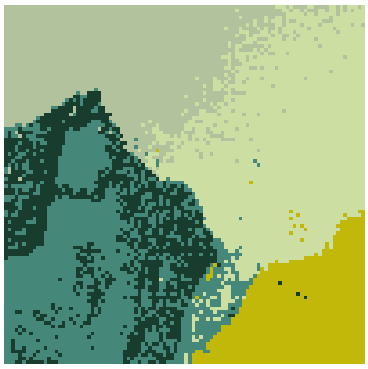
Part 2 discussion

When try more clusters, we need to more iterations to converge. We can see that when we use higher clusters, we can get more accurate clustering results. But it may overfit when k is too large. For example, when k=5, the results are below.

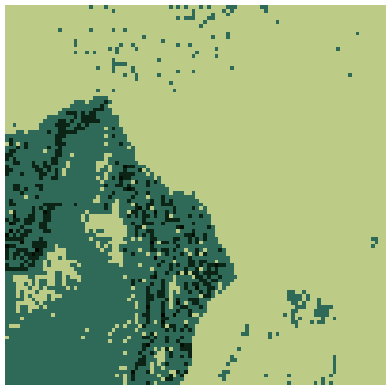
- image1



kernel k-means: method: k-means++ (k=5, iteration: 27)

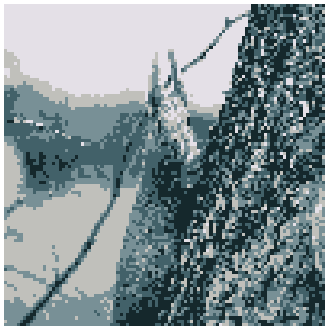


normalized cut: method: k-means++ (k=5, iteration: 28)

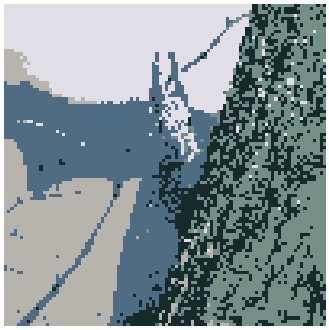


ratio cut: method: k-means++ (k=5, iteration: 42)

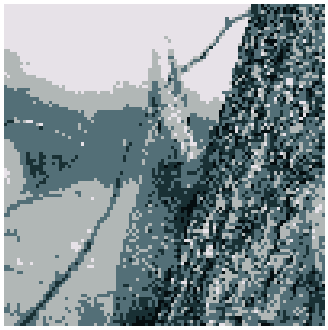
- image2



kernel k-means: method: k-means++ (k=5, iteration: 36)



normalized cut: method: k-means++ (k=5, iteration: 29)



ratio cut: method: k-means++ (k=5, iteration: 43)

There are some strange in the results with k=5. For example, in kernel k-means with k=5, there is a weird cluster in the left corner. In conclusion, we can see that the clustering results with k=5 is not better than the clustering result with k=4.

Part 3 discussion

We know that the initialization of k-means clustering is highly dependent to the final result, so the initialization of center is important. In part3, I implemented two methods, `random` and `k-means++`. Because we want to get the same result in each time, we use `np.random.seed` to fix the random seed. Overall, `k-means++` converges faster than `random` in particular when k is higher. There is an example in image2 with kernel k-means, when k=4, `random` needs 60 iterations but `k-means++` only needs 20 iterations. I think the reason is that the initialization center in `k-means++` is much closer to final centers.

But sometimes `k-means++` does not perform well as we expected. An example in image2 with spectral clustering(normalized cut), when k=2, although the iteration times in `k-means++` is much smaller than `random` , the clustering result is `random` better than `k-means++`. I think the reason is that `k-means++` is trapped in the local minimum.

Part 4 discussion

Although we have same eigenvecetors in the same setting(k=2, theta1=0.001, theta2=0.001), the clustering results are influenced by the k-means initialization method. So it would cause different results.

Also, we can see that there is often have extreme values in ratio cut, but normalized cut doe not have any extreme values. I think the reason is that we have already normalize the rows to norm 1. In conclusion, the normalize cut of results are more stable because it does not affected by extreme values.

c. observations and discussion

- In spectral clustering, we need to solve the eigenvalue problem. First of all, I use `np.linalg.eig` to get the eigenvalues and eigenvectors, but it is very slow. After some survey, `np.linalg.eigh` uses a faster algorithm when the matrix is symmetric. We know that our kernel function is symmetric, so I used `np.linalg.eigh` and save the eigenvalues and eigenvectors to speed up.
  - calculation time used `np.linalg.eig`: 527.00s
  - calculation time used `np.linalg.eigh`: 82.72s
- Hyper-parameters tuning: at first, I wrote a function `silhouette_score` to tune the hyper-parameters in kernels(theta1 and theta2). `silhouette_score` is one of the evaluation in unsupervised learning, the range from -1 to +1. The higher silhouette score means the better clustering model. The calculation and the code are shown below.

$$
\begin{aligned}
a(i) &= \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i,j) \\
b(i) &= \min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(i,j) \\
s(i) &= \frac{b(i) - a(i)}{max(a(i), b(i))} if |C_i| > 1 \\
silhouette - score &= mean(s)
\end{aligned}
$$

```
94  def silhouette_score(k, cluster_idx):
95      """
96      one of the evaluation in unsupervised learning, the range from -1 to +1.
97      The higher silhouette score means the better clustering model.
98      $a(i)=\frac{1}{|C_i|-1}\sum_{j\in C_i,i\neq j}d(i,j)$
99      $b(i)=\min_{k\neq i} \frac{1}{|C_k|}\sum_{j\in C_k}d(i,j)$
100     $s(i)=\frac{b(i)-a(i)}{max(a(i), b(i))} if |C_i| >1$
101     $silhouette-score = mean(s)$
102     :param k: the number of clusters
103     :param cluster_idx: the cluster of each pixel
104     :return: silhouette score
105     """
106     a = list()
107     tmp = list()
108     distance = np.triu(kernel_k_means_square_distance_point_point() ** 0.5)
```

```
109     for c_idx in range(k):
110         a.append(np.sum(distance[np.where(cluster_idx == c_idx)[0]]) /
111                  (len(np.where(cluster_idx == c_idx)[0]) - 1))
112         tmp.append(np.sum(distance[np.where(cluster_idx == c_idx)[0]]) /
113                    (len(np.where(cluster_idx == c_idx)[0])))
114     b = list()
115     for c_idx in range(k):
116         tmp2 = tmp.copy()
117         tmp2[c_idx] = np.inf
118         b.append(np.min(tmp2))
119     s = list()
120     for c_idx in range(k):
121         if len(np.where(cluster_idx == c_idx)[0]) > 1:
122             s.append((b[c_idx] - a[c_idx]) / max(a[c_idx], b[c_idx]))
123         else:
```

```
124             s.append(0)
125     return np.mean(s)
```

After tuning, we can see the best results below.

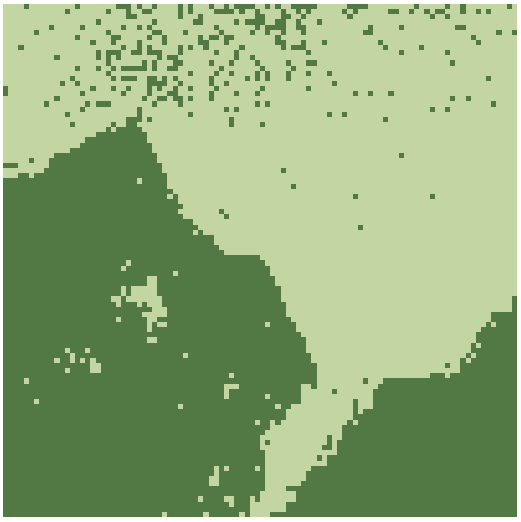


image1 with kernel k-means: method: k-means++ (k=2, iteration: 8, theta1: 2^(-19.5), theta2:2^(-15.5))

image2 with kernel k-means: method: k-means++ (k=2, iteration: 19, theta1: 2^(-14), theta2: 2^(-19))

  - We can see that image1 results are clustered perfectly, but image2 results are very bad. I think the problem is in the `silhouette_score` calculation. I use kernel distance to replace $d(i,j)$ in the equation, but kernel distance are highly dependant to theta1 and theta2. When we change the theta1 and theta2, the kernel distance will be also changed. There is no same distance can be used, which means the criterion of each hyper-parameters are not the same. So finally, I abandon this method because I think it is not work in hyper-parameters tuning.
- In clustering, there is an important question: How to determine the optimal number of clusters? In our case, we have already showed all results of k=2~5. As part2 discussion, we know k=4 is the best choice in our case. But there are some popular techniques to find the best k, for example: elbow method, and Silhouette Curve.