HW6 report

309552026

鄭偉丞

a. Code explanation:

K-means:

```
if __name__ == "__main__":
    gamma_C = 1e-3
    gamma_S = 1e-3

    filename = input("Filename: ")
    k = int(input("number of clusters: "))
    mode = int(input("0(k-means) or 1(k-means++): "))

    dataC, dataS, image_size = load(filename)
    Gram = kernel(gamma_S, gamma_C, dataS, dataC)

    history = kmeans(Gram, k, mode)
    visualize(history, image_size, filename, k, mode)
```

The main procedure of this part is 1. Generate Gram Matrix by kernel function as below and then 2. Do k-means procedure. 3. Show gif in function visualize.

```
def kernel(gamma_S, gamma_C, S, C):
    result = np.exp(-gamma_S*cdist(S, S, 'sqeuclidean'))
    result *= np.exp(-gamma_C*cdist(C, C, 'sqeuclidean'))
    return result
```

The kernel function is just as $k(x, x') = e^{-\gamma_S \|S(x)-S(x')\|^2} * e^{-\gamma_C \|C(x)-C(x')\|^2}$, and use scipy.spatial.distance.cdist to calculate the norm. $\gamma_S = \gamma_C = 0.001$ defined in main function.

```
def initial(Gram, k, mode):
    mean = np.zeros((k, Gram.shape[1]), dtype=Gram.dtype) # mark
    if mode == 0: # normal k-means -> random center
        center =  np.array(random.sample(range(0, 10000), k))
        mean = Gram[center,:]
    elif mode == 1: # k-means++
        mean[0] = Gram[np.random.randint(Gram.shape[0], size=1), :]
        for cluste_id in range(1, k):
            temp_dist = np.zeros((len(Gram), cluste_id))
            for i in range(len(Gram)):
                for j in range(cluste_id):
                    temp_dist[i][j] = np.linalg.norm(Gram[i]-mean[j])
            dist = np.min(temp_dist, axis=1)
            sum = np.sum(dist) * np.random.rand()
            for i in range(len(Gram)):
                sum -= dist[i]
                if sum <= 0:
                    mean[cluste_id] = Gram[i]
                    break
    return mean
```

This function generates k means as the initial points depending on the input mode (0 for normal k-means which return random k means) (1 for k-means++ which try to let each mean ‘far’ enough)

```python
def kmeans(Gram, k, mode):
    history = []

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

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

This is the main function for doing k-means. List history stores the clusters in each iteration. Each iteration of k-means can divide into E-step and M-step. In E-step, classify all data points with the nearest data center which is mean. In M-step, according to the result in E-step, update the new data center. Do k-means until the means are coverage.

```python
def visualize(history, image_size, filename, k, mode):
    gif = []
    color = [ImageColor.getrgb('Red'), ImageColor.getrgb('Green'), ImageColor.getrgb('Blue'), ImageColor.getrgb('Yellow')]

    iteration = len(history)
    for i in range(iteration):
        gif.append(Image.new("RGB", image_size))
        for y in range(image_size[0]):
            for x in range(image_size[1]):
                gif[i].putpixel((x, y), color[history[i][y*image_size[0]+x]])
    try:
        os.mkdir("./k_means_final")
        os.mkdir("./k_means_gif")
    except OSError:
        print("dir already exist")
    gif[0].save("./k_means_gif/" + filename + f"_modek{mode}_{k}.gif",
                format='GIF',
                save_all=True,
                append_images=gif[1:],
                duration=400, loop=0)
    gif[-1].save("./k_means_final/" + filename + f"_modek{mode}_{k}.jpg", format='JPEG')
```

Function visualize take history list as input to draw the plot and use PIL to generate image and save gif result.

<u>Spectral clustering:</u>

```python
from K_means import load, kernel, kmeans # reuse function in k-means
```

Reuse these function since they are the same as k-means part.

```python
if __name__ == "__main__":
    # global parameter
    gamma_C = 1e-3
    gamma_S = 1e-3

    filename = input("Filename: ")
    k = int(input("number of clusters: "))
    # normalized -> normalized cut
    # unnormalized -> ratio cut
    mode_s = int(input("0(normalized) or 1(unnormalized): "))
    mode_k = int(input("0(k-means) or 1(k-means++): "))

    print("loading...")
    dataC, dataS, image_size = load(filename)
    print("Calculate Gram Matrix...")
    Gram = kernel(gamma_S, gamma_C, dataS, dataC)
    print("Calculate L, eigenValue and eigenVector...")
    L = Laplacian(mode_s, Gram, filename)
    U = cal_eigen(mode_s, L, k, filename)

    print("do k-means...")
    history = kmeans(U, k, mode_k)
    # visualize
    visualize(history, image_size, filename, k, mode_s, mode_k)
    if k == 2:
        drawplot2D(U, history[-1])
    elif k == 3:
        drawplot3D(U, history[-1])
```

The main procedure of this part is 1. Generate Gram Matrix by kernel function as k-means part and then 2. Generate Graph Laplacian L depending which cut we use. 3. Calculate eigenValue and eigenVector and get U matric. 4. Do visualize.

```python
def Laplacian(mode_s, Gram, filename):
    if os.path.exists(f"Laplacian_modeS{mode_s}_"+filename+".npy"):
        L = np.load(f"Laplacian_modeS{mode_s}_"+filename+".npy")
    else:
        W = Gram
        D = np.diag(np.sum(W, axis=1))
        L = D - W # ratio cut
        if mode_s == 0: # normalized cut
            # L_sym = D^(-1/2)*L*D^(-1/2)
            D_sqrt_inv = np.diag(1/np.diag(np.sqrt(D)))
            L = D_sqrt_inv @ L @ D_sqrt_inv
        np.save(f"Laplacian_modeS{mode_s}_"+filename+".npy", L)
    return L
```

In Laplacian and cal_eigen function, I use np.save to save time when rerun this part. Function Laplacian calculate the L matric. If it's ratio cut mode (unnormalized Spectral clustering) $\mathrm{L} = \mathrm{D} - \mathrm{W}$. If it's normalized cut mode (normalized Spectral clustering) $\mathrm{L}_{\mathrm{sym}} = \mathrm{D}^{-\frac{1}{2}} * L * \mathrm{D}^{-\frac{1}{2}}$.

```python
def cal_eigen(mode_s, L, k, filename):
    if (os.path.exists(f"eigenValue_modeS{mode_s}_"+filename+".npy") and
        os.path.exists(f"eigenVector_modeS{mode_s}_"+filename+".npy")):
        eigenValue = np.load(f"eigenValue_modeS{mode_s}_"+filename+".npy")
        eigenVector = np.load(f"eigenVector_modeS{mode_s}_"+filename+".npy")
    else:
        eigenValue, eigenVector = np.linalg.eig(L)
        np.save(f"eigenValue_modeS{mode_s}_"+filename+".npy", eigenValue)
        np.save(f"eigenVector_modeS{mode_s}_"+filename+".npy", eigenVector)

    index = eigenValue.argsort()
    # select the smallest eigenValue except 0
    U = eigenVector[:, index[1: k+1]] # unnormalized
    if mode_s == 0: # normalized
        # T_ij = u_ij / (\sigma_k(u_ik**2))^(1/2)
        U /= np.sqrt(np.sum(np.power(U, 2), axis=1)).reshape(-1,1)
    return U
```

Function cal_eigen use np.linalg.eig to calculate eigenValue and eigenVector. Then collect kth-min eigenVectors except 0 to generate U. If it's ratio cut, $t_{ij} = \frac{u_{ij}}{(\sum_k u_{ik}^2)^{\frac{1}{2}}}$ and return T as U.

Visualize part is similar to visualize in k-means part.

b. Results & Observations

k-means:

image1:

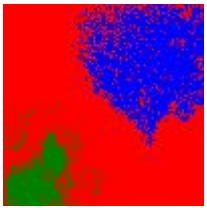
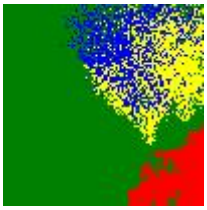
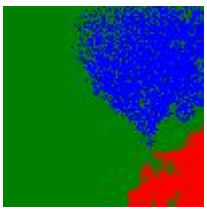
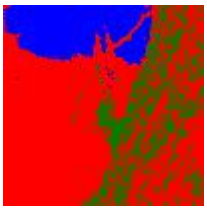
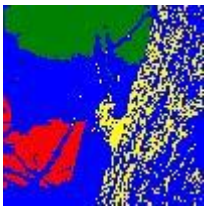
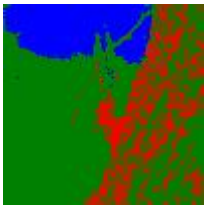
	K=2	K=3	K=4
k-means			
k-means++			

image2: 

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

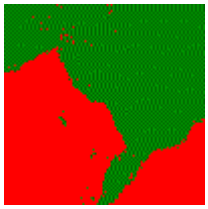
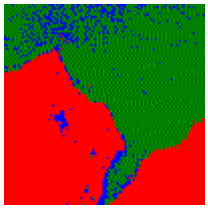
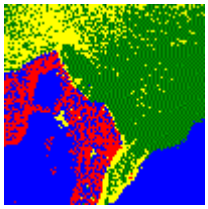
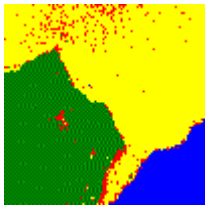
<u>Observation:</u>

Some time k-means will get same result as k-means++, but k-means++ is higher stability since k-means start with random means. For k=4 in image1, k-means suffered on noise at top of the image. For k=3 in image1, k-means and k-means++ get different result. I think both them are reasonable, k-means for land and sea, and k-means++ for dark and light sea.

The interesting thing is, in image1, for human eyes, the clusters will be land and sea if k=2. However, after trying deferent initial means, the difference between dark and light sea seems bigger than the difference between land and sea. It may because of the kernel function we use, it multiplying two PBF kernels with color and spatial data respectively. It makes k-means not only consider the color information, but also where the color is.

<u>Spectral clustering:</u>

Image1:

k-means	K=2	K=3	K=4
ratio cut			
normalized cut			

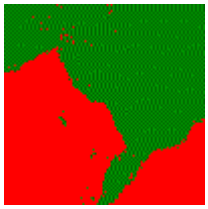
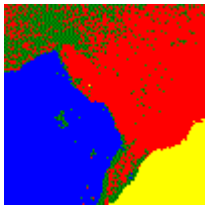
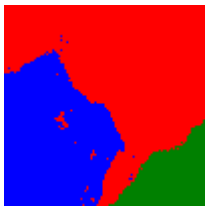
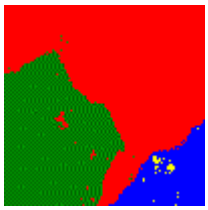
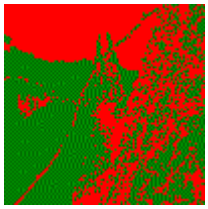
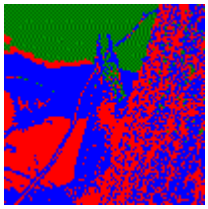
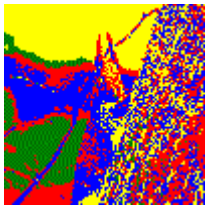
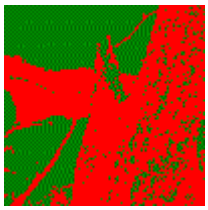
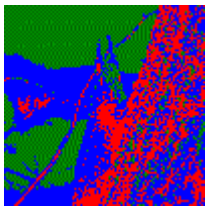
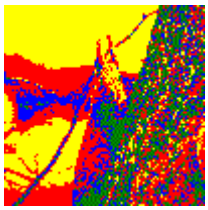
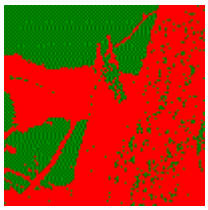
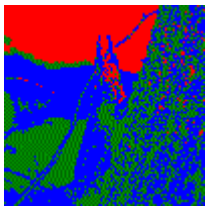
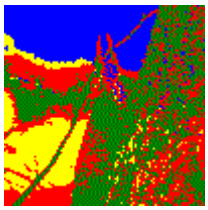
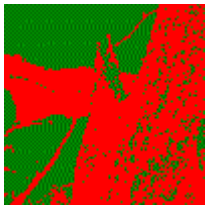
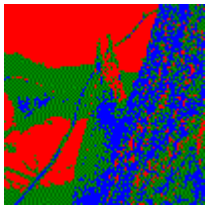
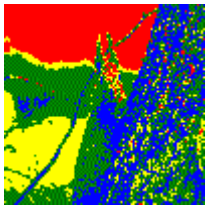
k-means++	K=2	K=3	K=4
ratio cut			
normalized cut			

image2: 

k-means	K=2	K=3	K=4
ratio cut			
normalized cut			

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

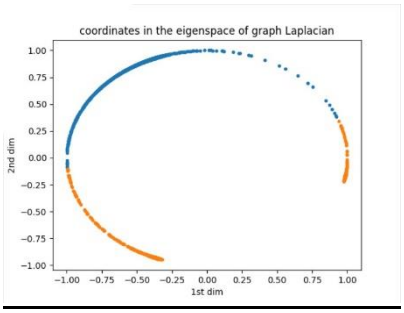
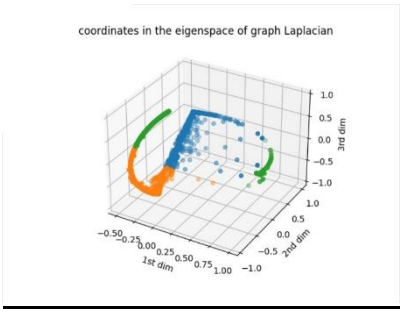
Observation:

- If the image is simple which has less boundary, there's less different between ratio cut and normalized cut.
- The more num of clusters, the more different between ratio cut and normalized cut. We can see this from both image1 and image2.
- Same as we find in k-means part, k-means is more likely to get bad result since it start with random means.

Part4: coordinates in eigenspace

The plots below show the coordinates in eigenspace for different cut in k=2 or k=3 (using k-means++).

image1:

	K=2	**K=3**
ratio cut		

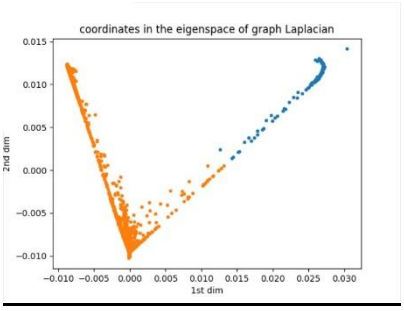
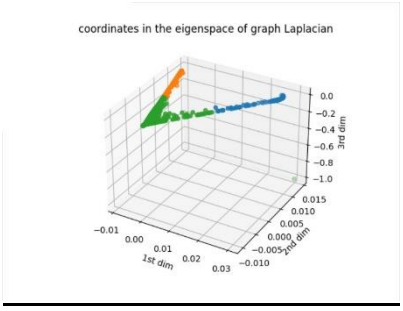
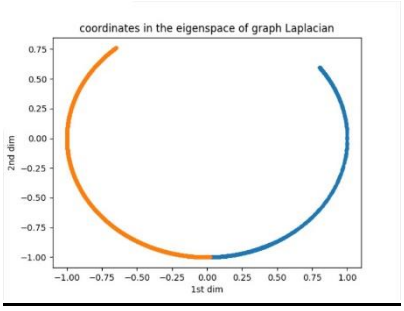
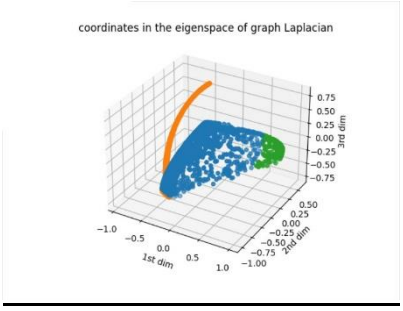
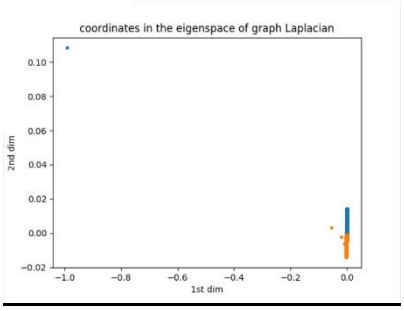
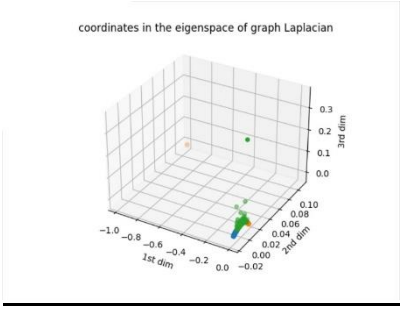
normalized cut		

image2:

	K=2	**K=3**
ratio cut		
normalized cut		

Observation:

- Not all dimensions are so useful for classifying the data like image1, normalized cut, k=3. The 3rd dimension is almost useless. And image2, normalized cut, k=2, the 1st dimension is useless.
- The data with same cluster is close in eigenspace. However, their coordinates are not the same.
- The plots for ratio cut and normalized cut are quite different.
- I also try plots for k-means. The plots generate form k-means are all similar to k-means++, since the difference between them are initial means. It won't affect the coordinates of data in eigenspace, it only affects the cluster results.