Machine Learning HW6

311551094 資科工碩一 廖昱瑋

I. Code

a. Kernel k-means

1. Main function

First, load the image. Then, computer kernel of the image and initialize clusters. After that, do kernel k-means in loop until converge.

```
img = np.asarray(Image.open(IMAGE_PATH).getdata()) #load image to 10000*3 numpy array

img_kernel = kernel(img)
clusters, C = init(img_kernel)

while not converge:
    print(f'iteration: {iteration}')
    pre_clusters = clusters

    clusters, C = kernel_kmeans(img_kernel, clusters, C)

    save_picture(clusters, iteration)
    converge = check_converge(clusters, pre_clusters)

    iteration += 1
```

2. Kernel function

spatial_distance is $\|S(x) - S(x)\|^2$, and color_distance is $\|C(x) - C(x)\|^2$.

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

```
def kernel(img):
    color_dist = cdist(img, img, 'sqeuclidean')
    coordinates = np.zeros((IMAGE_SIZE, 2))
    for row in range(100):
        row_index = row * 100
        for col in range(100):
            coordinates[row_index + col][0] = row
            coordinates[row_index + col][1] = col
    spatial_distance = cdist(coordinates, coordinates, 'sqeuclidean')
    img_kernel = np.multiply(np.exp(-GAMMA_C * color_dist), np.exp(-GAMMA_S * spatial_distance))

    return img_kernel
```

3. Initial clustering

Provide 2 modes which are random and k-means++. The number of initial clusters(k) are defined by users.

In random mode, randomly choose k data as centers. In k-means++, randomly choose 1 data as first center. Then, find farthest data from selected centers in the loop until we find k centers in total. After finding k centers, *init_cluster_c* and *construct_c* help to do clustering for the initial centers.

```python
def init(img_kernel):
    if MODE == 'random':
        centers = np.random.randint(IMAGE_SIZE, size=NUM_CLUSTER)
        clusters, C = init_cluster_c(img_kernel, centers)
    elif MODE == 'k-means++':
        centers = np.zeros(NUM_CLUSTER, dtype=int)
        centers[0] = np.random.randint(IMAGE_SIZE, size=1)

        for i in range(1, NUM_CLUSTER):
            distances = np.zeros(IMAGE_SIZE)
            for j in range(IMAGE_SIZE):
                min_dist = np.Inf
                for k in range(i):
                    temp_dist = img_kernel[centers[k]][j]
                    if temp_dist < min_dist:
                        min_dist = temp_dist
                distances[j] = min_dist
            distances = distances / np.sum(distances)
            centers[i] = np.random.choice(10000, size=1, p=distances)
        clusters, C = init_cluster_c(img_kernel, centers)
    else:
        print('Wrong input for cluster initialization!')

    return clusters, C
```

```python
def init_cluster_c(img_kernel, centers):
    clusters = np.zeros(IMAGE_SIZE, dtype=int)
    for pixel in range(IMAGE_SIZE):
        if pixel in centers:
            continue
        min_dist = np.Inf
        for cluster in range(NUM_CLUSTER):
            center = centers[cluster]
            temp_dist = img_kernel[pixel][center]
            if temp_dist < min_dist:
                min_dist = temp_dist
                clusters[pixel] = cluster

    C = construct_C(clusters)

    save_picture(clusters, 0)

    return clusters, C
```

```python
def construct_C(clusters):
    C = np.zeros(NUM_CLUSTER, dtype=int)
    for i in range(IMAGE_SIZE):
        C[clusters[i]] += 1

    return C
```

4. Do clustering by kernel k-means.

   sigma_pq is the second term, and *sigma_n* is the third term of the following equation.

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

```python
def kernel_kmeans(img_kernel, clusters, C):
    new_clusters = np.zeros(IMAGE_SIZE, dtype=int)
    pq = sigma_pq(img_kernel, clusters, C)
    for pixel in trange(IMAGE_SIZE):
        distances = np.zeros(NUM_CLUSTER)
        for cluster in range(NUM_CLUSTER):
            distances[cluster] = img_kernel[pixel][pixel]
            distances[cluster] -= sigma_n(img_kernel[pixel, :], clusters, cluster, C)
            distances[cluster] += pq[cluster]
        new_clusters[pixel] = np.argmin(distances)
    new_C = construct_C(new_clusters)

    return new_clusters, new_C
```

```python
def sigma_pq(img_kernel, clusters, C):
    sum = np.zeros(NUM_CLUSTER)
    for k in range(NUM_CLUSTER):
        ker = img_kernel.copy()
        for pixel in range(IMAGE_SIZE):
            if clusters[pixel] != k:
                ker[pixel, :] = 0
                ker[:, pixel] = 0
        sum[k] = np.sum(ker)/ (C[k] **2)

    return sum
```

```python
def sigma_n(pixel_kernel, clusters, k, C):
    sum = 0
    for n in range(IMAGE_SIZE):
        if clusters[n] == k:
            sum += pixel_kernel[n]

    return 2 / C[k] * sum
```

5. Check convergence

   Define converge as no data changes its cluster in this run.

```python
def check_converge(clusters, pre_clusters):
    for pixel in range(IMAGE_SIZE):
        if clusters[pixel] != pre_clusters[pixel]:
            return 0

    return 1
```

6. Save the result as image in each run.

```
def save_picture(clusters, iteration):
    pixel = np.zeros((10000, 3))
    for i in range(IMAGE_SIZE):
        pixel[i, :] = COLOR[clusters[i], :]
    pixel = np.reshape(pixel, (100, 100, 3))
    img = Image.fromarray(np.uint8(pixel))
    img.save(OUTPUT_DIR + '\%01d_%03d.png'%(NUM_CLUSTER, iteration), 'png')

    return
```

b. Spectral clustering – ratio cut

1. Main function

   First compute the similarity matrix W by *kernel*. Then, compute the unnormalized Laplacian L. After that, do eigen decomposition of L and construct U by the number of clusters defined by users. Use k-means algorithm to cluster rows of U. In the last, draw coordinates in the eigenspace if number of clusters are equal or less than 3.

```
W = kernel(img)
L, D = compute_laplacian(W)
U = eigen_decomposition(L)
clusters = kmeans(U)

if NUM_CLUSTER <= 3:
    draw_eigenspace(U, clusters)
```

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

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

2. Compute the unnormalized Laplacian, and do eigen decomposition

```
def compute_laplacian(W):
    L = np.zeros((IMAGE_SIZE, IMAGE_SIZE))
    D = np.zeros((IMAGE_SIZE, IMAGE_SIZE))
    for i in range(IMAGE_SIZE):
        D[i][i] = np.sum(W[i, :])
    L = D - W

    return L, D
```

```
def eigen_decomposition(L):
    eigenvalues, eigenvectors = np.linalg.eig(L)
    index = np.argsort(eigenvalues)
    eigenvectors = eigenvectors[:, index]

    return eigenvectors[:, 1:1+NUM_CLUSTER].real
```

3. k-means algorithm

   Initialize the clustering by *init_kmeans*. In E-step, assign cluster for each data. In M-step, compute mean for each cluster. Do k-means algorithm in loop until converge.

```
def kmeans(U):
    converge = 0
    iteration = 1
    means, clusters = init_kmeans(U)

    while not converge:
        print(f'iteration: {iteration}')
        pre_clusters = clusters

        clusters = E_step(U, means)
        means = M_step(U, clusters)

        save_picture(clusters, iteration)
        converge = check_converge(clusters, pre_clusters)

        iteration += 1

    return clusters
```

4. Initialize the clusters

   Provide 2 modes which are random and k-means++. The number of initial clusters(k) are defined by users.

```python
def init_kmeans(U):
    if INIT_METHOD == 'random':
        centers = np.random.randint(IMAGE_SIZE, size=NUM_CLUSTER)
        means, clusters = init_means_clusters(U, centers)
    elif INIT_METHOD == 'k-means++':
        centers = np.zeros(NUM_CLUSTER, dtype=int)
        centers[0] = np.random.randint(IMAGE_SIZE, size=1)

        for i in range(1, NUM_CLUSTER):
            distances = np.zeros(IMAGE_SIZE)
            for j in range(IMAGE_SIZE):
                min_dist = np.Inf
                for k in range(i):
                    temp_dist = square_distance(U[centers[k]], U[j])
                    if temp_dist < min_dist:
                        min_dist = temp_dist
                distances[j] = min_dist
            distances = distances / np.sum(distances)
            centers[i] = np.random.choice(10000, size=1, p=distances)
        means, clusters = init_means_clusters(U, centers)
    else:
        print('Wrong input for cluster initialization!')

    return means, clusters
```

```python
def init_means_clusters(U, centers):
    means = np.zeros((NUM_CLUSTER, NUM_CLUSTER))
    clusters = np.full(IMAGE_SIZE, -1, dtype=int)

    for i in range(NUM_CLUSTER):
        means[i] = U[centers[i]]
        clusters[centers[i]] = i

    return means, clusters
```

5. E-step

$$r_{nk} = \begin{cases} 1 & \text{if } k = \underset{k}{\operatorname{argmin}} \|x_n - \mu_k\| \\ 0 & \text{otherwise} \end{cases}$$

E-step

```python
def E_step(U, means):
    new_clusters = np.zeros(IMAGE_SIZE, dtype=int)
    for pixel in range(IMAGE_SIZE):
        min_dist = np.Inf
        for cluster in range(NUM_CLUSTER):
            temp_dist = square_distance(means[cluster], U[pixel])
            if temp_dist < min_dist:
                min_dist = temp_dist
                new_clusters[pixel] = cluster

    return new_clusters
```

6. M-step

$$\frac{\partial J}{\partial \mu_k} = 0 \Rightarrow 2\sum_{n=1}^{N} r_{nk}(x_n - \mu_k) = 0 \Rightarrow \mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}}$$

M-step

```python
def M_step(U, clusters):
    new_means = np.zeros((NUM_CLUSTER, NUM_CLUSTER))
    for cluster in range(NUM_CLUSTER):
        count = 0
        for pixel in range(IMAGE_SIZE):
            if clusters[pixel] == cluster:
                new_means[cluster] += U[pixel]
                count += 1
        new_means[cluster] /= count

    return new_means
```

7. Draw the coordinates in the eigenspace

```python
def draw_eigenspace(U, clusters):
    points_x, points_y, points_z = [], [], []

    if NUM_CLUSTER == 2:
        for _ in range(NUM_CLUSTER):
            points_x.append([])
            points_y.append([])
        for pixel in range(IMAGE_SIZE):
            points_x[clusters[pixel]].append(U[pixel][0])
            points_y[clusters[pixel]].append(U[pixel][1])

        for cluster in range(NUM_CLUSTER):
            plt.scatter(points_x[cluster], points_y[cluster], c=EIGENSPACE_COLOR[cluster])
        plt.savefig(f'{OUTPUT_DIR}\eigenspace_{NUM_CLUSTER}.png')
    elif NUM_CLUSTER == 3:
        fig = plt.figure()
        ax = fig.add_subplot(projection='3d')
        for _ in range(NUM_CLUSTER):
            points_x.append([])
            points_y.append([])
            points_z.append([])
        for pixel in range(IMAGE_SIZE):
            points_x[clusters[pixel]].append(U[pixel][0])
            points_y[clusters[pixel]].append(U[pixel][1])
            points_z[clusters[pixel]].append(U[pixel][2])

        for cluster in range(NUM_CLUSTER):
            ax.scatter(points_x[cluster], points_y[cluster], points_z[cluster], c=EIGENSPACE_COLOR[cluster])
        fig.savefig(f'{OUTPUT_DIR}\eigenspace_{NUM_CLUSTER}.png')
    plt.show()
```

c. Spectral clustering – normalized cut

1. Main function

First compute the similarity matrix W by *kernel*. Then, compute the Laplacian matrix L. Normalized L by *normalize_laplacian*. After that, do eigen decomposition of L_normal and construct U by the number of clusters defined by users. Re-substitute T from U by normalizing the rows. Use k-means algorithm to cluster rows of T. In the last, draw coordinates in the eigenspace if number of clusters are equal or less than 3.

```
W = kernel(img)
L, D = compute_laplacian(W)
L_normal, sqrt_D = normalize_laplacian(L, D)
U = eigen_decomposition(L_normal)
T = sqrt_D @ U
clusters = kmeans(T)

if NUM_CLUSTER <= 3:
    draw_eigenspace(T, clusters)
```

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

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

2. Normalize the Laplacian matrix $L_{normal} = D^{-1/2} * L * D^{-1/2}$

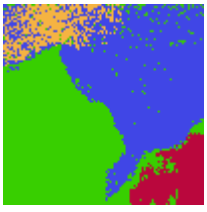
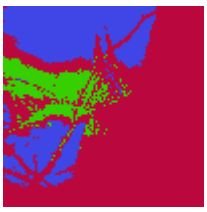
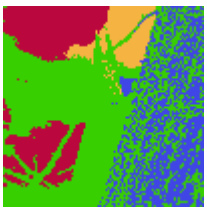
```
def normalize_laplacian(L, D):
    sqrt_D = np.zeros((IMAGE_SIZE, IMAGE_SIZE))
    for i in range(IMAGE_SIZE):
        sqrt_D[i][i] = D[i][i] ** (-0.5)
    L_n = sqrt_D @ L @ sqrt_D

    return L_n, sqrt_D
```

II. Result

a. Kernel k-means ($\gamma_s = 0.001$ 、 $\gamma_c = 0.001$)

1. random

	2 clusters	3 clusters	4 clusters
Image 1			
Image 2			

2. k-means++

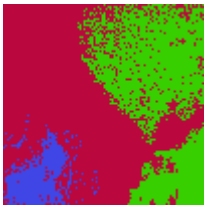
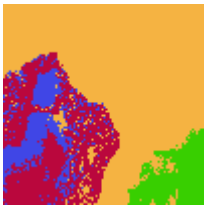
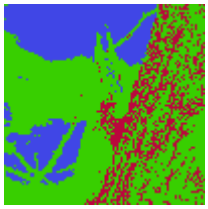
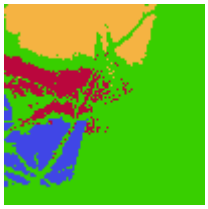
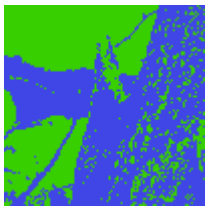
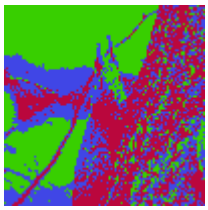
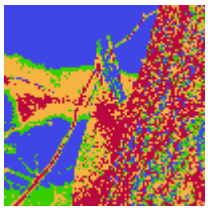
	2 clusters	3 clusters	4 clusters
Image 1			

Image 2			

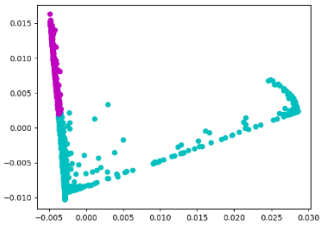
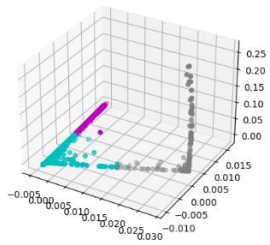
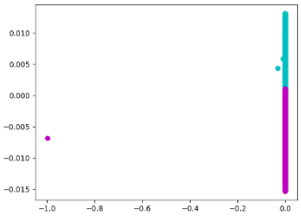
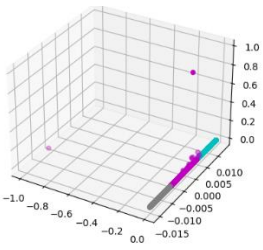
In the previous results, surprisingly, we can see that k-means++ does not have better result than random and both initial states turn out not too bad results. The reason may be the bad hyperparameters. However, k-mean++ for 4 clusters in image 1 has the better result than random.

b. Spectral clustering – ratio cut ($\gamma_s = 0.0001$ 、 $\gamma_c = 0.001$)

1. random

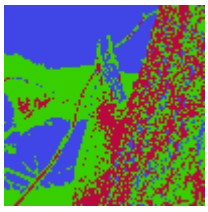
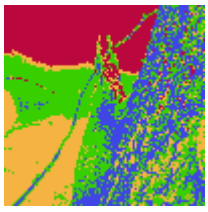
	2 clusters	3 clusters	4 clusters
Image 1			
Image 2			

The results are quite good. For image 1, 4 clusters is fairly equal to 3 clusters. This is reasonable because of the origin pattern of the image. However, for image 2, 4 clusters is needed to imply more information because the complexity of the image.

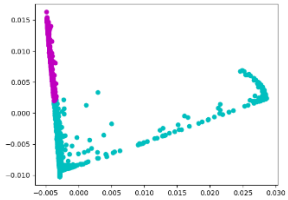
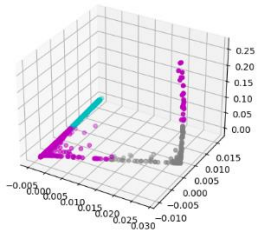
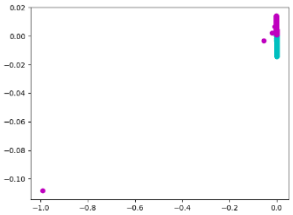
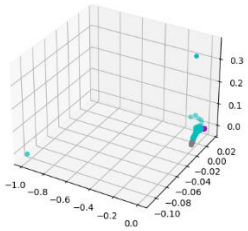
	2 clusters	3 clusters
Image 1		
Image 2		

In eigenspace, we can see that it separates quite well which confirms the above clustering results. The summation in each dimension of eigenvector is 0.

2. k-means++

	2 clusters	3 clusters	4 clusters
Image 1			
Image 2			

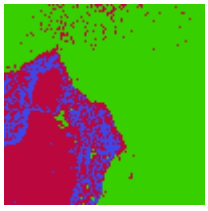
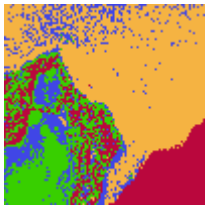
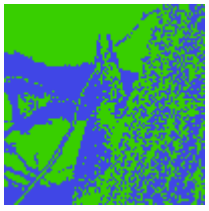
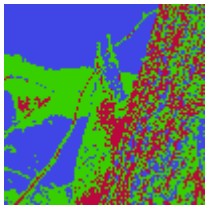
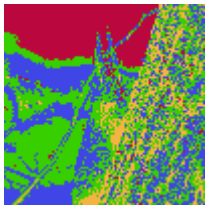
The k-means++ results look likely to random results.

	2 clusters	3 clusters
Image 1		
Image 2		

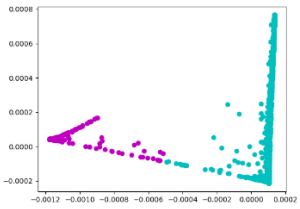
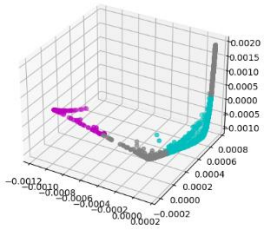
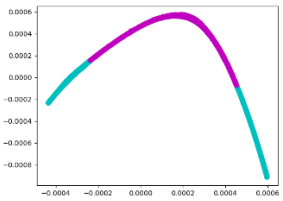
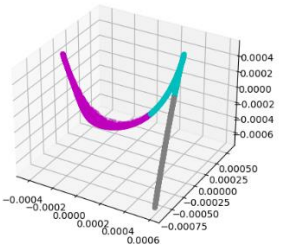
In eigenspace, we can see that it separates quite well which confirms the above clustering results.

c. Spectral clustering – normalized cut ($\gamma_s = 0.0001$ 、 $\gamma_c = 0.001$)

1. random

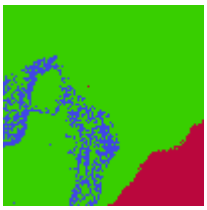
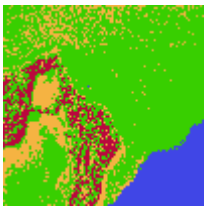
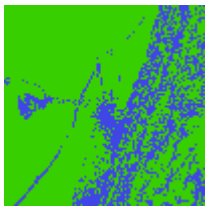
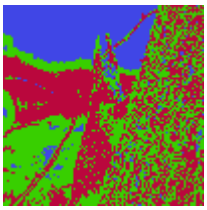
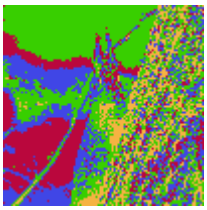
	2 clusters	3 clusters	4 clusters
Image 1			
Image 2			

The result is even better than ratio cut. For image 1, 4 clusters can show more information than 3 clusters. It provides mountains in island.

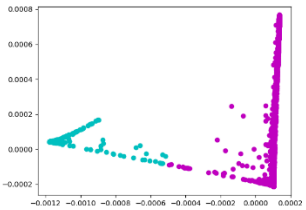
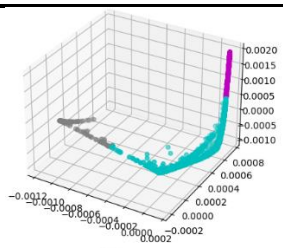
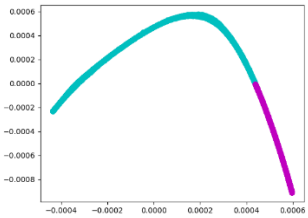
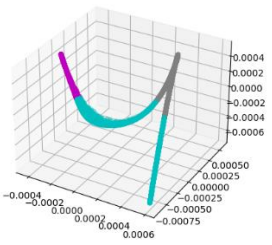
	2 clusters	3 clusters
Image 1		
Image 2		

For image 1, we can see that it has discontinuous clustering in grey part. This also shows in the above image results. Some blue dots are in the island, and some red dots are in the ocean.

2. k-means++

	2 clusters	3 clusters	4 clusters
Image 1			
Image 2			

The clustering results are quite good instead of 2 clusters. It's because the number of clusters is too small.

	2 clusters	3 clusters
Image 1		
Image 2		

Unlike random initialization, for image 1, the cluster result in eigenspace is great. The above image results also show very good clustering.

III. Discussion

a. Spectral clustering has better result than kernel k-means. It's because we don't update the means of clusters in kernel k-means. It relies more on the initial state but no matter in which initial method, it contains random properties.

b. It is likely to have different clustering results every time we run the program, even if all the hyperparameters are the same. It's because of different initial states.

c. The hyperparameters are very important. If we set bad hyperparameters, the result may be nonsense. If γ in RBF kernel is large, the decision boundary will be tight. If γ in RBF kernel is small, the decision boundary will be loose.

d. The hyperparameters are even important than initial method. If we set bad hyperparameters, k-means++ still has bad result.

e. K-mean++ has better time efficiency than random because k-means++ mostly converges in fewer runs.

f. It's hard to set number of clusters because it doesn't have correct answer to choose k in unsupervised learning. Therefore, we need to observe the results to determine good k. For image 1, it's good to choose k as 3. For image 2, it's good to choose k as 4.