Machine Learning HW6

309551064 張凱翔

1. Code With Detailed Explanations

Part 1.

K-means:

The figure below is my main function.

```
if __name__ == '__main__':
    spatial, color = read_data('image1.png')
    K_means('image1.png', spatial, color)
```

And, I will change the filename for another image.

```
K = 2
```

Furthermore, I declare a global variable “K” for the number of cluster and will change it manually for different parts.

First, set global variable ”K=2” for two cluster and read one of the images.

```
def read_data(file_name):
    img = Image.open(file_name)
    width, height = img.size
    pixel = np.array(img.getdata()).reshape((width*height, 3))

    position = []
    for i in range(100):
        for j in range(100):
            position.append([i, j])
    position = np.array(position)
    return position, pixel
```

I use “Image” in “PIL” for reading image into RGB mode and resize it from(10000, 3) to (100, 100, 3) for calculating the kernel. Then, I create a list to store the position information of each pixel and transform it into “numpy” array.

```
def K_means(file_name, spatial, color):
    gram_matrix = kernel(spatial, color)
    initial_methods = ['random_partition']
    for method in initial_methods:
        classification = initial(method)
        iteration = 0
        old_diff = 0
        visualization(classification, iteration, file_name, method)
        while iteration < 20:
            iteration += 1
            old_classification = classification
            classification = classify(gram_matrix, classification)
            diff = difference(classification, old_classification)
            visualization(classification, iteration, file_name, method)
            if diff == old_diff:
                break
            old_diff = diff
```

After reading image, I call the function “K_means” to cluster the pixel on image.

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

```
def kernel(spatial, color):
    gamma_c = 1/(255*255)
    gamma_s = 1/(100*100)
    spatial_sq_dists = squareform(pdist(spatial, 'sqeuclidean'))
    spatial_rbf = np.exp(-gamma_s * spatial_sq_dists)
    color_sq_dists = squareform(pdist(color, 'sqeuclidean'))
    color_rbf = np.exp(-gamma_c * color_sq_dists)
    kernel = spatial_rbf * color_rbf

    return kernel
```

Before going straight into the algorithm of “K means”, I call the function “kernel” to calculate the kernel as the spec asked. I set the gamma_c and gamma_s as the figure shown and calculate the kernel based on the algorithm given in spec.

I declare a list with one initial method “random partition” (and there is another in part3) and use a for loop to run different initial method.

```
def initial(method):
    if method == 'random_partition':
        classification = np.random.randint(0, K, size=10000)
    if method == 'Forgy':
        classification = np.zeros(10000, dtype=np.int)
        center = np.random.randint(0, 10000, size=K)
        for i in range(100):
            for j in range(100):
                near = 1e9
                for k in range(len(center)):
                    if (((i - (center[k] / 100)) ** 2) + (j - (center[k] % 100)) ** 2) < near:
                        classification[100 * i + j] = k
                        near = (((i - (center[k] / 100)) ** 2) + (j - (center[k] % 100)) ** 2)
    return classification
```

The initial method I using in Part1 is “random partition” and it is just randomly assign a cluster to each pixel.

```
def visualization(classification, iteration, file_name, method):
    img = Image.open(file_name)
    pixel = img.load()
    color = [(0,0,0), (125, 0, 0), (0, 255, 0), (255, 255, 255)]
    for i in range(100):
        for j in range(100):
            pixel[j, i] = color[classification[i * 100 + j]]
    img.save(file_name.split('.')[0] + '_' + str(K) + '_' + str(method) + '_' + str(iteration) + '.png')
```

After initialization, I call the function “visualization” to visual the status of 0 iteration. I load the image and assign a new color to each pixel according to its cluster (there are four cluster in part3 and here only use two colors) and save the new image with recognizable name.

Loop:

Stepping into the iteration step, the variable “iteration” add 1, the variable “old_classification” store the classification of last iteration and call the function “classify” to get the new classification.

```
def classify(gram_matrix, classification):
    new_classification = np.zeros(10000, dtype=np.int)
    third_term = np.zeros(K, dtype=np.int)
    cluster_num = np.zeros(K, dtype=np.int)
    for i in range(len(classification)):
        cluster_num[classification[i]] += 1
    for i in range(10000):
        for j in range(10000):
            if classification[i] == classification[j]:
                third_term[classification[i]] += gram_matrix[i][j]
    for i in range(len(third_term)):
        third_term[i] /= cluster_num[i] ** 2

    for i in range(10000):
        distance = np.zeros(K, dtype=np.float32)
        for j in range(K):
            second_term = 0
            count = 0
            for k in range(10000):
                if classification[k] == j:
                    second_term += gram_matrix[i][k]
                    count += 1
            second_term = second_term * 2 / count
            distance[j] = gram_matrix[i][i] - second_term + third_term[j]
        new_classification[i] = np.argmin(distance)

    return new_classification
```

The new classification is calculated based on the formula in pdf shown below.

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

First calculating the third term for each cluster in the formula, which is sum up the element of gram matrix where both “q” and “p” belong to the same cluster, and divided by the number of pixel in the cluster.

The second term is sum up the row of the pixel in gram matrix where “n” belongs to the cluster is now calculating. Then, for each pixel, calculating the distance for the pixel to each cluster and select the cluster of min distance and assign that cluster to the pixel.

```
def difference(classification, old_classification):
    diff = 0
    for i in range(len(classification)):
        diff += abs(classification[i] - old_classification[i])
    return diff
```

After having the new classification, I call the function “difference” to calculate the difference between the new classification and the last classification.

Visualize the classification of this iteration.

And if the difference is same as last time or iteration is more than 20, the iteration ends.

If not, then the variable “old_diff” store the difference of this iteration and loop again.

Spectral clustering:

The figure below is my main function.

```
if __name__ == '__main__':
    sspatial, color = read_data('image1.png')
    data = normalized_cut(spatial, color)
    spectral_clustering('image1.png', data)
```

I will change the filename for another image and change the function “normalized_cut” to “ratio_cut” for another method, respectively

```
K = 2
```

Furthermore, I declare a global variable “K” for the number of cluster and will change it manually for different parts.

First, set global variable ”K=2” for two cluster and read one of the images.

```
def read_data(file_name):
    img = Image.open(file_name)
    width, height = img.size
    pixel = np.array(img.getdata()).reshape((width*height, 3))

    position = []
    for i in range(100):
        for j in range(100):
            position.append([i, j])
    position = np.array(position)
    return position, pixel
```

I use “Image” in “PIL” for reading image into RGB mode and resize it from(10000, 3) to (100, 100, 3) for calculating the kernel. Then, I create a list to store the position information of each pixel and transform it into “numpy” array.

After reading image, I call the function “ratio_cut” or “normalized_cut” to get Laplacian and eigenvectors.

```
def ratio_cut(spatial, color):
    W = kernel(spatial, color)
    D = np.diag(np.sum(W, axis=1))
    L = D - W

    eigen_values, eigen_vectors = np.linalg.eig(L)
    idx = np.argsort(eigen_values)[1: K+1]
    U = eigen_vectors[:, idx].real.astype(np.float32)

    return U
```

```
def normalized_cut(spatial, color):
    W = kernel(spatial, color)
    D = np.diag(np.sum(W, axis=1))
    D_inverse_sqrt = np.diag(np.power(np.diag(D), -0.5))
    L_sym = np.identity(len(spatial)) - D_inverse_sqrt @ W @ D_inverse_sqrt

    eigen_values, eigen_vectors = np.linalg.eig(L_sym)
    idx = np.argsort(eigen_values)[1: K+1]
    U = eigen_vectors[:, idx].real.astype(np.float32)

    T = np.zeros((U.shape[0], U.shape[1]))
    for i in range(T.shape[0]):
        for j in range(T.shape[1]):
            sum_tmp = 0
            for k in range(T.shape[1]):
                sum_tmp += U[i][k] ** 2
            T[i][j] = U[i][j] / (sum_tmp ** 0.5)

    return T
```

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} $D^{-1/2} LD^{-1/2}$
- **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 $A_i = \{j | \, y_j \in C_i\}$.

- Compute the unnormalized Laplacian L.
- **Compute the first k generalized eigenvectors $u_1, \ldots, u_k$ of the generalized eigenproblem $Lu = \lambda Du$**
- 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 $A_i = \{j | \, y_j \in C_i\}$.

I find the formula on the pdf as the figure show above.

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

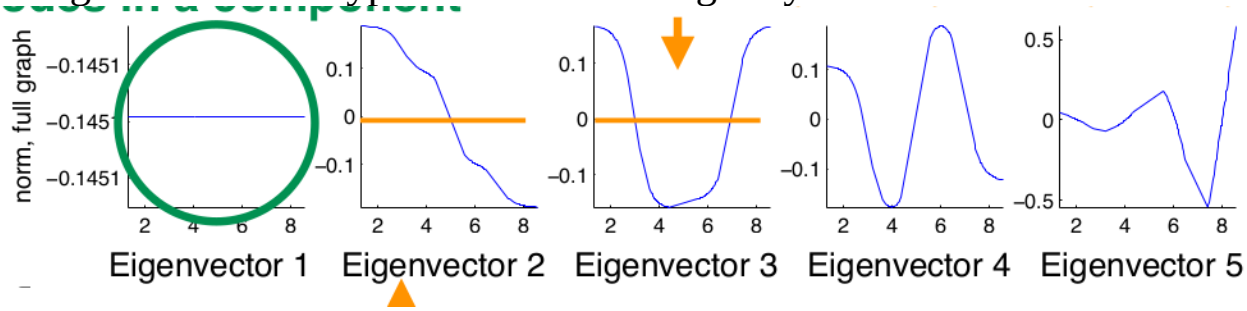
```
def kernel(spatial, color):
    gamma_c = 1/(255*255)
    gamma_s = 1/(100*100)
    spatial_sq_dists = squareform(pdist(spatial, 'sqeuclidean'))
    spatial_rbf = np.exp(-gamma_s * spatial_sq_dists)
    color_sq_dists = squareform(pdist(color, 'sqeuclidean'))
    color_rbf = np.exp(-gamma_c * color_sq_dists)
    kernel = spatial_rbf * color_rbf

    return kernel
```

This two methods both use the kernel in spec as weight, so before going straight into the algorithm of two methods, I call the function “kernel” to calculate the kernel as weight. I set the gamma_c and gamma_s as the figure shown and calculate the kernel based on the algorithm given in spec.

ratio cut:

The degree matrix is a diagonal matrix with each element on diagonal is the sum of that row and other elements are all 0. And the Laplacian matrix “L” is degree matrix “D” minus weight matrix “W” . Then call the module “np.linalg.eig” to get the eigen value and eigen vector of “L” matrix. Finally, select K eigen vectors with largest eigen value expect the first one (the reason is shown below ,

because eigenvector 1 indicates all node in a component) and change them to real type to eliminate imaginary number.



normalized cut:

The degree matrix is a diagonal matrix with each element on diagonal is the sum of that row and other elements are all 0. And, I should calculate the normalized Laplacian as the formula $L_{\text{sym}} := D^{-1/2}LD^{-1/2} = I - D^{-1/2}WD^{-1/2}$. Then, select K eigen vectors with largest eigen value expect the first one (the reason is shown above , because eigenvector 1 indicates all node in a component) and change them to real type to eliminate imaginary number. Finally, calculate the new normalized data using $t_{ij} = u_{ij}/(\sum_k u_{ik}^2)^{1/2}$.

```
def spectral_clustering(file_name, data):
    initial_methods = ['random_partition', 'kmeans++']
    for method in initial_methods:
        mu, classification = initial(method, data)
        iteration = 0
        old_diff = 1e9
        visualization(classification, iteration, file_name, method)
        while iteration < 20:
            iteration += 1
            old_classification = classification
            classification = classify(data, mu)
            diff = difference(classification, old_classification)
            visualization(classification, iteration, file_name, method)
            if diff == old_diff:
                break
            old_diff = diff
            mu = update(data, mu, classification)
        draw(classification, data)
```

After calculating the data, I call the function “spectral clustering” to cluster the pixel on image.

I declare a list with one initial method “random partition” (and there is another in part3) and use a for loop to run different initial method.

```
def initial(method, data):
    if method == 'random_partition':
        classification = np.random.randint(0, K, size=10000)
        mu = np.zeros((K, K), dtype=np.float)
        count = np.zeros(K, dtype=np.float)
        for i in range(len(classification)):
            mu[classification[i]] += data[i]
            count[classification[i]] += 1
        for i in range(K):
            mu[i] /= count[i]
    if method == 'kmeans++':
        initial_center = np.random.randint(0, 10000, size=1)
        mu = np.zeros((K, K), dtype=np.float)
        mu[0] = data[initial_center]
        for p in range(1, K):
            distance = np.zeros(10000, dtype=np.int)
            for i in range(len(data)):
                dis = np.zeros(p, dtype=np.int)
                for j in range(p):
                    tmp = 0
                    for k in range(len(data[0])):
                        tmp += (data[i][k] - mu[j][k]) ** 2
                    dis[j] = tmp
                distance[i] = min(dis)
            mu[p] = data[np.argmin(distance)]
        classification = np.zeros(10000, dtype=np.int)
        for i in range(10000):
            distance = np.zeros(K, dtype=np.float32)
            for j in range(K):
                for k in range(K):
                    distance[j] += abs(data[i][k] - mu[j][k]) ** 2
            classification[i] = np.argmin(distance)
```

The initial method I using in Part1 is “random partition” and it is just randomly assign a cluster to each pixel and calculate the central point of each cluster as variable “mu”.

```
def visualization(classification, iteration, file_name, method):
    img = Image.open(file_name)
    pixel = img.load()
    color = [(0,0,0), (125, 0, 0), (0, 255, 0), (255, 255, 255)]
    for i in range(100):
        for j in range(100):
            pixel[j, i] = color[classification[i * 100 + j]]
    img.save(file_name.split('.')[0] + '_' + str(K) + '_' + str(method) + '_' + str(iteration) + '.png')
```

After initialization, I call the function “visualization” to visual the status of 0 iteration. I load the image and assign a new color to each pixel according to its cluster (there are four cluster in part3 and here only use two colors) and save the new image with recognizable name.

Loop:

Stepping into the iteration step, the variable “iteration” add 1, the variable “old_classification” store the classification of last iteration and call the function “classify” to get the new classification.

```
def classify(data, mu):
    classification = np.zeros(10000, dtype=np.int)
    for i in range(10000):
        distance = np.zeros(K, dtype=np.float32)
        for j in range(K):
            for k in range(K):
                distance[j] += abs(data[i][k] - mu[j][k]) ** 2
        classification[i] = np.argmin(distance)
    return classification
```

The function calculates the distance of each data to each cluster and assign the cluster which distance is smallest to each data.

```
def difference(classification, old_classification):
    diff = 0
    for i in range(len(classification)):
        diff += abs(classification[i] - old_classification[i])
    return diff
```

After having the new classification, I call the function “difference” to calculate the difference between the new classification and the last classification.

Visualize the classification of this iteration.

And if the difference is same as last time or iteration is more than 20, the iteration ends.

```
def update(data, mu, classification):
    new_mu = np.zeros(mu.shape, dtype=np.float32)
    count = np.zeros(K, dtype=np.int)
    for i in range(len(classification)):
        new_mu[classification[i]] += data[i]
        count[classification[i]] += 1
    for i in range(len(new_mu)):
        if count[i] == 0:
            count[i] = 1
        new_mu[i] = new_mu[i] / count[i]
    return new_mu
```

If not, then the variable “old_diff” store the difference of this iteration, call the function “update” to update the central point of each cluster and loop again.

The function calculate the central point of each cluster as variable “new_mu”.

Part 2.

K-means:

I only change the variable “K” mentioned in Part 1. to change the number of cluster.

Spectral clustering:

I only change the variable “K” mentioned in Part 1. to change the number of cluster.

Part 3.

K-means:

```
def initial(method):
    if method == 'random_partition':
        classification = np.random.randint(0, K, size=10000)
    if method == 'Forgy':
        classification = np.zeros(10000, dtype=np.int)
        center = np.random.randint(0, 10000, size=K)
        for i in range(100):
            for j in range(100):
                near = 1e9
                for k in range(len(center)):
                    if (((i - (center[k] / 100)) ** 2) + (j - (center[k] % 100)) ** 2) < near:
                        classification[100 * i + j] = k
                        near = (((i - (center[k] / 100)) ** 2) + (j - (center[k] % 100)) ** 2)
    return classification
```

Another method I used for initialization is “Forgy”. This method is randomly select K center points and assign each pixel to the nearest cluster. And the feature I used for calculate the distance is spatial information.

Spectral clustering:

```
def initial(method, data):
    if method == 'random_partition':
        classification = np.random.randint(0, K, size=10000)
        mu = np.zeros((K, K), dtype=np.float)
        count = np.zeros(K, dtype=np.float)
        for i in range(len(classification)):
            mu[classification[i]] += data[i]
            count[classification[i]] += 1
        for i in range(K):
            mu[i] /= count[i]
    if method == 'kmeans++':
        initial_center = np.random.randint(0, 10000, size=1)
        mu = np.zeros((K, K), dtype=np.float)
        mu[0] = data[initial_center]
        for p in range(1, K):
            distance = np.zeros(10000, dtype=np.int)
            for i in range(len(data)):
                dis = np.zeros(p, dtype=np.int)
                for j in range(p):
                    tmp = 0
                    for k in range(len(data[0])):
                        tmp += (data[i][k] - mu[j][k]) ** 2
                    dis[j] = tmp
                distance[i] = min(dis)
            mu[p] = data[np.argmin(distance)]
        classification = np.zeros(10000, dtype=np.int)
        for i in range(10000):
            distance = np.zeros(K, dtype=np.float32)
            for j in range(K):
                for k in range(K):
                    distance[j] += abs(data[i][k] - mu[j][k]) ** 2
            classification[i] = np.argmin(distance)
```

Another method I used for initialization is “kmeans++”.

This method is:

1. randomly select one center points
2. calculate the distance of each data point to the nearest central points
3. select the largest distance data point as a new central point
4. repeat the procedure 2, 3 until there are K central points.

Then, calculate the distance of each data to each central point and assign the cluster which distance is smallest to each data.

Part 4.

```
def draw(classification, data):
    color = [(0,0,0), (0.5, 0, 0), (0, 1, 0), (1, 1, 1)]
    for i in range(len(data)):
        plt.scatter(data[i][0], data[i][1], s=8, c=[color[classification[i]]])
    plt.show()
```

After the iteration ends, I call the function “draw” to visualize the point in eigenspace with first two column. I give the data points in the same cluster a same color to let the point well recognized with different clusters.

2. experiments settings and results & discussion

Part 1.

K-means:

experiment settings and results:

gamma_c = 1 / (255 * 255)

gamma_s = 1 / (100 * 100)

K = 2

image1:

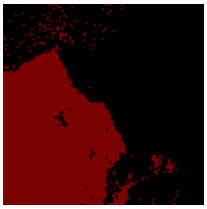
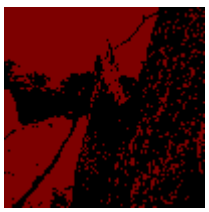


image2:



Spectral Clustering:

ratio cut:

experiment settings and results:

gamma_c = 1 / (255 * 255)

gamma_s = 1 / (100 * 100)

K = 2

image1:

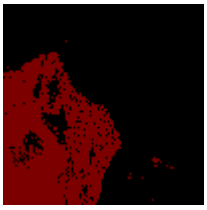
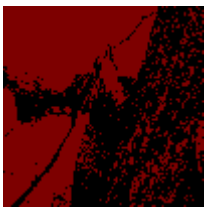


image2:



normalized cut:

experiment settings and results:

gamma_c = 1 / (255 * 255)

gamma_s = 1 / (100 * 100)

K = 2

image1:

image2:

discussion:

I think two clusters is not strong enough to represent the information in two image and it would be better when there are more clusters.

Part 2.

K-means:

experiment settings and results:

gamma_c = 1 / (255 * 255)

gamma_s = 1 / (100 * 100)
K = 3
image1:

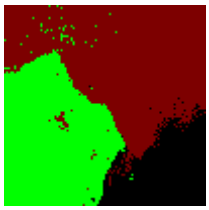
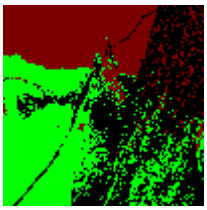


image2:



experiment settings and results:

gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 4
image1:

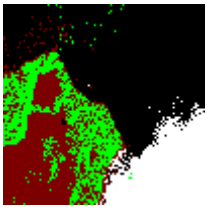
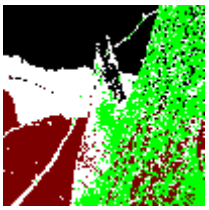


image2:



Spectral Clustering:

ratio cut:

experiment settings and results:

gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 3
image1:

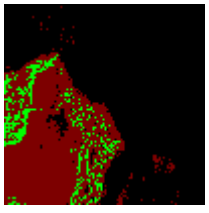
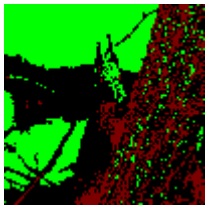


image2:



normalized cut:

experiment settings and results:

gamma_c = 1 / (255 * 255)

gamma_s = 1 / (100 * 100)

K = 3

image1:

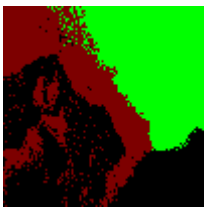
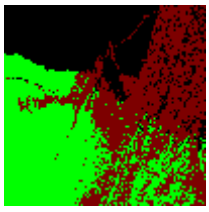


image2:



ratio cut:

experiment settings and results:

gamma_c = 1 / (255 * 255)

gamma_s = 1 / (100 * 100)

K = 4

image1:

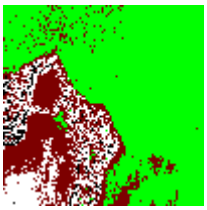
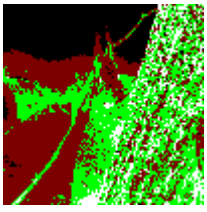


image2:



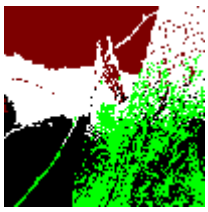
normalized cut:

experiment settings and results:

gamma_c = 1 / (255 * 255)

gamma_s = 1 / (100 * 100)
K = 4
image1:

image2:



discussion:

I think three clusters will be better for both image. However, the squirrel on the image2 tend to be blend in with tree or the background.

Part 3.

K-means:

experiment settings and results:

initial method = Forgy
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 2
image1:

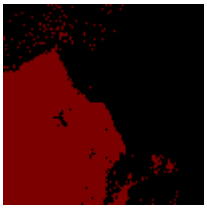
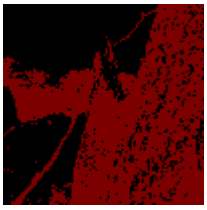


image2:



initial method = Forgy
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 3
image1:

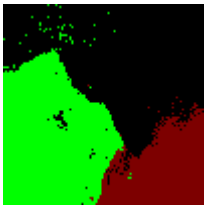
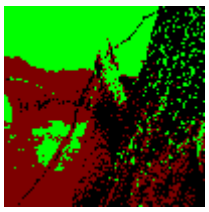


image2:



initial method = Forgy
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 4
image1:

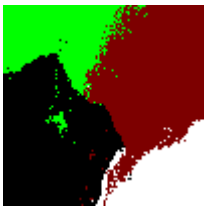
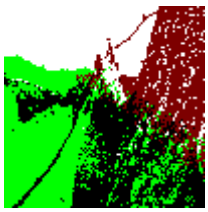


image2:



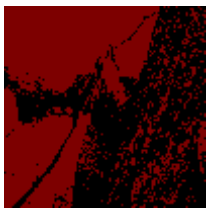
Spectral clustering:

ratio cut:

experiment settings and results:

initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 2
image1:

image2:



initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)

K = 3

image1:

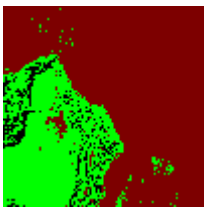
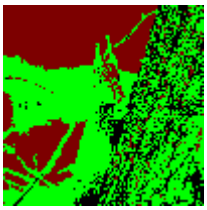


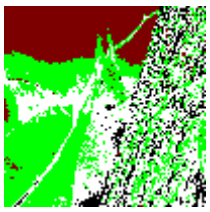
image2:



Initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 4

image1:

image2:



normalized cut:

experiment settings and results:

initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 2

image1:

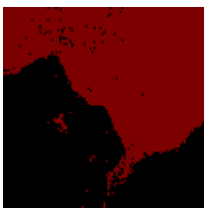
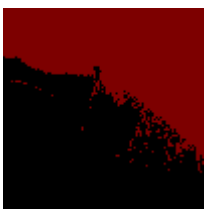


image2:



initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 3
image1:

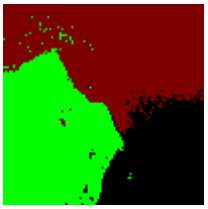
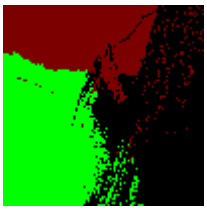
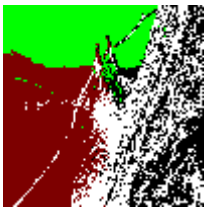


image2:



Initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 4
image1:

image2:



discussion:

I use “Forgy” as another initial methods, and the method is already considered spatial information, so it converges more quickly than random partition most of the time. “K-means++” also converges very fast, because

the method initially consider the most suitable central points.

Part 4.

ratio cut:

experiment settings and results:

initial method = random partition
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 2
image1:

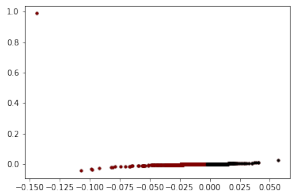
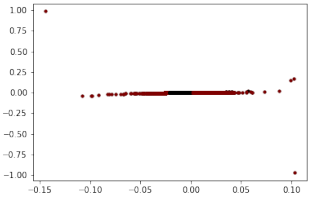


image2:



initial method = random partition
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 3
image1:

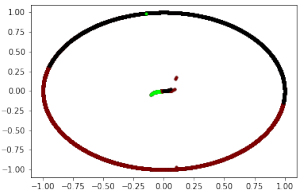
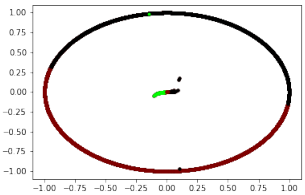


image2:



initial method = random partition
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 4

image1:

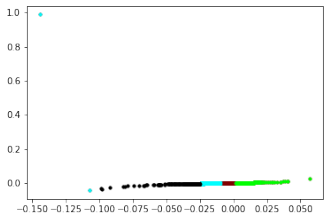
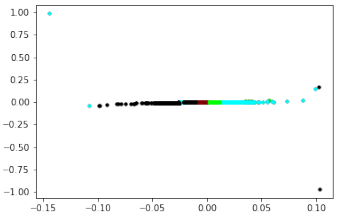


image2:



normalized cut:

experiment settings and results:

initial method = random partition
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 2
image1:

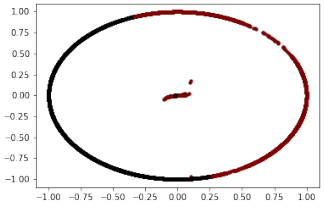
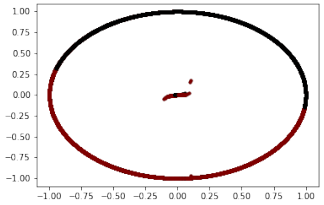


image2:



initial method = random partition
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 3
image1:

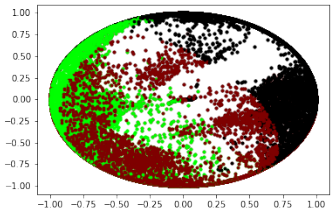
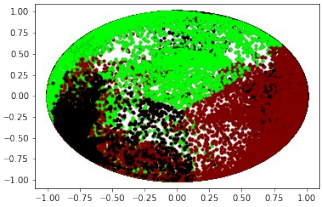


image2:



Initial method = random partition
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 4
image1:

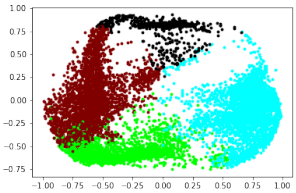
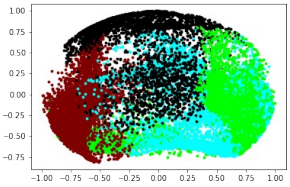


image2:



ratio cut:

experiment settings and results:

initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 2
image1:

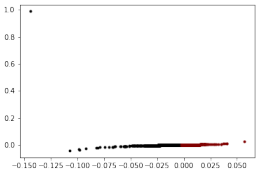
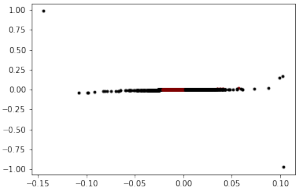


image2:



initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 3
image1:

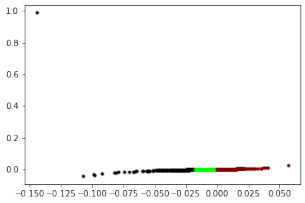
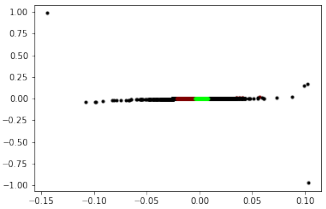


image2:



Initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 4
image1:

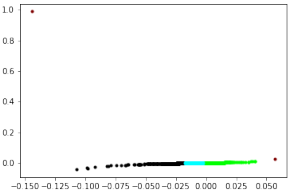
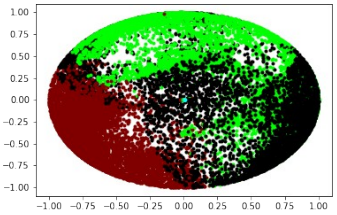


image2:



normalized cut:

experiment settings and results:

initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 2
image1:

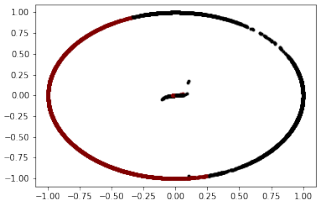
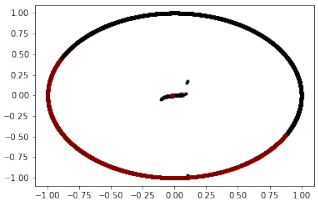


image2:



initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 3
image1:

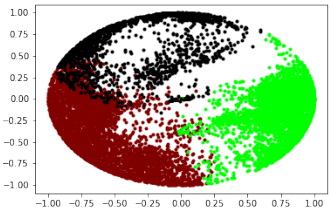
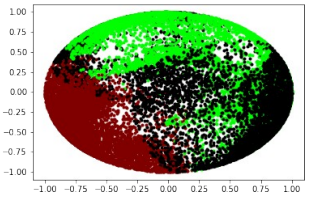


image2:



Initial method = k-means++
gamma_c = 1 / (255 * 255)
gamma_s = 1 / (100 * 100)
K = 4
image1:

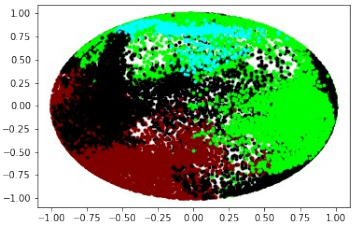
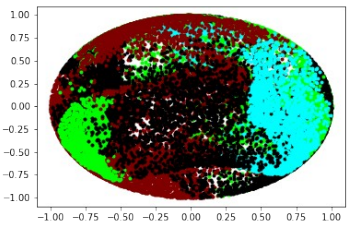


image2:

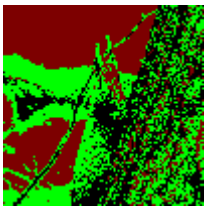


discussion:

It seems the data point of same cluster tend to be stayed together and have an obvious border but some of the data point with different cluster may overlap. I think the reason is that I only use the first two column of data point and I think they would be well distinguished if it can show high-dimensional plot.

3. observations and discussion

I observe that the squirrel in image2 tend to be blend in with tree or the background, so I try to get rid out of spatial information as I think this is the reason why the squirrel is not well clustered.



However, the result is not what I expect. Therefore, I think there is something else that I have not came up with.