NCTU Machine Learning 2021Spring HW6

Kernel K-means and Spectral Clustering

0610168 沈立淞

Kernel K-means Part

I. Code Explanations:

Kernel k-means is a variation of k-means clustering. This could be applied in solving the problem with no linear separable data set.

The key idea is to project the original data point into the new feature space, and then apply the original k-means algorithm. With kernel tricks, we can just implement the kernel k-means with kernel function and kernel matrix, without knowing the feature space.

Figure 1 shows how I implement 'data processing' and global parameters. The input data is an image with size 100X100X3(R G B pixel). For the sake of convenience for training, I reshape the image to a 10000X3 numpy array.

For parameters:

data size: the total number of pixels, which is 100X100 = 10000.

rs and rc: parameters for clustering function(mentioned below),

cluster_num: the target number of clusters we want.

max_iteration: the maximum iteration for doing kmeans clustering.

```
data_size = 10000
rs = 1/(data_size)
rc = 1/(256*256)
cluster_num = 4
max_iteration = 40
img_name = 'image1.png'
img = cv2.imread(img_name)
img = img.reshape((-1, 3))
img = img.astype(float)
```

Fg.1 data processing and parameters

Figure 2 shows the procedure of my kernel k-means algorithm. Parameter ***method*** here indicates the method to initialize k-means clustering centroids and all labels for each data point.

```python
def kernel_kmeans(img,method):
    labels, cluster_cnt= init(img, method)
    kernel_mat = kernel_function(img)
    iteration = 0
    while(iteration < max_iteration):
        iteration +=1
        print('iteration:{}'.format(iteration))
        pre_labels = np.copy(labels)
        mask = get_mask(labels)
        term2 = term_2(cluster_cnt, kernel_mat, mask)
        term3 = term_3(cluster_cnt, kernel_mat, mask)
        for i in range(data_size):
            distance_vector = []
            for k in range(cluster_num):
                distance = kernel_mat[i,i] - term2[i,k] + term3[k]
                distance_vector.append(distance)
            predict_label = np.argmin(distance_vector)
            labels[i] = predict_label
        error = error_cal(labels, pre_labels)
        cluster_cnt = count_cluster_num(labels)
        print('Error:{}'.format(error))
        visualization(labels, iteration,method)
        if(error == 0):
            break
    return labels
```

Fg.2 kernel k-means clustering

First, using ***init(img, method)*** (shown in figure 3, 4 and 5) to initialize labels for each data pointm, and compute the cluster size for each clusters. For the method, I use **random**, **kmeans++** and **mod** method to complete the initializing task.

For random method, just randomly assign a label sign based on the number of our deired number of clusters to each data point.

```python
def init(img, method):

    '''
    initialize the centroids and labels(indicating what cluster is belonging to for each data point)
    using 3 methods here: random, kmeans++ and mod method
    '''

    '''
    RANDOM:
    randomly assign a label sign to each data point
    '''

    if (method == 'random'):
        labels = np.random.randint(cluster_num, size = data_size)
        values, cluster_cnt = np.unique(labels, return_counts=True)
        visualization(labels, 0, 'random')
        return labels, cluster_cnt
```

Fg.3 init: random method

For kmeans++, (1)randomly pick up one data point as first centroid, (2)for each data point x, compute the square difference between the nearest chosen centroid and itself, noted as D(x)(3) The higher the D(x) is, the higher probability the x has to be chosen as new centroid. Choose a new centroid based on the probability. Repeat(2)、(3) until we get enough centroids.

```
'''
KMEANS++:
1. randomly pick up one data point as first centroid
2. for each data point x, compute the short distance between each discovered centroid and itself, set as D(x)
3. the larger the D(x) is , x has more probability to be chosen as new centroid,
'''
elif (method == 'kmeans++'):
    centroid = np.zeros((cluster_num, 2))
    centroid_1 = np.random.randint(0, 100, (1,2)) #randomly pick up a pixel in 100X100 image
    centroid[0,:] = centroid_1

    for k in range(1, cluster_num):
        distance_vector = []
        for i in range(100):
            for j in range(100):
                dist = []
                for cluster_idx in range(k):
                    val = (i-centroid[cluster_idx, 0])**2 +  (j-centroid[cluster_idx, 1])**2
                    dist.append(val)
                min_dist = np.min(dist)
                distance_vector.append(min_dist)

        #prob is the probability of all pixel to be chosen as centroid
        prob = distance_vector / np.sum(distance_vector)
        #choose new_centroid  based on prob
        new_centroid = np.random.choice(range(data_size), p=prob)
        centroid[k,0] = int(new_centroid/100)
        centroid[k,1] = int(new_centroid%100)
    labels = clustering(centroid)
    values, cluster_cnt = np.unique(labels, return_counts=True)
    visualization(labels, 0, 'kmeans++')
    return labels, cluster_cnt
```

Fg.4 init: kmeans++

For mod method, assign labels **(i%cluster_num)** to data point I.

```
elif (method == 'mod'):
    labels = np.zeros((data_size),dtype=int)
    for i in range(data_size):
        labels[i] = i%cluster_num
    values, cluster_cnt = np.unique(labels, return_counts=True)
    visualization(labels, 0, 'mod       ')
    return labels, cluster_cnt
```

Fg.5 init: mod

Secondly, compute kernel matrix using ***kernel_function(img)*** . The kernel function is defined as follows: multiplying two RBF kernel to consider both spatial similarity and color similarity at the same time.

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

rs, rc is functional parameter which can be tuned. I set rs = 1/(data_size), and rc = 1/(256*256). S(x) is the 2d coordinate of each pixel on original image, C(x) is the RGB value of each pixel.

```python
def kernel_function(img):

    '''
    compute kerlen function using two RBF kernel to consider
    spatial similarity and color similarity at the same time.
    '''
    pos = np.zeros((10000,2),dtype=int)
    for i in range(100):
        for j in range(100):
            pos[100*i+j][0] = i
            pos[100*i+j][1] = j
    pos_norm = np.sum(pos**2, axis = -1)
    M1 = pos_norm[:,None] + pos_norm[None,:] - 2 * np.dot(pos, pos.T)
    img_norm = np.sum(img**2, axis = -1)
    M2 = img_norm[:,None] + img_norm[None,:] - 2 * np.dot(img, img.T)
    kernel_mat = np.exp((-rs * M1) + (-rc * M2))
    return kernel_mat
```

Fg.6 kernel_matrix function

For each iteration, using labels and ***get_mask(labels)*** to get indicator vector for each clusters.

```python
def get_mask(labels):
    '''
    compute indicator vector foe each cluster
    '''
    mask = []
    for k in range(cluster_num):
        mask.append(np.where(labels!=k,0,1).reshape(-1,1))
    mask = np.array(mask)
    return mask
```

Fg.7 compute indicator vector.

And then, it's time to do the EM step in k-means algorithm. We can use the following formula to compute the distance between each data point and centroids in feature space. For each point Xj, we assign the nearest centroid in feature space as Xj's label.

We firstly use the following formula for all Xj belonging to data points, and all μ_k belonging to centroids, then iteratively traverse all data point to assign the nearest centroids as their labels.

$$\|\emptyset(Xj) - \mu_k\| = \mathrm{k(Xj,Xj)} - \frac{2}{|C_k|}\sum_n \alpha_{kn}\mathrm{k(Xj,Xn)} + \frac{2}{|C_k|^2}\sum_p\sum_q \alpha_{kp}\alpha_{kq}\,\mathrm{k(Xj,Xq)}$$

$$= \mathrm{k(Xj,Xj)} - \frac{2}{|C_k|}M_{row\ j} * f_k + \frac{2}{|C_k|^2}{f_k}^T * M * f_k$$

$$= term1 - term2 + term3$$

M is kernel matrix, f_k is indicator vector for cluster k.

Figure. 8 and 9 shows the implementation of computing term2 and term3.

```
def term_2(cluster_cnt, kernel_mat, mask):
    output = np.zeros((data_size, cluster_num), dtype=float)
    for k in range(cluster_num):

        if(cluster_cnt[k]!=0):
            term2_vector =  (2/cluster_cnt[k])*kernel_mat.dot(mask[k])
        else:
            term2_vector =  (2)*kernel_mat.dot(mask[k])
        output[:,k] = term2_vector[:,0]
    return output
```

Fg.8 compute term2

```
def term_3(cluster_cnt, kernel_mat, mask):
    output = np.zeros((cluster_num), dtype=float)
    for k in range(cluster_num):
        if(cluster_cnt[k]!= 0):
            term3_value = (1/cluster_cnt[k]**2) * mask[k].T @ kernel_mat @mask[k]
        else:
            term3_value = mask[k].T @ kernel_mat @mask[k]
        output[k] = term3_value
    return output
```

Fg.9 compute term3

After assigning new labels to each points, use ***error_cal(labels, pre_labels):*** to compute the difference between previous labels and new labels. Use ***count_cluster_num(labels)*** to re-compute cluster sizes for each clusters.

The whole algorithm will continue until the error becomes 0, or reaching maximum iteration.

```
def error_cal(labels, pre_labels):
    diff = abs(labels - pre_labels)
    error = np.sum(diff)
    return error
```

Fg.10 compute error.

```
def count_cluster_num(labels):
    cluster_cnt = np.zeros((data_size), dtype=int)
    for i in range(data_size):
        cluster_cnt[labels[i]]+=1
    return cluster_cnt
```

Fg.11 compute cluster sizes.

The last thing is to do visualization, and we can just assign different colors in for

different clusters for each pixel. The code is shown in figure. 12.

```
def visualization(labels, iteration, method):
    img = cv2.imread(img_name)
    color = [(0,0,0), (255,0,0), (0,255,0), (0,0,255), (255,255,0), (255,0,255), (0, 255, 255) ,(255,255,255)]

    for i in range(100):
        for j in range(100):
            img[i,j] = color[labels[i*100+j]]
    cv2.imwrite('image_result'+method +'_it' + str(iteration)+'.png', img )
```

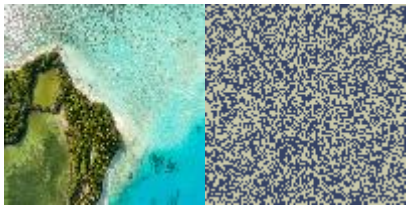
Fg.12 visualization

II. Experimenting and results:

Image1

2 clusters:

Iteration 0



Iteration 0

Iteration 0

Iteration 14

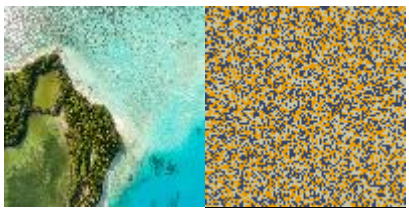
Iteration 8

Iteration 8

Random **K-means++** **Mod**

3 clusters:

Iteration 0



Iteration 0

Iteration 0

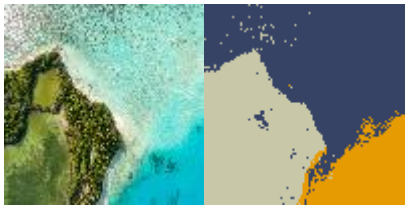
Iteration 14

Random

Iteration 32

K-means++

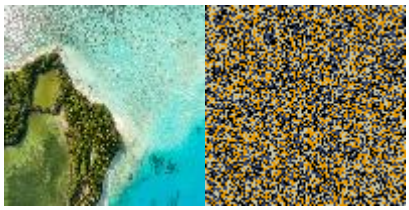
Iteration 34



Mod

4 clusters:

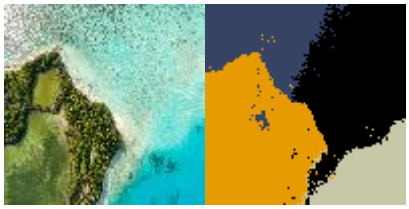
Iteration 0



Iteration 0

Iteration 0

Iteration 19

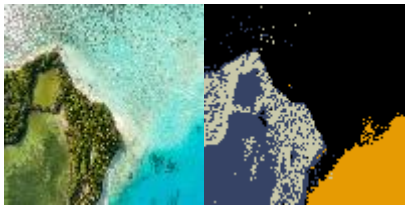


Random

Iteration 17

K-means++

Iteration 18



Mod

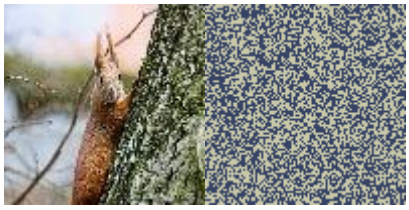
Intuitively, the k-means++ have the fastest speed to converge in the algorithm, for it has higher probability to get appropriate initial centroids. However, in the experiment, it doesn't get much advantage over others. In 2 clusters, it runs 8 iterations, which is better than random, but the same as mod. In 3 clusters, it run 32 iterations, which is slower than random. And for 4 clusters, it has the fewest iteration. The experiment shows that there is no significant difference among each method's converge speed.

For the clustering result, I think there's no much difference among each method when cluster number is low. However, as we shall see in 4 cluster, Mod method has obviously different result in down-left corner.

Image2

2 clusters:

Iteration 0



Iteration 0

Iteration 0

Iteration 11

Iteration 21

Iteration 13

Random **K-means++** **Mod**

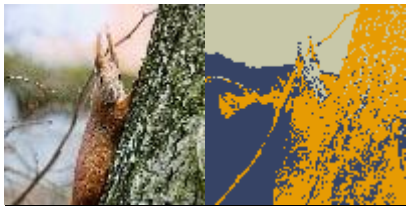
3 clusters:

Iteration 0

Iteration 0

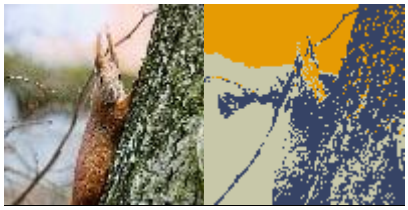
Iteration 0

Iteration 25



Iteration 13

Iteration 29



Random **K-means++** **Mod**

4 clusters:

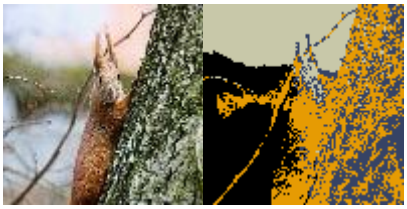
Iteration 0

Iteration 29

Random

Iteration 0

Iteration 36



Kmeans++

Iteration 0

Iteration 17

Mod

Perhaps the higher complexity of image2, it takes larger iterations to converges. We can see that in 4 clusters, Mod method has difficulty discriminate the object on the tree. With the result in image1, I think there's little advantage to use mod method when it comes to larger cluster num, for the way to assign labels may make it harder to reshape the original object.

Kernel K-means Part

I. Code Explanations:

Spectral Clustering is another method to do clustering, which is popular as it can find clusters of almost arbitrary shape. The main idea is to construct similarity graph by kernel function or other methods(e.g. knn), then transform the original data into eigen space, and finally use k-means to get the result clustering.

Data Preprocessing and Parameters:

For data preprocessing and global parameters, it's just the same as kernel k-means.

```
data_size = 10000
rs = 1/(data_size)
rc = 1/(256*256)
cluster_num = 2
max_iteration = 40
img_name = 'image1.png'
img = cv2.imread(img_name)
img = img.reshape((-1, 3))
img = img.astype(float)
```

Fg.13 Spectral Clustering data preprocessing and global parameters

Spectral Clustering algorithm:

Figure. 14 shows the main algorithm for Spectral Clustering. Just the same as kernel k-means, we set parameter **method** to define the way to initialize the centroids. And for similarity graph, we use the kernel matrix defined in kernel k-means to represent as our similarity graph.

```
methods = ['random', 'kmeans++']
for method in methods:
    print('target clusters:{}'.format(cluster_num))
    kernel_mat = kernel_function(img)

    #ratio cut
    print('ratio cut')
    L = laplacian_matrix(kernel_mat, 'ratio')
    U = eigmat_calulate(L)
    kmeans(U, 'ratio cut', method)

    #normalized  cut
    print('normalized cut')
    L_sym = laplacian_matrix(kernel_mat, 'normalized')
    U = eigmat_calulate(L_sym)
    T = normalizing_rows(U)
    kmeans(U, 'normalized cut', method)
```

Fg.14 Spectral Clustering algorithm

```
def kernel_function(img):       #O(n^2)
    pos = np.zeros((10000,2),dtype=int)
    for i in range(100):
        for j in range(100):
            pos[100*i+j][0] = i
            pos[100*i+j][1] = j
    pos_norm = np.sum(pos**2, axis = -1)
    M1 = pos_norm[:,None] + pos_norm[None,:] - 2 * np.dot(pos, pos.T)
    img_norm = np.sum(img**2, axis = -1)
    M2 = img_norm[:,None] + img_norm[None,:] - 2 * np.dot(img, img.T)
    kernel_mat = np.exp((-rs * M1) + (-rc * M2))
    return kernel_mat
```

Fg.15 Similarity graph (kernel matrix)

Next, we need to construct our graph laplacian.

Ratio cut:

For ratio cut, we use the following formula to compute graph laplacian:

$$\mathrm{L} = \mathrm{D} - \mathrm{W}$$

D is a diagonal, degree matrix for every nodes

W is the original similarity graph

Nomalized cut:

For normalized cut, we use the following formula to compute graph laplacian:

$$\mathrm{L}_{sym} = \mathrm{I} - D^{-1/2}\mathrm{W}D^{-1/2}$$

D is a diagonal, degree matrix for every nodes

W is the original similarity graph

```
def laplacian_matrix(kernel_mat, cut):
    if (cut == 'ratio'):
        D = np.sum(kernel_mat, axis=1)
        L = D - kernel_mat
        return L
    if (cut == 'normalized'):
        D  = np.diag(np.sum(kernel_mat, axis=1))
        D_sqrt = np.diag(np.power(np.diag(D), -0.5))
        L_sym = np.identity(data_size) - D_sqrt @ kernel_mat @ D_sqrt
    return L_sym
```

Fg.15 Similarity matrix(kernel matrix)

Compute Eigenvectors:

For the similarity graph is constructed by kernel function, which means our similarity graph is a connected graph, we need to discard the first eigenvector corresponding to smallest eigenvalue (zero). Compute eigenvector and eigenvalues by **numpy.linalg.eig(mat)**, get N.O. 2 to k+1 eigenvectors corresponding to 2 to k+1 smallest eigenvalues.

```
def eigmat_calulate(mat):
    print('computing eigen...')
    eigenVal, eigenVec = linalg.eig(mat)
    idx = np.argsort(eigenVal)[1: cluster_num+1]
    U = eigenVec[:,idx].real.astype(np.float32)
    return U
```

Fg.16 Compute eigenvector

For normalized cut, we need to normalized our eigenmatrix (U) gotten above.
Use following formula to do normalize:

$$t_{ij} = u_{ij}/(\sum_k u_{ik}{}^2)^{1/2}$$

```
def normalizing_rows(U):
    T = np.empty_like(U)
    for i in range(len(U[:,0])):
        denominator = math.sqrt(np.sum(U[i,:]**2))
        T[i,:] = U[i,:] / denominator
    return T
```

Fg.17 normalizing eigen matrix

K-means:

After we get eigen matrix, we could view each row as data with k-dimension. Using the data we created ,we can then just apply normal k-mean algorithm to get result. The overall process is just the same as what we have done in kernel k-means.

```
def kmeans(data, cut, method):
    print(data)
    centroid = centroid_init(data, method)
    print('centroid:{}'.format(centroid))
    labels = np.zeros((data_size), dtype = int)
    print(labels)
    iteration = 0
    while(iteration < max_iteration):
        iteration +=1
        print('iteration:{}'.format(iteration))
        pre_labels = np.copy(labels)
        labels = clustering(data, centroid, labels)
        print(labels)
        error = error_cal(labels, pre_labels)
        print('Error:{}'.format(error))
        centroid = update_centroid(data, labels, centroid)
        visualization(labels, iteration, method, cut)
        if (error==0):
            eigen_visualization(labels, data, cut, method)
            break
```

Fg.18 k-means algorithm implementation

To initialize centroids, we use **centroid_init(data, method).** I implement two method here: random and k-means++.

For random method, I use uniform distribution to randomly get data for each dimension based on each dimension's min and max.

For k-means++, the overall implementation is just the same as what we do in kernel k-means.

```
def centroid_init(data, method):

    # get centroid by sampling from uniform distribution for k-dimention data
    if(method == 'random'):
        data_min = data.min(axis=0)
        data_max = data.max(axis=0)
        diff = data_max - data_min
        centroid = np.random.rand(cluster_num, cluster_num)
        for i in range(cluster_num):
            centroid[i,:] = data_min[i] + diff[i] * centroid[i,:]
        return centroid
    if (method == 'kmeans++'):
        data_min = data.min(axis=0)
        data_max = data.max(axis=0)
        diff = data_max - data_min
        centroid = np.random.rand(cluster_num, cluster_num)
        for i in range(cluster_num):
            centroid[i,:] = data_min[i] + diff[i] * centroid[i,:]
        for k in range(1, cluster_num):
            distance_vector = []
            for i in range(data_size):
                dist = []
                for cluster_idx in range(k):
                    val = np.sum((data[i,:] - centroid[cluster_idx,:])**2)
                    dist.append(val)
                min_dist = np.min(dist)
                distance_vector.append(min_dist)
            prob = distance_vector / np.sum(distance_vector)
            new_centroid = np.random.choice(range(data_size), p=prob)
            centroid[k,:] = data[new_centroid,:]
        return centroid
```

Fg.19 initialize centroids

For each iteration, we cluster all data point and give them labels based on previous centroids. For each data point, we could just check all centroids and using square distance to and assign the smallest distance centroid as the data point's label.

```
def clustering(data, centroid, labels):
    for i in range(data_size):
        dist_vec = []
        for k in range(cluster_num):
            dist = np.sum((data[i,:] -  centroid[k,:] )**2)
            dist_vec.append(dist)
        cluster_idx = np.argmin(dist_vec)
        labels[i] = cluster_idx
    return labels
```

Fg.20 clustering

After assigning labels to each data point, compute the difference with the previous labels. The whole algorithm will continue until the error reaches 0 or iteration reachs maximum iteration.

```
def error_cal(labels, pre_labels):
    diff = abs(labels - pre_labels)
    error = np.sum(diff)
    return error
```

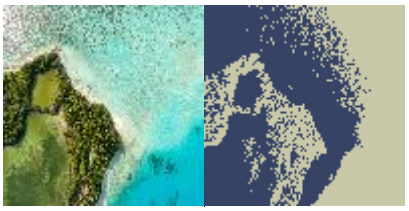
Fg.21 error calculating

II. Experimenting and results:

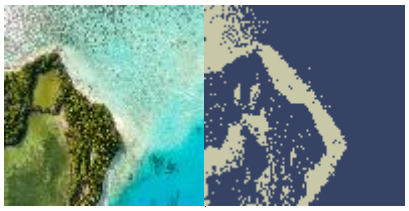
Image1

2 clusters ratio cut:

Iteration 1

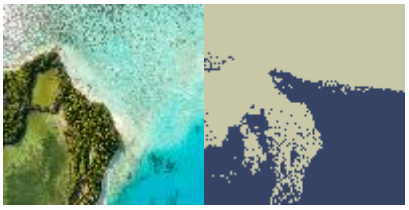


Iteration 1

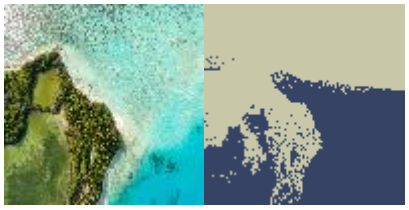


Iteration 31

Iteration 40



Random



K-means++

2 clusters normalized cut:

Iteration 1

Iteration 1

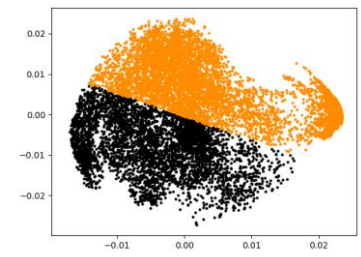
Iteration 18

Random

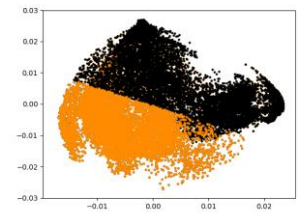
Iteration 9

K-means++

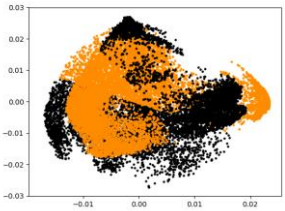
2 clusters eigen space:

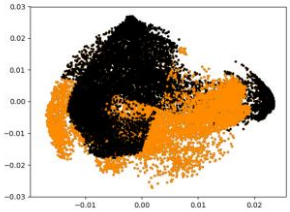


Random, ratio cut



K-means++,ratio cut

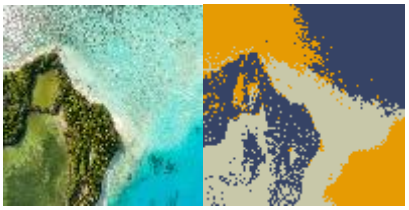




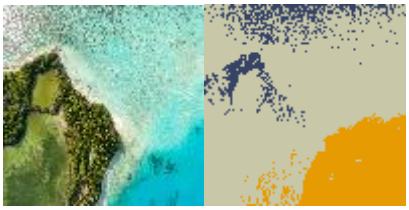
Random, normalized cut **K-means++, normalized cut**

3 clusters ratio cut:

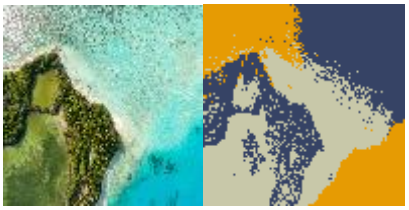
Iteration 1



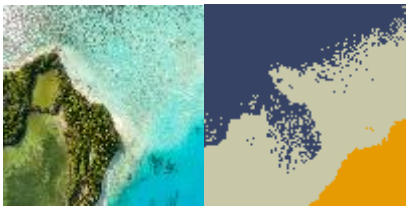
Iteration 1



Iteration 19



Iteration 17

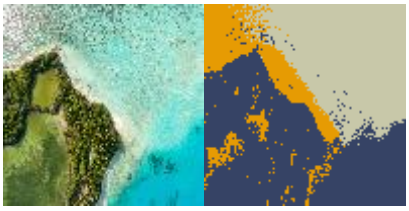


Random **K-means++**

3 clusters normalized cut:

Iteration 0

Iteration 0



Iteration 24

Iteration 20

Random **K-means++**

Image2

2 clusters ratio cut:

Iteration 1

Iteration 18

Random

Iteration 1

Iteration 12

K-means++

2 clusters normalized cut:

Iteration 1

Iteration 22

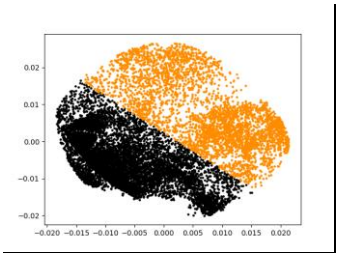
Random

Iteration 1

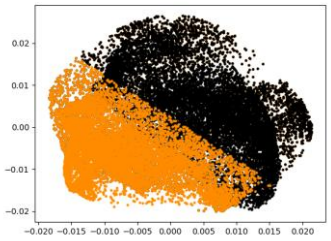
Iteration 20

K-means++

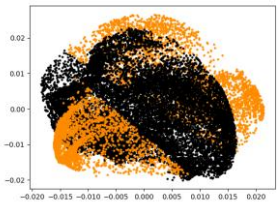
2 clusters eigen space:



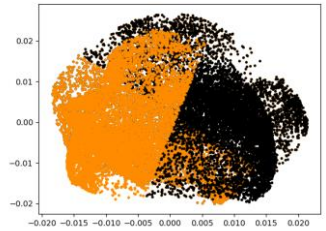
Random, ratio cut



K-means++,ratio cut



Random, normalized cut



K-means++, normalized cut

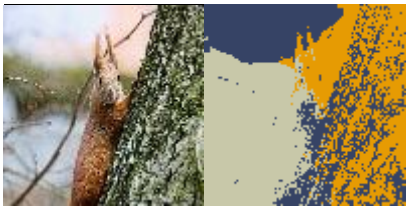
3 clusters ratio cut:

Iteration 1

Iteration 1

Iteration 34

Iteration 22



Random

K-means++

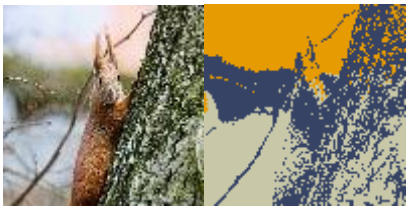
3 clusters normalized cut:

Iteration 1

Iteration 1

Iteration 22

Iteration 11



Random **K-means++**

As we shall see, spectral clustering has more clearer contour than kernel k-means in complicated image like image2. This shows the feature that spectral clustering has the ability to cluster arbitrary shape.

Observation and Discussion

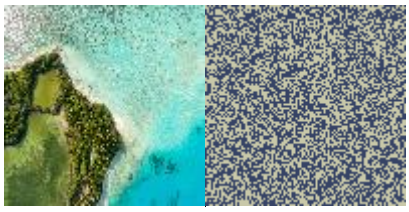
The kernel function here consider both spatial effect and color effect. I'm curious about what if we just use color effect as our kernel function? Therefore, I decide to try kernel k-means with modified kernel, which removes the spatial RBF kernel term in original kernel function. i.e.:

$$\text{new } \mathrm{k}(\mathrm{x}, \mathrm{x}') = e^{-rc\|C(x)-C(x')\|^2}$$

Image1

2 clusters:

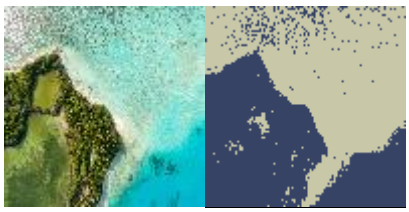
Iteration 0



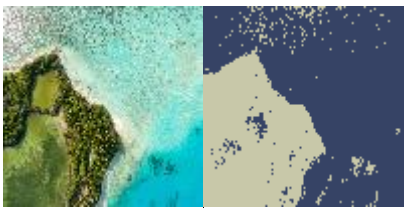
Iteration 0

Iteration 0

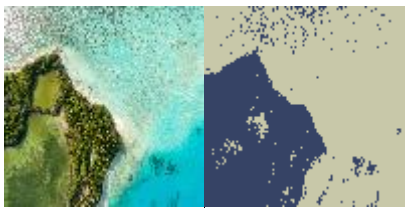
Iteration 6



Iteration 9



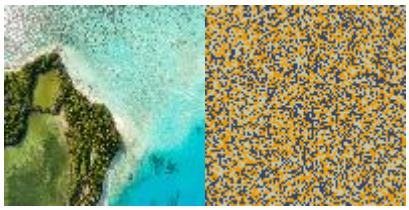
Iteration 9



Random **K-means++** **Mod**

3 clusters:

Iteration 0

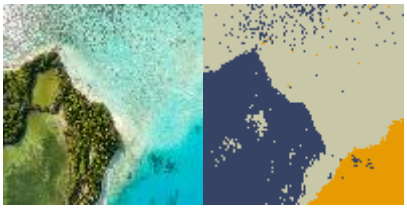


Iteration 0

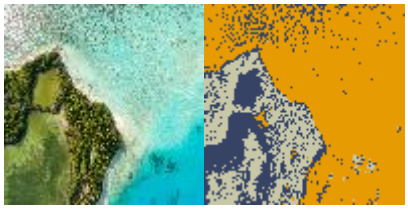
Iteration 0

Iteration 12

Iteration 7



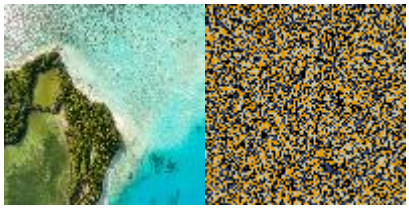
Iteration 10



Random **K-means++** **Mod**

4 clusters:

Iteration 0

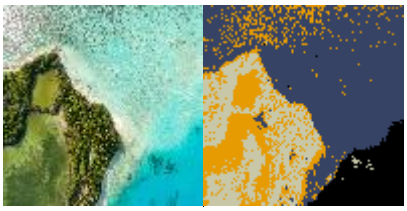


Iteration 0

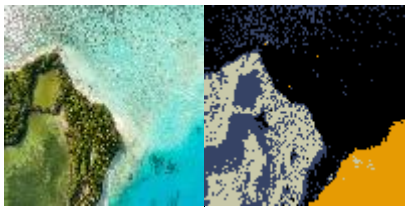
Iteration 0

Iteration 16

Iteration 14



Iteration 21

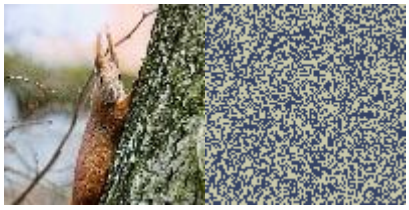


Random **K-means++** **Mod**

Image2

2 clusters:

Iteration 0



Iteration 0

Iteration 0

Iteration 18

Iteration 17

Iteration 16

Random **K-means++** **Mod**

3 clusters:

Iteration 0

Iteration 0

Iteration 0

Iteration 18

Iteration 16

Iteration 13

Random **K-means++** **Mod**

4 clusters:

Iteration 0

Iteration 0

Iteration 0

Iteration 17

Iteration 40

Iteration 18

Random

K-means++

Mod

As we shall see in image1, the overall contour could be determined by the modified kernel. I guess that because image1 is simple, and the color in image is few, the colors in a single object is similar, and the color between different object has much difference.

As for image2, for the complexity of the image, object's contour basically can't be determined my our modified kernel. The situation gets worse when cluster number gets higher.