a. code with detailed explanations (40%)

> The implementation's detail writes in the program comments.
>
> **Kernel Kmeans (part1 & part2 & part 3)**
>
> > The goal of kernel kmeans is to minimize the distance between the data points and the centers in the "feature" space, so I use the formula at PPT p.22, $\arg\min_{(C_1,u_1),\ldots,(C_k,\mu_k)} \sum_{i=1}^{k}\sum_{x_j\in C_i} ||\phi(x_j)-\mu_k^{\phi}|| = k(x_j,x_j) - \frac{2}{|C_k|}\sum_n \alpha_{kn}k(x_j,x_n) + \frac{1}{|C_k|^2}\sum_p\sum_q \alpha_{kp}\alpha_{kq}k(x_p,x_q)$, where $\alpha_{kn} = 1$ if data point x_n is assigned to the k-th cluster, in the function DistanceKernel.
> >
> > Also, kernel kmeans uses the kernel trick which defined as $k(x,x') = \exp(-\gamma_s||S(x)-S(x')||^2) * \exp(-\gamma_c||C(x)-C(x')||^2)$ in the function Kernel, default γ_s and γ_c is 0.001.
> >
> > **[part 1]** The algorithm of kernel kmeans and the visualization setting are defined in the function of KernelKmeans. First it calls the Initialization function to get the initial labels, and then put the original image into the Kernel function to get the similarity of pairwise datapoints. Second it starts to do the clustering procedure until the label difference is smaller than 1e-10. In the procudure it use the function of DistanceKernel to compute the distance between data points and the center in the feature space (M step), and then labelizing each data points based on the shortese distance's center(E step). Third, it will do the visualization setting to store the clustering result. Finally after it converges, it returns the visualization during the procedure which is stored in the variable images.
> >
> > **[part 2]** Different number of clusters is defined in the main as variable ks whose value is from 2 to 5.
> >
> > **[part 3]** The initialization of kernel kmeans has two methods, random and kmeans++, which is defined in the function Initialization. The initialization method of random randomly picks k centers and makes each cluster will have at least one data point belonging to them. The initialization method of kmeans++ first randomly pick a center and then continue pick the next k-1 centers based on the probability of $\frac{distance(x)^2}{\sum distance(x)^2}$, where the $distance(x)$ is defined as the distance between the data point and the "nearest" center. In this way, the farer distance is more likely to be chosen and the k centers will have more chance to be far away to each others. After choosing the k centers, it returns each data points' belonging clusters.

```python
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.image as mpimg
from scipy.spatial.distance import pdist,cdist,squareform
from array2gif import write_gif

#refer to PPT p.22
def DistanceKernel(image_kernel,labels,c):
 len_image_kernel = len(image_kernel)
 k_xj_xj = np.diag(image_kernel).reshape(-1,1) #the similarity between each pixel and itself
 Ck = np.sum(labels == c) #the number of pixels belonging to cluseter c
 k_xj_xn = image_kernel[:, labels == c] #the similarity between each pixel xj and the other
pixel xn which belongs to cluster c
 k_xp_xq = image_kernel[labels == c,:][:,labels == c] #the similarity between the pairwise
pixels that both belonging to cluster c; first trancate the row, then trancate the column to
extract the pairwise pixels both belonging to cluster c
 distance = k_xj_xj - (2/Ck)*np.sum(k_xj_xn,axis=1).reshape(-1,1) + (1/(Ck**2))*np.sum(k_xp_xq)
#the distance between each pixel and the center in the feature space
 return distance.reshape(-1)


def Kernel(C,gamma_s=1e-3,gamma_c=1e-3):
 height,width = int(np.sqrt(C.shape[0])),int(np.sqrt(C.shape[0]))
 #color information : C =>(10000,3)
 #spatial information : S =>(10000,2)
 S = np.zeros((height*width,2)) #the coordinate of the pixel
 for h in range(height):
     for w in range(width):
         S[height*h+w] = [h,w]

 #pdist : compute pairwise distances between observations in "one" n-dimensional space. =>
10000*(10000-1)/2
 #cdist : compute distance between each pair of the "two" collections of inputs. =>
(10000,10000)
 return np.exp(-gamma_s*cdist(S,S,'sqeuclidean')) * np.exp(-gamma_c*cdist(C,C,'sqeuclidean'))

def Initialization(image,k,init_type):
 labels = None #to let first while run
 while len(np.unique(labels))!=k: #until every cluster has at least one pixel
     #initialize k-menas' centers
     height,width,nDimensions = int(np.sqrt(image.shape[0])), int(np.sqrt(image.shape[0])),
image.shape[1]
     if init_type == 'kpp':
         #k-means++ ,refer to https://www.cnblogs.com/yixuan-xu/p/6272208.html
         #center initialization
         centers = np.zeros((k,nDimensions)) #initialize k cluseters' center
         centers[0] = image[np.random.randint(height*width)] #randomly choose first cluseter's
center

         #compute the left k-1 centers
         for c in range(1,k): #k-1 centers c
             #compute the distance of every data points to the nearest center
             for c_computed in range(c): #choose the "nearest" center
                 distance = np.sqrt(np.sum((image - centers[c_computed])**2,axis=1))

             #pick the next center based on the probability of [(distance(X)^2)/(sum of
distance(x)^2)], the farer distance is more likly to be chosen
             probability = distance**2 / np.sum(distance**2) #probabiltiy
             mask = np.random.choice(height*width, p=probability) #randomly choose one pixel
based on the probability
             centers[c] = image[mask] #assign next center

     elif init_type == 'random':
         #random(uniform)
         centers = np.zeros((k,nDimensions)) #initialize k cluseters' center
         mask = np.random.choice(height*width,size=k,replace=False) #randomly(uniform) choose k
clusters' center's index without replacement
         for c in range(k):
             centers[c] = image[mask[c]]

     #assign each pixel's label to the nearest cluster's center
     distance = np.zeros((k,height*width))
     for c in range(k): #for every centers
         distance[c] = np.sqrt(np.sum((image-centers[c])**2,axis=1))#the distance between each
data points and that particular center
     labels = np.argmin(distance,axis=0) #assign the nearest one
 return labels

#refer to https://github.com/algostatml/UNSUPERVISED-
ML/blob/master/KMEANS%20AND%20KERNEL%20VERSION/KERNEL%20KMEANS/kernelkmeans.py
def KernelKmeans(image,k,init_type):
 #initialization
 images = [] #to store gif
 labels_diff = 1e+10
 height,width,nDimensions = int(np.sqrt(image.shape[0])), int(np.sqrt(image.shape[0])),
image.shape[1]
 colormap = np.array([[255, 255, 255], [255, 255, 0], [255, 0, 255], [255, 0, 0], [0, 255,
255], [0, 255, 0], [0, 0, 255], [0, 0, 0]])#np.random.choice(range(256),size=(k,3)) #randomly
pick an color, RGB
 labels0 = Initialization(image,k,init_type) #initialize labels

 #set kernel
 image_kernel = Kernel(image)

 while labels_diff >= 1e-10:
     distance = np.zeros((k,height*width))
     for c in range(k): #M step: for every centers
         #distance[c] = np.sqrt(np.sum((image_kernel-centers_kernel0[c])**2,axis=1)) #compute
the distance between every data points and the centers
         distance[c] = DistanceKernel(image_kernel,labels0,c)
     labels1 = np.argmin(distance,axis=0) #E step: assign the pixel's label with the nearest
cluster's center

     labels_diff = np.sqrt(np.sum((labels1-labels0)**2))
     labels0 = labels1
     print("difference: ",labels_diff)

     #use labels to do visualization
     image_cur = np.zeros((height,width,3))
     for h in range(height):
         for w in range(width):
             image_cur[w,h] = colormap[labels0[height*h+w]] #not [h,w], because labels is from
the top to bottom and then from left to right, not from left to right and then from the top to
bottom
     images.append(image_cur.astype('uint8')) #store this time's image
     #plt.imshow(image_cur.astype('uint8'))
#https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html
     #plt.pause(0.1)
 return images


if __name__=='__main__':

 #load file and set initialization
 ks = [2,3,4,5]
 init_types = ['kpp','random']
 image_filenames = ['image1','image2']

 for image_filename in image_filenames: #for every images
     image = mpimg.imread('./data/{}.png'.format(image_filename)) #=>(100,100,3)
     image = (image * 255).astype('uint8') #convert float32(0~1) to uint8(0~255)
     height,width,colors = image.shape
     image = image.reshape(height*width,colors) #flatten ; =>(10000,3); image[h][w] =>
image[height*h+w]

     #part 2 : in addition to cluster data into 2 clusters, try more clusters (e.g. 3 or 4)
     for k in ks :
         #part 3 : try different ways to do initialization of k-means clustering eg. k-means++
         for init_type in init_types :
             #part 1 : show the clustering procedure of kernel k-means
             images=KernelKmeans(image,k,init_type)
```

```
            #print the process by gif
            write_gif(images,
'./result/KernelKmeans_{}_{}clusters_{}.gif'.format(image_filename,k,init_type))
            print('KernelKmeans_{}_{}clusters_{}
converge!'.format(image_filename,k,init_type))
```

Spectral Learning (part1 & part3 & part 4)

> Spectral Learning import Kmeans which is similar to kernel kmeans.
>
> Let's see what Kmeans does first.
>
> **[part of part 1]** The Initialization function also has two method, random and kmeans++, which is the same as definition in kernel kmeans' Initialization function. But now the Initialization function in k-means just returns the k centers, not each data points' labels.
>
> The Kmeans function implement the algoritm of kmeans. It first initialize the k centers by the Initialization function. Then it start to do the clustering with E step and M step until the difference between the centers. In E step, it try to cluster each data point to the nearest centers. In M step, it recomputes each cluster's centers. After the EM step, it record the clustering result into the images to do the visualization.

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.image as mpimg
from scipy.spatial.distance import pdist,cdist,squareform
from array2gif import write_gif

def Initialization(image,k,init_type):
 labels = None #to let first while run
 while len(np.unique(labels))!=k: #until every cluster has at least one pixel
     #initialize k-menas' centers
     height,width,nDimensions = int(np.sqrt(image.shape[0])), int(np.sqrt(image.shape[0])),
image.shape[1]
     if init_type == 'kpp':
         #k-means++ ,refer to https://www.cnblogs.com/yixuan-xu/p/6272208.html
         #center initialization
         centers = np.zeros((k,nDimensions)) #initialize k cluseters' center
         centers[0] = image[np.random.randint(height*width)] #randomly choose first cluseter's
center

         #compute the left k-1 centers
         for c in range(1,k): #k-1 centers c
             #compute the distance of every data points to the nearest: center
             '''
             distance = np.full((height*width),1e+10)
             for s in range(height*width): #for every data points
                 for c_computed in range(c): #choose the "nearest" center
                     distance_cur = np.sqrt(np.sum((image[s] - centers[c_computed])**2))
                     if distance_cur < distance[s] and np.all(image[s]!=centers[c_computed]):
#to avoid compare with itself
                         distance[s] = distance_cur
             '''
             for c_computed in range(c): #choose the "nearest" center
                 distance = np.sqrt(np.sum((image - centers[c_computed])**2,axis=1))
```

```
            #pick the next center based on the probability of [(distance(X)^2)/(sum of distance(x)^2)], the farer D(x) is more likly to be chosen
            probability = distance**2 / np.sum(distance**2) #probabiltiy
            mask = np.random.choice(height*width, p=probability) #randomly choose one pixel based on the probability
            centers[c] = image[mask] #assign next center

    elif init_type == 'random':
        #random(uniform)
        centers = np.zeros((k,nDimensions)) #initialize k cluseters' center
        mask = np.random.choice(height*width,size=k,replace=False) #randomly(uniform) choose k clusters' center's index without replacement
        for c in range(k):
            centers[c] = image[mask[c]]

    #assign each pixel's label to the nearest cluster's center
    distance = np.zeros((k,height*width))
    for c in range(k): #for every centers
        distance[c] = np.sqrt(np.sum((image-centers[c])**2,axis=1))#the distance between each data points and that particular center
    labels = np.argmin(distance,axis=0) #assign the nearest one
 return centers

#refer to https://github.com/algostatml/UNSUPERVISED-ML/blob/master/KMEANS%20AND%20KERNEL%20VERSION/KERNEL%20KMEANS/kernelkmeans.py
def Kmeans(image,k,init_type):
 #initialization
 images = [] #to store gif
 centers_diff = 1e+10
 height,width,nDimensions = int(np.sqrt(image.shape[0])), int(np.sqrt(image.shape[0])), image.shape[1]
 colormap = np.array([[255, 255, 255], [255, 255, 0], [255, 0, 255], [255, 0, 0], [0, 255, 255], [0, 255, 0], [0, 0, 255], [0, 0, 0]])#np.random.choice(range(256),size=(k,3)) #randomly pick an color, RGB
 centers0 = Initialization(image,k,init_type) #initialize centers

 while centers_diff >= 1e-10:
    #E step : classify all samples according to closet centers
    distance = np.zeros((k,height*width))
    for c in range(k): #for every centers
        distance[c] = np.sqrt(np.sum((image-centers0[c])**2,axis=1)) #compute the distance between every data points and the centers
    labels = np.argmin(distance,axis=0) #assign the pixel's label with the nearest cluster's center

    #M step : re-compute the centers which are the mean of that cluster
    centers1 = np.zeros((k,nDimensions))
    for c in range(k): #for every centers
        mask = np.argwhere(labels==c).reshape(-1) #find which pixel belongs to cluster c
        centers1[c] = np.average(image[mask],axis=0) #compute new centers

    centers_diff = np.sqrt(np.sum((centers1-centers0)**2))
    centers0 = centers1

    #use labels to do visualization
    image_cur = np.zeros((height,width,3)) #RGB
    for h in range(height):
        for w in range(width):
            image_cur[w,h] = colormap[labels[height*h+w]] #not [h,w], because labels is from
the top to bottom and then from left to right, not from left to right and then from the top to
bottom
     images.append(image_cur.astype('uint8')) #store this time's image
     #plt.imshow(image_cur.astype('uint8'))
#https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html
     #plt.pause(0.1)
 return images,labels
```

Spectral Learning

[part 1]

The spectral learning's algorithm involves some steps.

First, compute the similarity matrix W by the kernel function defined in the kernelkmeans.

Second, compute the degree matrix D based on the formula in PPT p.28,
$D_{v,u} = d_v \delta_{vu}, d_v = \sum_{u \in V} W_{vu}, \delta_{vu} = 1 \text{ if } v = u, \delta_{vu} = 0 \text{ if } v \neq u.$

Third, compute the graph Laplacian matrix L based on the formula in PPT p.38, $L = D - W$.

Forth, compute the eigenspace based on the part 3's procedure.

Finally, call the Kmeans fucntion to do the Kmeans clustering with each row of the eigenspace matrix (U for ratio cut, T for normalized cut) that is the coordinate of data points in the eigen space and record the clustering result to visualize clustering procedure.

[part 3]

Ratio cut is based on the algorithm in PPT p.56. It computes the first k eigenvectors with smallest non-zero eigenvalues of unnormalized graph Laplacian matrix L to construct matrix U. the row of U means the coordinates in the eigen-space, the column of U means the eigenvector.

Normal cut version is based on the algorithm in PPT p.73. It computes the normalized graph Laplacian matrix as L_{sym} based on the formula $L_{sym} = D^{-1/2} L D^{-1/2}$in PPT p.71,p.72. Then it computes the first k eigenvectors with smallest non-zero eigenvalues of normalized graph Laplacian matrix L_{sym} to construct matrix U, where the row of U means the coordinates in the eigen-space, the column of U means the eigenvector. Finally, normalized U as T based on the formula $t_{ij} = u_{ij} / (\sum_k u_{ik}^2)^{1/2}$ in PPT p.73

[part 4]

Project the labelized data points from the space spanned by the eigen-vector to the 2D-plane. Here I use the first two dimension of the eigenvector to do the projection.

```
import matplotlib.pyplot as plt
import numpy as np
import os
import matplotlib.image as mpimg
from scipy.spatial.distance import pdist,cdist,squareform
from array2gif import write_gif
from HW06_KernelKmeans import Kernel #self-defined
from HW06_Kmeans import Kmeans #self-defined

def plot_eigen(k, U, labels,result_filepath):
 colormap = ['red', 'orange', 'yellow', 'green', 'blue', 'purple', 'black', 'gray']
 for c in range(k): #for every cluster
     #actually you can choose any other two dimension to see how it looks like for different
angle
     plt.scatter(U[labels == c, 0], #the first dimension of coordinate of data points in c
cluster
```

```
                U[labels == c, 1], #the second dimension of coordinate of data points in c
cluster
                c=colormap[c], s=1)
 plt.savefig(result_filepath)
 plt.show()

def compute_eigen(init_cut_type,image_filename,k,L):

 file_path_eigenvalue =
'./HW06_SpectralLearning_{}_{}Cut_eigenvalues.npy'.format(image_filename,init_cut_type)
 file_path_eigenvector =
'./HW06_SpectralLearning_{}_{}Cut_eigenvectors.npy'.format(image_filename,init_cut_type)

 if os.path.isfile(file_path_eigenvalue): # if the pre-computed eigen file exist
     eigenvalues = np.load(file_path_eigenvalue)
     eigenvectors = np.load(file_path_eigenvector)
 else:
     eigenvalues, eigenvectors = np.linalg.eig(L) #compute the eigens, slow; the column
eigenvectors[:,i] is the eigenvector corresponding to the eigenvalue eigenvalues[i]; the
eigenvalues are not necessarily ordered
     np.save(file_path_eigenvalue,eigenvalues) #store it to avoid compute the eigenvalues
slowly again
     np.save(file_path_eigenvector,eigenvectors) #store it to avoid compute the eigenvectors
slowly again

 #eigen decomposition may return a complex number with small imaginary part, so here turns it
to the real number; refer to https://stackoverflow.com/questions/60366008/numpy-always-gets-
complex-eigenvalues-and-wrong-eigenvectors
 if np.all(eigenvalues.imag < 1e-10): #if the imaginary part is close to 0
     eigenvalues = eigenvalues.real #np.real_if_close(eigenvalues)
     eigenvectors = eigenvectors.real #np.real_if_close(eigenvectors)

 k_smallest_nonzero_eigenvalues_index = np.argsort(eigenvalues)[1:1+k] #k smallest "non-zero"
eigenvalue
 U = eigenvectors[:,k_smallest_nonzero_eigenvalues_index] #refer to PPT p.53
 return U

if __name__=='__main__':

 image_filenames = ['image1','image2']
 k = 5
 init_kmeans_type = 'kpp'
 init_cut_types = ['unnormalized','normalized'] #relaxing Ncut leads to normalized spectral
clustering, while relaxing RatioCut leads to unnormalized spectral clustering , refer to A
Tutorial on spectral clustering,
https://people.csail.mit.edu/dsontag/courses/ml14/notes/Luxburg07_tutorial_spectral_clustering.
pdf

 for image_filename in image_filenames: #for every images
     #load file and set initialization
     image = mpimg.imread('./data/{}.png'.format(image_filename)) #=>(100,100,3)
     image = (image * 255).astype('uint8') #convert float32(0~1) to uint8(0~255)
     height,width,colors = image.shape
     image = image.reshape(height*width,colors) #flatten ; =>(10000,3); image[h][w] =>
image[height*h+w]
     W = Kernel(image) #similarity matrix
     D = np.diag(np.sum(W,axis=1)) #degree matrix, refer to PPT p.28
     L = D - W #graph Laplacian, refer to PPT p.38
```

```
    #part 3 : try different ways to do initialization of spectral clustering eg. ratio cut, normalized cut
    for init_cut_type in init_cut_types:
        #======unnormalized(ratio) cut version, refer to PPT p.56,66======#
        if init_cut_type == 'unnormalized':
            U = compute_eigen(init_cut_type,image_filename,k,L) #U contains the first k eigenvectors of L, refer to PPT p.65,66 (H there is U here)
            images , labels = Kmeans(U,k,init_kmeans_type)

        #======normalized cut version, refer to PPT p.73======#
        elif init_cut_type == 'normalized':
            Lsym = np.diag(np.diag(D)**(-1/2)) @ L @ np.diag(np.diag(D)**(-1/2)) #normalized Laplacian Lsym = D^{-1/2}LD^{-1/2} = I-D^{-1/2}WD^{-1/2} refer to PPT p.71,72
            U = compute_eigen(init_cut_type,image_filename,k,Lsym) #U contains the first k eigenvectors of Lsym, refer to PPT p.70,73
            T = U / np.sqrt(np.sum(np.square(U),axis=1)).reshape(-1,1) # refer to PPT p.73
            images , labels = Kmeans(T,k,init_kmeans_type)

        #part 1 : show the clustering procedure of spectral clustering
        result_filepath = './result/SpectralLearning_{}_{}_{}clusters_{}'.format(image_filename,init_cut_type,k,init_kmeans_type)
        write_gif(images, '{}.gif'.format(result_filepath)) #print the process by gif
        print('{} converge!'.format(result_filepath))

        #part 4: examine whether the data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian or not
        plot_eigen(k, U, labels, '{}.png'.format(result_filepath))
```

b. experiments settings and results (20%) & discussion (30%)

All the experiment results is in the folder "result", and the gif's name represent different algorithm, number of clusters, inital methods.

Part1 (5%) & (5%)

[results] The gif name contains KernelKmeans or Spectral Learning. The Kernel's default parameters, γ_s and γ_c, are set as 0.001.

[discussion]

The convergence time in spectral learning is much faster than kernel kmeans.

Part2 (5%) & (5%)

[results] The number of clusters are 2,3,4, and 5 in Kernel Kmeans, so the gif name contains 2clusters, 3clusters, 4clusters, 5clusters. Default number of cluster is 5 in Spectral Learning, , so the gif name contains 5clusters.

[discussion] When the number of clusters is small, it will be classified according to the one with the greatest color difference. When the number of clusters starts to increase, we will start to classify the ones with smaller color differences.

Part3 (5%) & (10%)

[results] The initial method contains kmeans++ and random in KernelKmeans, ratio cut and normalized cut in Spectral Learning, so the gif name contains kpp, random, unnormalized, and normalized respectively. Default Kmeans' inital method is kpp in Spectral Learning.

> > **[discussion]** The initialization method of kmeans++ separates the different center points as far apart as possible, so at the beginning of the classification is very obvious that different colors of points are in different places, and there is be no color mixing place. The initialization method of random picks the centers randomly, so the classification at the beginning is sometimes good, sometimes bad.
>
> **Part4 (5%) & (10%)**
>
> > **[results]** The distribution of different cluster in eigenspace stores as png picture, and the name is the same as corresponding gif animation, eg. The corresponding png picture of "SpectralLearning_image2_unnormalized_5clusters_kpp.gif" is "SpectralLearning_image2_unnormalized_5clusters_kpp.png".
> >
> > **[discussion]** The picture shows that different cluster filled with different colors does distribute in different place in eigenspace. There is no color-mixed areas. Both ratio cut and normalized cut in image1's projected pictures in eigenspace consists of line segments and looks similar, but in image2 ratio cut and normalized cut are not alike because normalized cut contains the outlier in the bottom right corner.

c. observations and discussion (10%)

> 1. Though spectral learning needs lots of mathematical proof to understand, it's easier to implement (just need to do the matrix computation) than Kmeans. Also, spectral learning runs faster than Kmeans.
> 2. When the number of clusters increase, the execute time also increase.
> 3. The initialization method of Kmeans++ doesn't guarantee the faster execute time than random, sometimes it is slower than random initialization method.