Report

- Code with Detailed Explanation
  - Part 1: Clustering procedure

Read image and normalize features by minmax such that the position and the color lead to equal effects under the same γ. In addition, to avoid too small numbers when computing gram matrix, I scaled the features by 100.

```
import cv2
def imread(filename):
    img = cv2.imread(filename) # B, G, R
    data = np.zeros((HEIGHT*WIDTH,5)) # attribute: r, c, b, g, r
    for i in range(HEIGHT):
        for j in range(WIDTH):
            b, g, r = img[i, j]
            data[i*HEIGHT+j] = [i, j, b, g, r]
    data = ((data - data.min(axis=0))/(data.max(axis=0)-data.min(axis=0)))*100
    return data
```

Gram matrix

Calculate gram matrix by the kernel function below, and vectorize it by numpy.

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

```
from scipy.spatial.distance import cdist
def kernel(x, gamma_s, gamma_c):
    s, c = x[:, :2], x[:, 2:]
    return np.exp(-gamma_s*cdist(s, s,'sqeuclidean')-gamma_c*cdist(c, c, 'sqeuclidean'))
```

Spectral clustering for Ratio-Cut (unnormalized)

1. Use gram matrix as our weighted adjacency matrix W
2. Compute the unnormalized graph Laplacian by W and degree matrix D

   Degree matrix D: $d_{ii} = \sum_{j=1}^{n} w_{ij}$ (diagonal matrix)

   Unnormalized graph Laplacian: $L = D - W$
3. Use numpy.linalg.eigh (eigendecomposition $Lu = \lambda u$) to calculate eigenvalues and eigenvectors.

   Besides, since L is a Hermitian matrix, we can use eigh instead of eig for faster computing. In addition, the returned eigenvalues and eigenvectors are in ascending order, we don't need to sort it.

   Let $U \in R^{n \times k}$ be the matrix containing the first k eigenvectors which eigenvalues are not zeros. (drop the eigenvectors w/ eigenvalues approximate zeros in practice)
4. View $U = [u_1, u_2, \ldots, u_k]$ as $Y = [y_1, y_2, \ldots, y_n]^\top$, each y_i is the data point represented in the eigenspace.
5. Specify input data ($Y = U$) and number of clusters (k), and then run KMeans to obtain the clustering result.

(Code is at next page)

Spectral clustering for N-Cut (normalized)

The main difference b/w unnormalized and normalized spectral clustering is the way to form the eigenspace.

Instead of $Lu = \lambda u$, we use $Lu = \lambda Du$ to obtain the eigenvectors, and the rest procedures are the same. However, since numpy doesn't have generalized eigendecomposition method, we can use $L_{rw} = D^{-1}L$ or $L_{sym} = D^{-\frac{1}{2}}LD^{-\frac{1}{2}}$ instead.

For my version of normalized SC, I tried calculating L_{rw} only, but numpy.linalg.eigh gave an incorrect eigenvalues and eigenvectors; thus, I used L_{sym} and formed T by normalizing each row of U to 1.
Likewise, finally plugged in T to KMeans and got the clustering results.

```
import numpy as np
import numpy.linalg as LA
def SpectralClustering(S, n_clusters, method, init, path):
    '''
    S: Similarity matrix (n,n)
    n_cluster: k
    '''
    # 1. Construct a similarity graph, Let W be its weighted adjacency matrix
    W = S
    # 2. Compute the unnormalized Laplacian
    # Degree matrix
    D = W.sum(axis=1) * np.eye(W.shape[0])
    # Unnormalized Graph Laplacian
    L = D - W
    # 3. Compute the first k eigenvectors u1~uk of L, and
    #    let U = (n,k) be the matrix containing the vectors u1~uk as columns
    # 4. For i = 1~n, let yi = (k,) be the vector corresponding to the i-th row of U
    U = np.zeros((10000,3))

    if method == 'normalized':
        D_inv_sqrt = np.diag(1/np.diag(np.sqrt(D)))
        Lsym = D_inv_sqrt @ L @ D_inv_sqrt
        eigenvalues, eigenvectors = LA.eigh(Lsym)

        for i, ev in enumerate(eigenvalues):
            if ev < 1e-10: continue
            U = eigenvectors[:,np.arange(i, i+n_clusters)]
            break
        # 5. Form the matrix T = (n.k) from U by normalizing the rows to norm 1
        T = U/np.sqrt(np.sum(np.square(U),axis=1)).reshape(-1,1)
        # 6. Cluster the points (yi)i=1~n in (k,) with the kmeans algorithm into clusters C1~Ck
        labels = KMeans(T, n_clusters, init, path)
        plot_eigenspace(T, labels, "{}_{}_{}_{}.png".format(path.split('/')[1], method, n_clusters, init))
    else:
        eigenvalues, eigenvectors = LA.eigh(L)
        for i, ev in enumerate(eigenvalues):
            if ev < 1e-10: continue
            U = eigenvectors[:,np.arange(i, i+n_clusters)]
            break
        # 5. Cluster the points (yi)i=1~n in (k,) with the kmeans algorithm into clusters C1~Ck
        labels = KMeans(U, n_clusters, init, path)
        plot_eigenspace(U, labels, "{}_{}_{}_{}.png".format(path.split('/')[1], method, n_clusters, init))

    return labels
```

K-means

Input: data points, number of clusters, init method
Initialize centroids by several methods will be discussed later
E step: calculating the distances from each data point x_i to the centroids, and determine the nearest cluster that x_i belongs to.
M step: updating the centroids by taking average in each cluster.
Until reaching maximal iteration or centroids converges.
Output: labels

```
def KMeans(X, n_clusters, init, path):
    '''
    X: (# data, # features)
    n_clusters: k
    init: the method to initialize k means
    '''
    max_iter = 100
    tol = 1e-6

    n_data, n_features = X.shape
    labels = np.zeros((n_data,)) # class: ()
    centroids = init_centroids(X, n_clusters, init) # (n_data,n_features)
    frames = []

    for _iter in range(max_iter):
        # E step: Calculate distances and determine the nearest cluster that x belongs to
        labels_new = np.argmin(LA.norm(X-centroids[:,np.newaxis], axis=2), axis=0)
        frames.append(visualize(labels_new, "{}_{}".format(path, _iter)))
        # M step: update centroids
        centroids_new = np.empty((n_clusters,n_features))
        for k in range(n_clusters):
            cluster_k_members = np.where(labels_new == k)[0]
            centroids_new[k] = X[cluster_k_members,:].sum(axis=0)/len(cluster_k_members)

        if LA.norm(centroids_new-centroids) < tol:
            gen_gif(path, frames)
            break
        labels = labels_new
        centroids = centroids_new

    return labels
```

Kernel k-means

Input: precomputed gram matrix, number of clusters, init method.
Initialize R where $\gamma_{ik} = 1$ if the data point x_i belongs to k, otherwise $\gamma_{ij} = 0$ for $j \neq k$.
E step: calculating distances b/w data points and centroids using kernel trick (will be discussed later)
M step: update R, i.e. reassign each data point to its nearest centroid.
Until reaching maximal iteration or the distance matrix D converges.
Output: labels

```python
def KernelKMeans(G, n_clusters, init, path):
    '''
    Parameters
    ----------
    G: Gram matrix (10000,10000)
    n_clusters
    init: the method to initialize k means

    Returns
    -------
    ndarray, shape=(10000,), value=kth cluster
    '''
    max_iter = 100
    tol = 1e-6

    n_data = G.shape[0]
    D = np.full((n_data, n_clusters), np.inf)
    R = initR(G, n_clusters, init)
    frames = []

    for _iter in range(max_iter):
        frames.append(visualize(R.argmax(axis=1), "{}_{}".format(path, _iter)))
        # E step: calculate distance b/w points and centroids using kernel trick
        D_new = kernel_trick(G, R)
        # M step: update R (Class matrix), reassign each points to its nearest centroid
        R_new = np.zeros(R.shape)
        R_new[np.arange(n_data), np.argmin(D_new, axis=1)] = 1

        # Converge check
        if LA.norm(D_new-D) < tol:
            frames.append(visualize(R.argmax(axis=1), "{}_{}".format(path, _iter+1)))
            gen_gif(path, frames)
            break
        D = D_new
        R = R_new

    return R.argmax(axis=1)
```

Kernel Tricks

Element-wise:

$$||\phi(x_i) - \mu_k||^2 = \phi(x_i)^\top \phi(x_i) - 2\phi(x_i)^\top \mu_k + \mu_k^\top \mu_k \text{ with } \mu_k = \frac{\sum_{x_i \in C_k} \phi(x_i)}{|C_k|} = \frac{\sum_{i=1}^n \gamma_{ik} \phi(x_i)}{n_k}$$

We have

$$\begin{aligned} ||\phi(x_i) - \mu_k||^2 &= \phi(x_i)^\top \phi(x_i) - \frac{2}{n_k} \sum_{j=1}^n \gamma_{jk} \phi(x_i)^\top \phi(x_j) + \frac{1}{n_k^2} \sum_{j=1}^n \sum_{l=1}^n \gamma_{jk}\gamma_{lk} \phi(x_j)^\top \phi(x_l) \\ &= \kappa(x_i, x_i) - \frac{2}{n_k} \sum_{j=1}^n \gamma_{jk} \kappa(x_i, x_j) + \frac{1}{n_k^2} \sum_{j=1}^n \sum_{l=1}^n \gamma_{jk}\gamma_{lk} \kappa(x_j, x_l) \end{aligned}$$

Let's vectorize it into matrix form

Matrix-wise:

$$\text{diag}(\mathbf{G})\mathbf{1}_{n\times k} - \frac{2}{n_k}\mathbf{GR} + \frac{1}{n_k^2}\mathbf{1}_{n\times k}\,\text{diag}(\mathbf{R}^\top\mathbf{GR})$$

where **G** is gram matrix, **R** is cluster matrix that defined in kernel k-means,

$\mathbf{1}_{n\times k}$ = np.ones((n,k)): all elements are 1

```
def kernel_trick(G, R):
    '''
    Parameters
    ----------
    G: Gram matrix (10000,10000)
    R: label matrix (10000, n_clusters)
       Rik = 1 if xi belongs to kth cluster

    Returns
    -------
    D: Distance matrix (10000, n_clusters)
    '''
    n_data = G.shape[0]
    n_clusters = R.shape[1]

    Rk = R.sum(axis=0)

    D = np.matmul(G*np.eye(n_data), np.ones((n_data, n_clusters))) - 2*(np.matmul(G, R)/Rk) + \
        (np.matmul(np.ones((n_data, n_clusters)), np.matmul(np.matmul(R.T, G), R)*np.eye(n_clusters)))/(Rk**2)

    return D
```

Visualization & make videos

After obtaining the labels, using opencv to generate images and imageio to generate gif

```
def visualize(X, filename):
    length = int(np.sqrt(X.shape[0]))
    r = [107,140,231,214]
    g = [110,162,186, 97]
    b = [207, 82, 82,107]

    img = np.zeros([length,length,3], dtype=np.uint8)

    vfunc_r = np.vectorize(lambda k: r[int(k)])
    vfunc_g = np.vectorize(lambda k: g[int(k)])
    vfunc_b = np.vectorize(lambda k: b[int(k)])

    img[:,:,0] = vfunc_r(X).reshape(length, -1)
    img[:,:,1] = vfunc_g(X).reshape(length, -1)
    img[:,:,2] = vfunc_b(X).reshape(length, -1)
    cv2.imwrite(filename + '.png', cv2.cvtColor(img,cv2.COLOR_RGB2BGR))
    return img

def gen_gif(path, frames):
    folders = path.split('/')
    filename = os.path.join('media', folders[1]+'_videos', folders[2])
    imageio.mimsave(filename+'.gif', frames)
```

- Part 2: Different number of clusters

  Parse arguments and pass the number of clusters into specific clustering algorithm.

```
def argparser(argv):
    try:
        opts, args = getopt.getopt(argv, "hi:k:path:method:init:gamma_s:gamma_c", ["infile=", "k=",
                                                    "method=", "init=", "gamma_s=", "gamma_c="])
    except getopt.GetoptError:
        print('AS6.py -i <infile> ...')
        sys.exit(2)

    infile, k, path = '', 2, ''
    method, init, gamma_s, gamma_c = '', '', '', ''
    for opt, arg in opts:
        if opt == '-i': infile = arg
        elif opt == '-k': k = int(arg)
        elif opt == '--method': method = arg
        elif opt == '--init': init = arg
        elif opt == '--gamma_s': gamma_s = float(arg)
        elif opt == '--gamma_c': gamma_c = float(arg)
    return infile, k, method, init, gamma_s, gamma_c
```

```
def main(argv):

    infile, k, method, init, gamma_s, gamma_c = argparser(argv)

    # Load image and flatten it into data
    data = imread(infile)

    outfile = os.path.join('media', infile.split('.')[0], '_'.join([method, str(k), init,
                                                                    ''.join(str(gamma_s).split('.')),
                                                                    ''.join(str(gamma_c).split('.'))]))
    G = kernel(data, gamma_s, gamma_c)
    if method == 'NSC': # spectral clustering
        prediction = SpectralClustering(G, k, 'normalized', init, outfile)
    elif method == 'UNSC':
        prediction = SpectralClustering(G, k, 'unnormalized', init, outfile)
    elif method == 'kkmeans':
        prediction = KernelKMeans(G, k, init, outfile)
```

- Part 3: Different initializations
  Assume K = number of clusters
  K-means
  K-means++
  1. Choose one centroid uniformly at random among the data points
  2. For each data point x not chosen yet, compute D(x): the distance b/w x and the nearest centroid that has already been chosen.
  3. Choose one new data point at random following the probability distribution of $D(x)^2$ as a new centroid.
  4. Repeat 2 and 3 until K centroids have been chosen

  Random
  Calculate the mean and std of the given data points, and use the mean and std to randomly choose K centroids. (follows normal distribution)

```
def init_centroids(X, n_clusters, init):
    '''
    X: (# data, # features)
    '''
    n_data, n_features = X.shape # n: # data points, p: # features
    centroids = np.zeros((n_clusters, n_features))

    if init == 'kmeans++':
        # 1. Choose one center uniformly at random among the data points.
        centroids[0] = X[np.random.randint(0, n_data),:]

        # 4. Repeat Steps 2 and 3 until k centers have been chosen.
        for k in range(1, n_clusters):

            # 2. For each data point x not chosen yet, compute D(x),
            #    the distance between x and the nearest center that has already been chosen.
            D = np.min(LA.norm(X-centroids[:k][:, np.newaxis], axis=2), axis=0)

            # 3. Choose one new data point at random as a new center,
            #    using a weighted probability distribution where a point x
            #    is chosen with probability proportional to D(x)^2.
            prob_distr = (D**2)/(D**2).sum()
            centroids[k] = X[np.random.choice(np.arange(n_data), p=prob_distr),:]
        # Now that the initial centers have been chosen, proceed using standard k-means clustering.

    elif init == 'random':
        X_mean = X.mean(axis=0)
        X_std  = X.std(axis=0)
        centroids = np.random.normal(loc=X_mean, scale=X_std, size=(n_clusters, n_features))

    return centroids
```

Kernel k-means

For kernel k-means, we initialize R instead of centroids.

K-means++

Use gram matrix to compute the distances from $\phi(x_i)$ to $\phi(x_j)$ for all data points. (all_D)

$$||\phi(x_i) - \phi(x_j)||^2 = \phi(x_i)^\top \phi(x_i) + 2\phi(x_i)^\top \phi(x_j) + \phi(x_j)^\top \phi(x_j)$$
$$= \kappa(x_i, x_i) + 2\kappa(x_i, x_j) + \kappa(x_j, x_j)$$

1. Choose one centroid uniformly at random among the data indices
2. For each data index i not chosen yet, compute D(i): the distance b/w i and the nearest centroid that has already been chosen.
3. Choose one new data point at random following the probability distribution of $D(i)^2$ as a new centroid.
4. Repeat 2 and 3 until k centroids have been chosen
5. Form R by assigning each data points to K clusters by specifying $\gamma_{ik} = 1$.

Random

Randomly assign each data points to one of the K clusters by specifying $\gamma_{ik} = 1$.

```
def initR(G, n_clusters, init):

    n_data = G.shape[0]
    R = np.zeros((n_data, n_clusters))
    if init == 'kmeans++':

        data_indices = np.arange(n_data) # record remaining data indices
        centroids = np.zeros((n_clusters,), dtype=np.ulonglong) # record data indices that are chosen to be centroids

        # compute distances from phi_i to phi_j
        all_D = G.diagonal()[:,np.newaxis] -2*G + G.diagonal()[np.newaxis,:]
        # 1. Choose one center uniformly at random among the data points.
        centroids[0] = np.random.randint(0, n_data)

        # 4. Repeat Steps 2 and 3 until k centers have been chosen.
        for k in range(1, n_clusters):

            # 2. For each data point x not chosen yet, compute D(x),
            #    the distance between x and the nearest center that has already been chosen.
            D = np.min(all_D[centroids[:k],:], axis=0)

            # 3. Choose one new data point at random as a new center,
            #    using a weighted probability distribution where a point x
            #    is chosen with probability proportional to D(x)^2.
            prob_distr = (D**2)/(D**2).sum()
            centroids[k] = np.random.choice(np.arange(n_data), p=prob_distr)

        R[np.arange(n_data), np.argmin(all_D[centroids,:], axis=0)] = 1

    elif init == 'random':
        R[np.arange(n_data), np.random.randint(n_clusters,size=n_data)] = 1

    return R
```

- Part 4: Whether data in the same cluster have the same coord. in the eigenspace of Graph Laplacian

  Plot the eigenspace by plotting data points in each eigenvector coordinate, and also sort the data points by labels(which cluster they belong to).

```
def plot_eigenspace(eigenvectors, labels, filename):
    '''
    eigenvectors: (n_data, n_coordinates)
    labels: clusters that the data points belong to
    '''
    n_coord = eigenvectors.shape[1]
    fig, axes = plt.subplots(n_coord, 1, figsize=(10, 5*n_coord))
    for i in range(n_coord):
        plot(axes[i], eigenvectors.T[i, labels.argsort()], np.sort(labels), "Eigenvector "+str(i+1))
    fig.savefig(os.path.join('media','eigenspace',filename+'.png'),
                format="png", dpi=600, bbox_inches="tight")


def plot(ax, eigenvector, labels, title):
    ax.set_title(title)
    f = lambda x: colormap[x]
    x = np.arange(labels.shape[0])
    ax.scatter(x, eigenvector, color=list(map(f, labels)), marker='.')
```

- Experiments Settings and Results & Discussions
  - Part 1 & Part2 & Part 3:

    Clustering procedure w/ K=2,3,4, and init=kmeans++, random

    Unnormalized spectral clustering (R-cut)

    tol=1e−6, max_iter=100

K=2, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=2

K=3, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=3

K=4, init=kmeans++
γ_s=5e−2, γ_c=1e−3
iterations=10

K=3, init=random
γ_s=1e−2, γ_c=1e−2
iterations=4

K=2, init=random
γ_s=1e−2, γ_c=1e−2
iterations=2

K=3, init=random
γ_s=5e−2, γ_c=1e−3
iterations=11

K=4, init=random
γ_s=5e−2, γ_c=1e−3
iterations=13

For unnormalized spectral clustering, since its graph Laplacian L is unnormalized, when doing the eigendecomposition, we will find that the corresponding eigenvalues and eigenvectors may be vulnerable/sensitive to the small numbers in L (i.e. the scale of the values in L); as a matter of fact, if L does not approximate sparse matrix, it may lead to incorrect eigenvalues and eigenvectors in practice.

As my observation of unnormalized spectral clustering (R-cut), it was more sensitive to how my data were scaled or what parameters γ_s and γ_c I set. If setting $\gamma_s > \gamma_c$, it will be easier for R-cut to cluster the data, since the position information is much more pure than the color information(color information has more noises). (Part1 & other observations)

As the clustering results of image1 I showed above, if I set γ_s and γ_c both equal to 1e−2, for K=4, it cannot cluster the data well to 4 clusters; yet if I set γ_s=5e−2 and γ_c=1e−3 (position weighs more than color), it can cluster them to 4 clusters.

K=2, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=2

K=3, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=4

K=4, init=kmeans++
γ_s=5e−2, γ_c=1e−3
iterations=11

K=2, init=random
γ_s=1e−2, γ_c=1e−2
iterations=6

K=3, init=random
γ_s=1e−2, γ_c=1e−2
iterations=6

K=4, init=random
γ_s=5e−2, γ_c=1e−3
iterations=12

Furthermore, we may required to tune different parameters for different K. From my experiments, I found γ_s and γ_c both equal to 1e−2 will be suitable for K=2 and 3, but unsuitable for K=4. This problem is that the variances of the first K eigenvectors computed by unnormalized graph Laplacian are small, k-means thought the corresponding representations of data in eigenspace are alike. Besides, the maximal iterations increases as K increases. (Part2)

In addition, for different init methods: kmeans++ and random, I found that it may be way faster to converge for kmeans++ than random. I speculated that it is because the initial centroids that kmeans++ generated are closer to its final centroids. Besides, kmeans++ will be more likely to cluster the data well in some cases—such as the case

of K=3, γ_s and γ_c both equal to 1e−2: in this case, kmeans++ can cluster data clearly into 3 clusters, but random cannot under this parameter setting. (Part3)

Normalized spectral clustering (N-cut)

tol=1e−6, max_iter=100

K=2, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=2

K=3, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=2

K=4, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=5

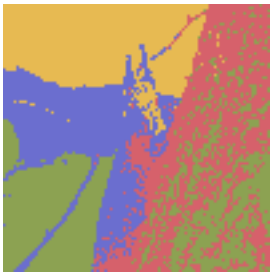
K=2, init=random
γ_s=1e−2, γ_c=1e−2
iterations=3

K=3, init=random
γ_s=1e−2, γ_c=1e−2
iterations=4

K=4, init=random
γ_s=1e−2, γ_c=1e−2
iterations=20

K=2, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=3

K=3, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=5



K=4, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=10

K=2, init=random
γ_s=1e−2, γ_c=1e−2
iterations=4

K=3, init=random
γ_s=1e−2, γ_c=1e−2
iterations=7

K=4, init=random
γ_s=1e−2, γ_c=1e−2
iterations=10

Compared with R-cut, we can set both γ_s and γ_c equal to 1e−2 for N-cut, and the results are all clustered well into clearly K clusters.

Besides, I first tried scaling the original data to [0,1], and tried to manually give

different settings of γ_s and γ_c, but I didn't find a suitable parameters γ_s and γ_c for R-cut; thereby, I scaled the data to [0,100], so that the values in graph Laplacian may not be too small and not sparse enough.

However, for N-cut, the scale will not be such an important issue as R-cut, since it normalizes the graph Laplacian. This will clearly be observed by their eigenspaces. If the results of R-cut is not well clustered, it may be due to the unnormalized graph Laplacian, since the bad values in unnormalized graph Laplacian may result in poor eigendecomposition.

Although N-cut took more time doing the calculation, the results performed way better than R-cut. In addition, N-cut is less vulnerable/sensitive to the parameter settings. (more generalized)—we can use parameters γ_s and γ_c both equal to 1e−2 across different K and different init method, which cannot be done by R-cut.

For more detailed explanation, if we use R-cut, we should be careful the scale of our original data and the scale of the gram matrix we calculated.

Furthermore, the maximal iterations increases as K increases. (Part1 & Part 2other observations)

Additionally, different init methods lead to different clustering results. It seems like kmeans++ is way better than normally random based on the maximal iterations. (Part3)

Kernel k-means

tol=1e−6, max_iter=100

Unlike the parameters I set above, I use γ_s and γ_c both equal to 1e−3 for kernel k-means for the sake of approximately similar results among different clustering algorithms.

K=2, init=kmeans++
γ_s=1e−3, γ_c=1e−3
iterations=14

K=3, init=kmeans++
γ_s=1e−3, γ_c=1e−3
iterations=15

K=4, init=kmeans++
γ_s=1e−3, γ_c=1e−3
iterations=23

K=2, init=random
γ_s=1e−3, γ_c=1e−3
iterations=19

K=3, init=random
γ_s=1e−3, γ_c=1e−3
iterations=26

K=4, init=random
γ_s=1e−3, γ_c=1e−3
iterations=31

K=2, init=kmeans++
γ_s=1e−3, γ_c=1e−3
iterations=10

K=3, init=kmeans++
γ_s=1e−3, γ_c=1e−3
iterations=70

K=4, init=kmeans++
γ_s=1e−3, γ_c=1e−3
iterations=20

K=2, init=random
γ_s=1e−3, γ_c=1e−3
iterations=21

K=3, init=random
γ_s=1e−3, γ_c=1e−3
iterations=24

K=4, init=random
γ_s=1e−3, γ_c=1e−3
iterations=52

Since I adjusted the parameters of γ_s and γ_c both equal to 1e−3 for kernel k-means, their results are similar to each other. Besides, since I did vectorization when computing the kernel trick, my kernel k-means is rather faster than spectral clustering—eigendecomposition is time-consuming in spectral clustering; yet, it obviously took more iterations for kernel k-means than spectral clustering.

In addition, if considering color to be the clustering criteria, kernel k-means clustered somewhat better than spectral clustering in image2 for K=2, yet worse than spectral clustering in image1 for K=2 (difference b/w kmeans++ and random in pictures above). (Part1 & other observations)

For different number of clusters K, it may need more iterations as K increases except for the case of (K=3, init=kmeans++, image2). (Part2)

Compared with spectral clustering, we can observe that the clustering results of kernel k-means are more vulnerable to the init methods. The results of spectral clustering are similar with same settings but different init methods, yet the results of kernel k-means look different with same settings but different init methods.

Besides, it's not really apparent that k-means++ always has fewer iterations than random. This may be because of the random method I used in kernel k-means is different form the random method I used in spectral clustering. I picked normally random based on $N(\mu_{data}, \sigma^2_{data})$, but completely uniformly random method in kernel k-means. Except for the case (K=3, init=kmeans++, image2), it may need more

iterations for random method rather than kmeans++. (Part3)

Summary

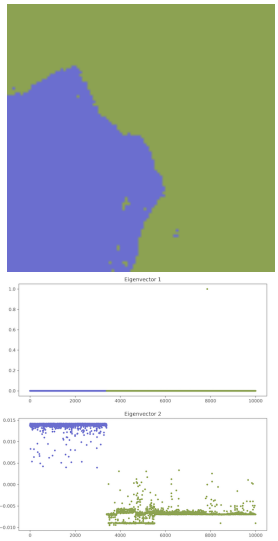
In my opinion, if I go for time-effectiveness, I will choose kernel k-means. Yet, if I desire to have results with less dependency to the scale of data, parameter settings, and different init method, I will go for normalized spectral clustering (N-cut).

Last but not the least, I may not use unnormalized spectral clustering (R-cut) due to its vulnerability to the scale of data, parameter settings, and different init method.

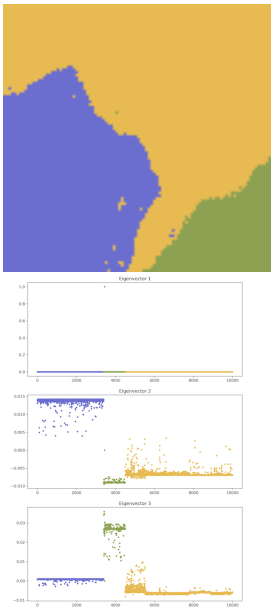
- Part 4: Whether data in the same cluster have the same coord. in the eigenspace of Graph Laplacian

Unnormalized spectral clustering (R-cut)

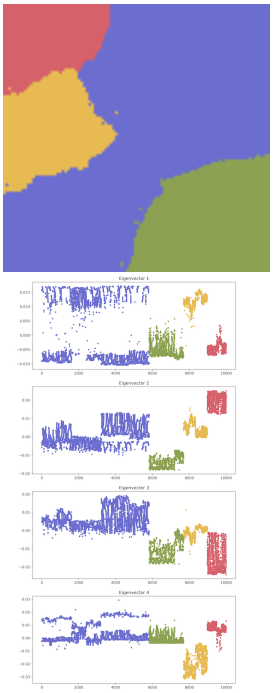
K=2, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=2



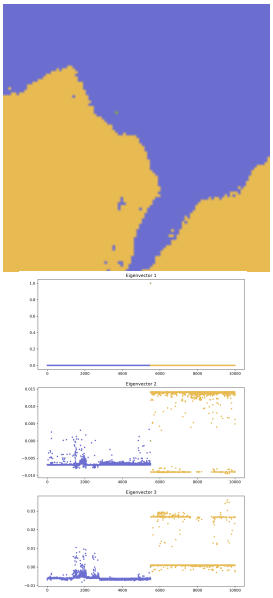
K=3, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=3



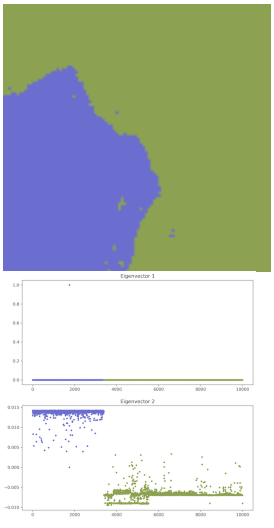
K=4, init=kmeans++
γ_s=5e−2, γ_c=1e−3
iterations=10



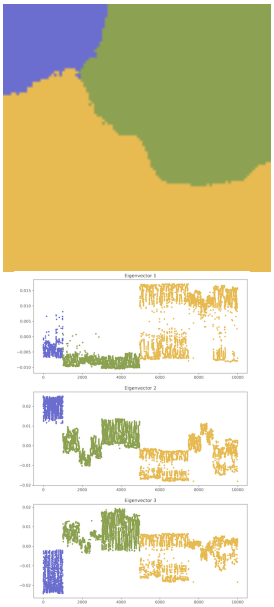
K=3, init=random
γ_s=1e−2, γ_c=1e−2
iterations=4



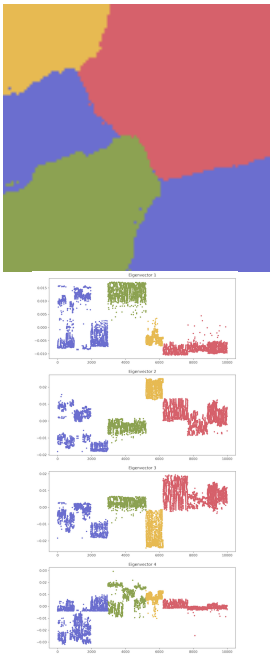
K=2, init=random
γ_s=1e−2, γ_c=1e−2
iterations=2



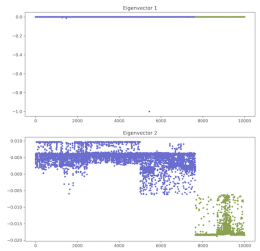
K=3, init=random
γ_s=5e−2, γ_c=1e−3
iterations=11



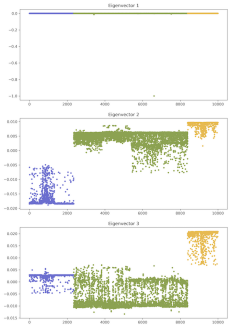
K=4, init=random
γ_s=5e−2, γ_c=1e−3
iterations=13



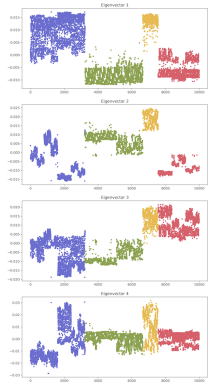
K=2, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=2



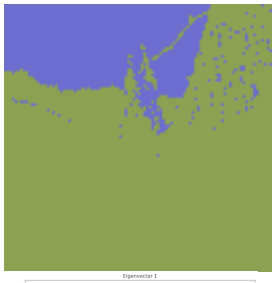
K=3, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=4

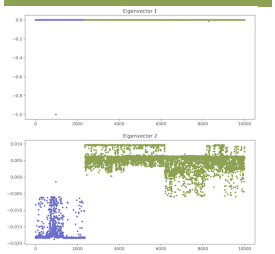


K=4, init=kmeans++
γ_s=5e−2, γ_c=1e−3
iterations=11

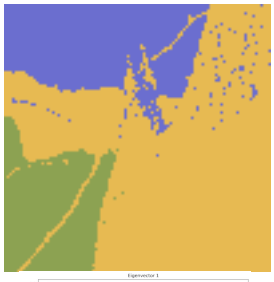


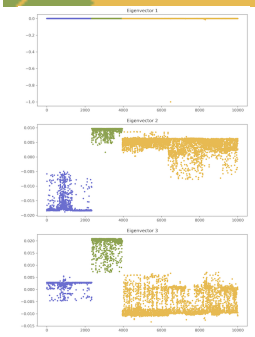
K=2, init=random
γ_s=1e−2, γ_c=1e−2
iterations=6



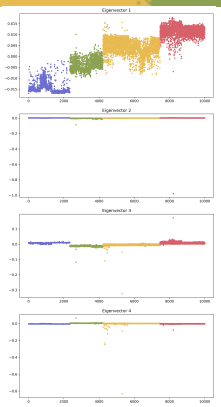


K=3, init=random
γ_s=1e−2, γ_c=1e−2
iterations=6





K=4, init=random
γ_s=5e−2, γ_c=1e−3
iterations=12



For unnormalized spectral clustering (R-cut) under my settings of parameters, some of the indicator vectors (first K eigenvectors) we choose may not split the data well.

For example, the first eigenvector in K=2 and 3 w/ kmeans++ in image1, their data represented in the first eigenvector approximate constant, yet the rest eigenvectors provides enough information for k-means to split. This means that the data in different clusters might represented alike in some coordinates in the eigenspace(the first eigenvector for K=2 and 3 w/ kmeans++ in image1)

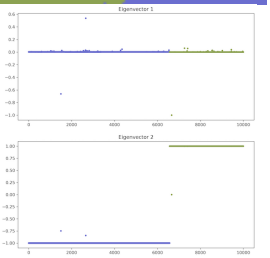
From general observation, the data in the same cluster have the same coordinates in eigenspace. Take the same example of K=2 and 3 in image1.

K=2 w/ kmeans++ in image1:
Take threshold ≈ 0.004 in eigenvector2, the above is cluster1, and the below is cluster2.
K=3 w/ kmeans++ in image1:
Take threshold ≈ 0.004 in eigenvector2, the above is cluster1, and the below is cluster2 and 3; take threshold ≈ 0.01 in eigenvector3, the above is cluster2, and the below is cluster1 and 3
⇒ cluster1 (e1, e2>0.004, e2<0.01)
cluster2 (e1, e2<0.004, e2>0.01)
Cluster3 (e1, e2<0.004, e2<0.01)

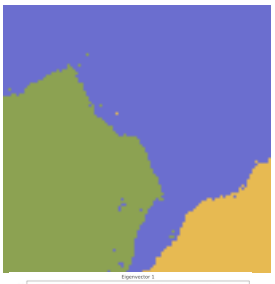
Actually it doesn't matter to use different init methods, their eigenspaces are the same, to mention different init methods is just because the cluster order may be different.

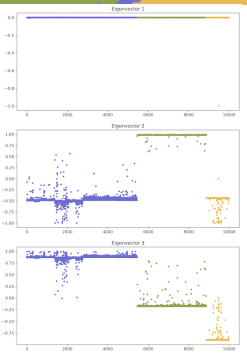
Normalized spectral clustering (N-cut)

K=2, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=2

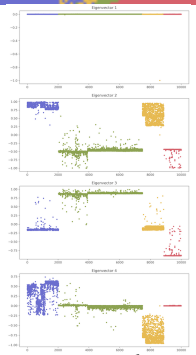


K=3, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=2

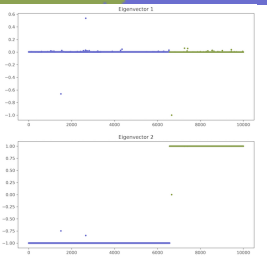




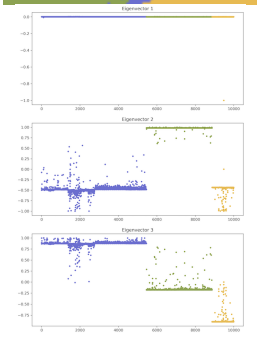
K=4, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=5



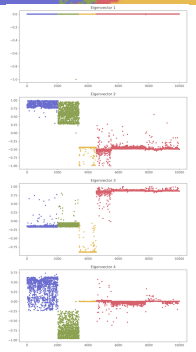
K=2, init=random
γ_s=1e−2, γ_c=1e−2
iterations=3



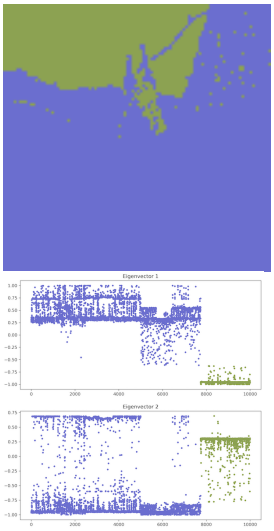
K=3, init=random
γ_s=1e−2, γ_c=1e−2
iterations=4



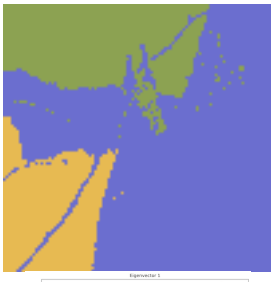
K=4, init=random
γ_s=1e−2, γ_c=1e−2
iterations=20

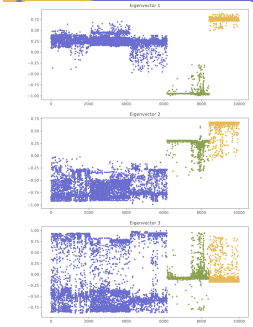


K=2, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=3

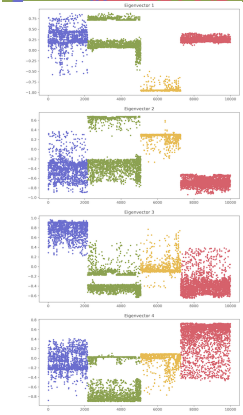


K=3, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=5

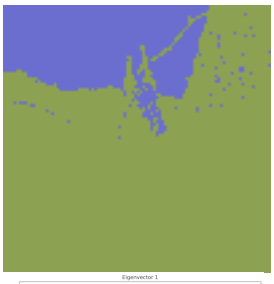


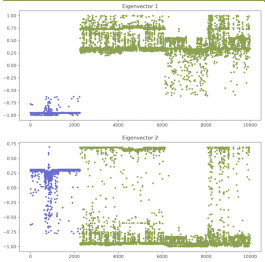


K=4, init=kmeans++
γ_s=1e−2, γ_c=1e−2
iterations=10

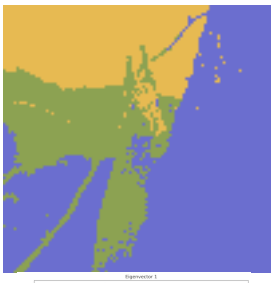


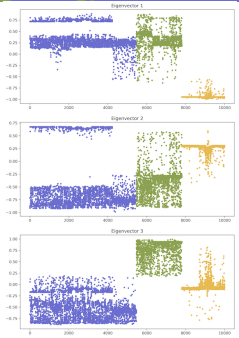
K=2, init=random
γ_s=1e−2, γ_c=1e−2
iterations=4



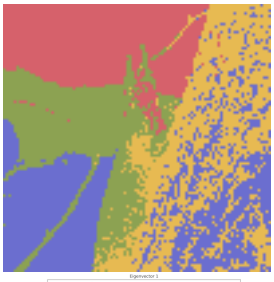


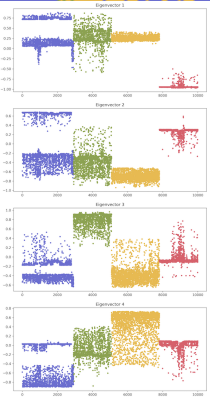
K=3, init=random
γ_s=1e−2, γ_c=1e−2
iterations=7





K=4, init=random
γ_s=1e−2, γ_c=1e−2
iterations=10





Compared w/ R-cut, the eigenvectors of normalized graph Laplacian are more evenly distributed. In addition, if setting γ_s and γ_c both equal to 1e−2, the eigenvectors in R-cut of K=4 will be constant such that the clustering results contain less than 4 clusters.

Generally, the data in the same clusters have the same coordinates in the eigenspace. For example, the case of K=2 w/ kmeans++ in image2, we can use the first eigenvector then we can have a good clustering result.

Nonetheless, considering the case of K=4 w/kmeans++ in image2, we cannot directly split data into clusters by our eyes according to the four eigenvectors. (e1, e3 provide information to split cluster1,2,4 from cluster3 and cluster2,3,4 form cluster1; yet, I found it hard to find a threshold for e2 and e4).

- Observations & discussions

  Some observations have been discussed above.

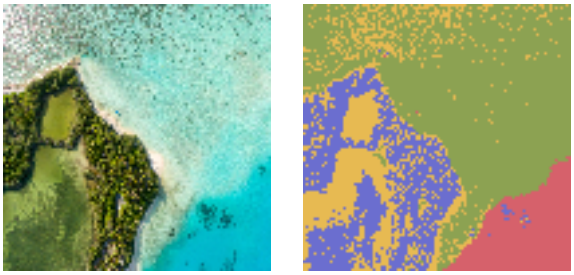
  Parameter settings.

  γ_s and γ_c correspond to position features and color features.

  If tuning γ_s much higher than γ_c, it's reasonable that the clustering results will be prone to affected by position.

Kernel k-means
K=4, init=kmeans++
γ_s=1e−3, γ_c=1e−5
iterations=39

On the other hand, if tuning γ_c higher than γ_s, the clustering will be tend to influenced by colors.



Kernel k-means
K=4, init=kmeans++
γ_s=1e−5, γ_c=1e−3
iterations=11

Performance comparison

Due to the eigendecomposition of high-dimensional matrix, spectral clustering is much slower than kernel k-means.

Efficiency: kernel k-means > R-cut > N-cut

Iterations: kernel k-means > R-cut ≈ N-cut

If the graph Laplacian does not approximate a sparse matrix, the eigendecomposition may fail. (incompleteness)