ML HW6 - 0856136 鍾昀諠

In the following 2 sections I will illustrate my implementation with code segments step by step.

Since there are a lot of images to be showed, I place all images in directory *GIF_RESULTS* instead of directly putting all of them in.

Naming Principle for each GIF explained in Appendix

Kernel K-means

1. **Extract data, normalize to 0~1 and transform to 10000X3 numpy array**

```
im1 = np.array(Image.open('./image1.png')).reshape((10000, 3)) / 255
im2 = np.array(Image.open('./image2.png')).reshape((10000, 3)) / 255
```

---

2. **Setup Hyper-parameter**

```
Gamma_s = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
Gamma_c = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
K = [2, 3, 4, 5]
```

Note that here I use a list for each hyper-parameter **for Grid Search** since there's no time left for me to study how to learn kernel.

---

3. **Calculate kernel, Perform Grid Search and save the result**

```
def kernel(x, y, gamma_s, gamma_c):
    dist_color = (distance.cdist(x, y, metric='euclidean'))**2
    coord_list = list()
    for i in range(100):
        for j in range(100):
            coord_list.append((i, j))
    coord_x = np.array(coord_list)
    coord_y = np.copy(coord_x)
    dist_spatial = (distance.cdist(coord_x, coord_y, metric='euclidean'))**2

    return np.dot(np.exp(-gamma_s * dist_spatial), np.exp(-gamma_c *
dist_color))

def plot_res(prd_res, name, clusters):
    prd_res = prd_res.reshape((100, 100))
    plt.imshow(prd_res)
    plt.savefig(os.path.join(f'./{clusters}/{name}.png'))

    return

def kernel_kmeans(img):
    for gamma_c in Gamma_c:
```

```
        for gamma_s in Gamma_s:
            kernel = make_kernel(img, img, gamma_s, gamma_c)
            for k in K:
                # Initialization method 1
                # tmp = list()
                # for i in range(k-1):
                #     tmp += [i]*10000/k
                # tmp += [k-1]*(10000-len(tmp))
                # prd_res = np.array(tmp)

                # Initialization method 2
                prd_res = np.random.randint(k, size=10000)
                prd_prev = np.zeros(10000)
                iteration = 0
                while not np.array_equal(prd_res, prd_prev) and iteration <
60:
                    dist = np.tile(np.diag(kernel_im1), k).reshape(-1, k)
                    prd_prev = np.copy(prd_res)
                    print ('iteration:', iteration)
                    num_points = np.array([len(prd_res[prd_res==c]) for c in
range(k)])
                    print (num_points)
                    for c in range(k):
                        # print (kernel_im1[:, prd_res==c].shape)
                        dist[:, c] -= 2/num_points[c] * np.sum(kernel[:,
prd_res==c], axis=1)
                        dist[:, c] += (1/num_points[c])**2 *
np.sum(kernel[prd_res==c][:, prd_res==c])
                    prd_res = np.argmin(dist, axis=1)
                    iteration += 1

                plot_res(prd_res, f"{gamma_c}_{gamma_s}_{k}", k)

    return
```

kernel receives 2 numpy array and return the corresponding gram matrix.

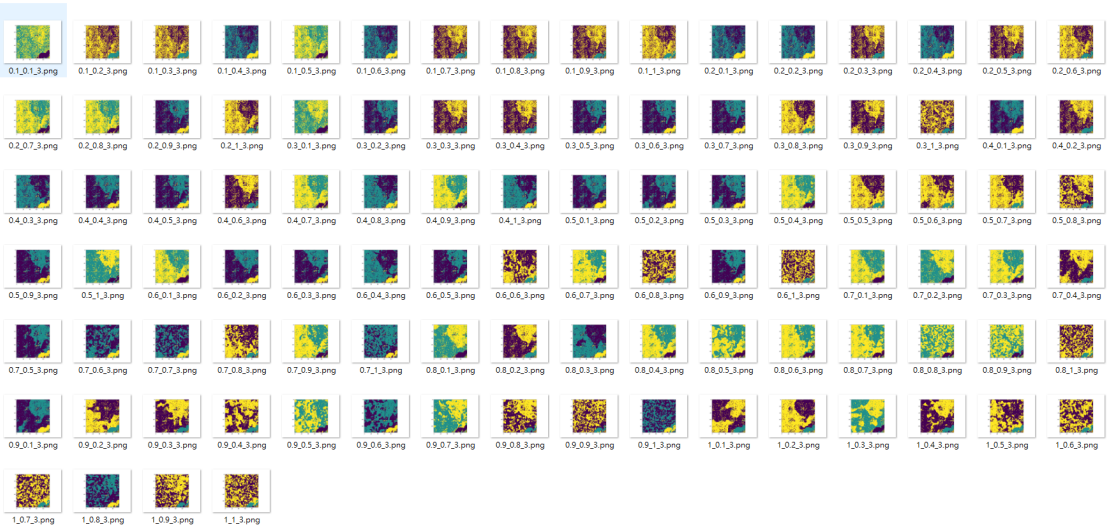
plot_res creates the prediction illustration and saves it.

kernel_kmeans implements the kernel kmeans algorithm and performs Grid Search on given range of hyper-parameters.

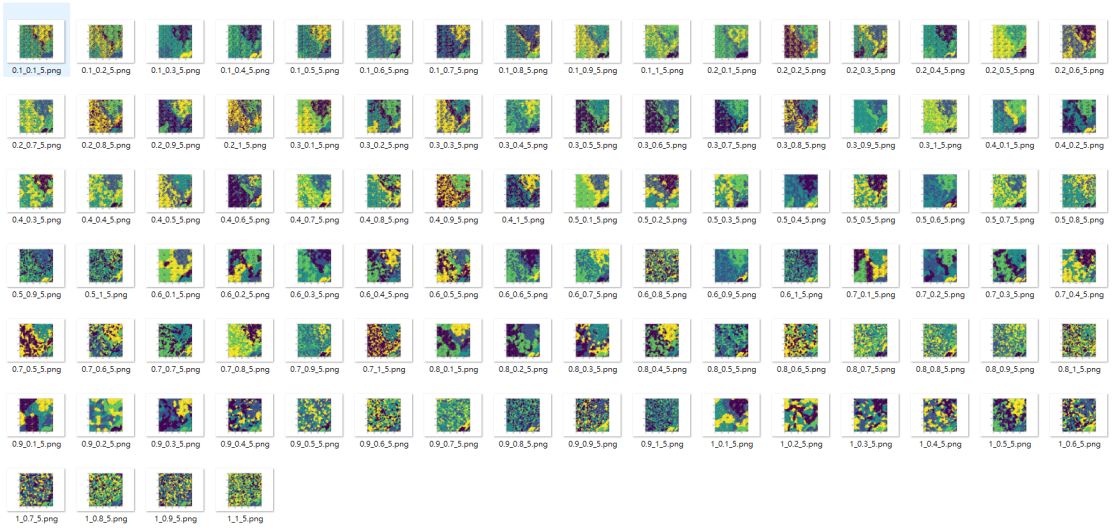
The converge condition is either the prediction holds still or #iterations have exceed 60.

Results:


0.1_0.1_2.png
0.1_0.2_2.png
0.1_0.3_2.png
0.1_0.4_2.png
0.1_0.5_2.png
0.1_0.6_2.png
0.1_0.7_2.png
0.1_0.8_2.png
0.1_0.9_2.png
0.1_1_2.png
0.2_0.1_2.png
0.2_0.2_2.png
0.2_0.3_2.png
0.2_0.4_2.png
0.2_0.5_2.png
0.2_0.6_2.png
0.2_0.7_2.png
0.2_0.8_2.png
0.2_0.9_2.png
0.2_1_2.png
0.3_0.1_2.png
0.3_0.2_2.png
0.3_0.3_2.png
0.3_0.4_2.png
0.3_0.5_2.png
0.3_0.6_2.png
0.3_0.7_2.png
0.3_0.8_2.png
0.3_0.9_2.png
0.3_1_2.png
0.4_0.1_2.png
0.4_0.2_2.png
0.4_0.3_2.png
0.4_0.4_2.png
0.4_0.5_2.png
0.4_0.6_2.png
0.4_0.7_2.png
0.4_0.8_2.png
0.4_0.9_2.png
0.4_1_2.png
0.5_0.1_2.png
0.5_0.2_2.png
0.5_0.3_2.png
0.5_0.4_2.png
0.5_0.5_2.png
0.5_0.6_2.png
0.5_0.7_2.png
0.5_0.8_2.png
0.5_0.9_2.png
0.5_1_2.png
0.6_0.1_2.png
0.6_0.2_2.png
0.6_0.3_2.png
0.6_0.4_2.png
0.6_0.5_2.png
0.6_0.6_2.png
0.6_0.7_2.png
0.6_0.8_2.png
0.6_0.9_2.png
0.6_1_2.png
0.7_0.1_2.png
0.7_0.2_2.png
0.7_0.3_2.png
0.7_0.4_2.png
0.7_0.5_2.png
0.7_0.6_2.png
0.7_0.7_2.png
0.7_0.8_2.png
0.7_0.9_2.png
0.7_1_2.png
0.8_0.1_2.png
0.8_0.2_2.png
0.8_0.3_2.png
0.8_0.4_2.png
0.8_0.5_2.png
0.8_0.6_2.png
0.8_0.7_2.png
0.8_0.8_2.png
0.8_0.9_2.png
0.8_1_2.png
0.9_0.1_2.png
0.9_0.2_2.png
0.9_0.3_2.png
0.9_0.4_2.png
0.9_0.5_2.png
0.9_0.6_2.png
0.9_0.7_2.png
0.9_0.8_2.png
0.9_0.9_2.png
0.9_1_2.png
1_0.1_2.png
1_0.2_2.png
1_0.3_2.png
1_0.4_2.png
1_0.5_2.png
1_0.6_2.png
1_0.7_2.png
1_0.8_2.png
1_0.9_2.png
1_1_2.png


0.1_0.1_3.png
0.1_0.2_3.png
0.1_0.3_3.png
0.1_0.4_3.png
0.1_0.5_3.png
0.1_0.6_3.png
0.1_0.7_3.png
0.1_0.8_3.png
0.1_0.9_3.png
0.1_1_3.png
0.2_0.1_3.png
0.2_0.2_3.png
0.2_0.3_3.png
0.2_0.4_3.png
0.2_0.5_3.png
0.2_0.6_3.png
0.2_0.7_3.png
0.2_0.8_3.png
0.2_0.9_3.png
0.2_1_3.png
0.3_0.1_3.png
0.3_0.2_3.png
0.3_0.3_3.png
0.3_0.4_3.png
0.3_0.5_3.png
0.3_0.6_3.png
0.3_0.7_3.png
0.3_0.8_3.png
0.3_0.9_3.png
0.3_1_3.png
0.4_0.1_3.png
0.4_0.2_3.png
0.4_0.3_3.png
0.4_0.4_3.png
0.4_0.5_3.png
0.4_0.6_3.png
0.4_0.7_3.png
0.4_0.8_3.png
0.4_0.9_3.png
0.4_1_3.png
0.5_0.1_3.png
0.5_0.2_3.png
0.5_0.3_3.png
0.5_0.4_3.png
0.5_0.5_3.png
0.5_0.6_3.png
0.5_0.7_3.png
0.5_0.8_3.png
0.5_0.9_3.png
0.5_1_3.png
0.6_0.1_3.png
0.6_0.2_3.png
0.6_0.3_3.png
0.6_0.4_3.png
0.6_0.5_3.png
0.6_0.6_3.png
0.6_0.7_3.png
0.6_0.8_3.png
0.6_0.9_3.png
0.6_1_3.png
0.7_0.1_3.png
0.7_0.2_3.png
0.7_0.3_3.png
0.7_0.4_3.png
0.7_0.5_3.png
0.7_0.6_3.png
0.7_0.7_3.png
0.7_0.8_3.png
0.7_0.9_3.png
0.7_1_3.png
0.8_0.1_3.png
0.8_0.2_3.png
0.8_0.3_3.png
0.8_0.4_3.png
0.8_0.5_3.png
0.8_0.6_3.png
0.8_0.7_3.png
0.8_0.8_3.png
0.8_0.9_3.png
0.8_1_3.png
0.9_0.1_3.png
0.9_0.2_3.png
0.9_0.3_3.png
0.9_0.4_3.png
0.9_0.5_3.png
0.9_0.6_3.png
0.9_0.7_3.png
0.9_0.8_3.png
0.9_0.9_3.png
0.9_1_3.png
1_0.1_3.png
1_0.2_3.png
1_0.3_3.png
1_0.4_3.png
1_0.5_3.png
1_0.6_3.png
1_0.7_3.png
1_0.8_3.png
1_0.9_3.png
1_1_3.png


0.1_0.1_4.png
0.1_0.2_4.png
0.1_0.3_4.png
0.1_0.4_4.png
0.1_0.5_4.png
0.1_0.6_4.png
0.1_0.7_4.png
0.1_0.8_4.png
0.1_0.9_4.png
0.1_1_4.png
0.2_0.1_4.png
0.2_0.2_4.png
0.2_0.3_4.png
0.2_0.4_4.png
0.2_0.5_4.png
0.2_0.6_4.png
0.2_0.7_4.png
0.2_0.8_4.png
0.2_0.9_4.png
0.2_1_4.png
0.3_0.1_4.png
0.3_0.2_4.png
0.3_0.3_4.png
0.3_0.4_4.png
0.3_0.5_4.png
0.3_0.6_4.png
0.3_0.7_4.png
0.3_0.8_4.png
0.3_0.9_4.png
0.3_1_4.png
0.4_0.1_4.png
0.4_0.2_4.png
0.4_0.3_4.png
0.4_0.4_4.png
0.4_0.5_4.png
0.4_0.6_4.png
0.4_0.7_4.png
0.4_0.8_4.png
0.4_0.9_4.png
0.4_1_4.png
0.5_0.1_4.png
0.5_0.2_4.png
0.5_0.3_4.png
0.5_0.4_4.png
0.5_0.5_4.png
0.5_0.6_4.png
0.5_0.7_4.png
0.5_0.8_4.png
0.5_0.9_4.png
0.5_1_4.png
0.6_0.1_4.png
0.6_0.2_4.png
0.6_0.3_4.png
0.6_0.4_4.png
0.6_0.5_4.png
0.6_0.6_4.png
0.6_0.7_4.png
0.6_0.8_4.png
0.6_0.9_4.png
0.6_1_4.png
0.7_0.1_4.png
0.7_0.2_4.png
0.7_0.3_4.png
0.7_0.4_4.png
0.7_0.5_4.png
0.7_0.6_4.png
0.7_0.7_4.png
0.7_0.8_4.png
0.7_0.9_4.png
0.7_1_4.png
0.8_0.1_4.png
0.8_0.2_4.png
0.8_0.3_4.png
0.8_0.4_4.png
0.8_0.5_4.png
0.8_0.6_4.png
0.8_0.7_4.png
0.8_0.8_4.png
0.8_0.9_4.png
0.8_1_4.png
0.9_0.1_4.png
0.9_0.2_4.png
0.9_0.3_4.png
0.9_0.4_4.png
0.9_0.5_4.png
0.9_0.6_4.png
0.9_0.7_4.png
0.9_0.8_4.png
0.9_0.9_4.png
0.9_1_4.png
1_0.1_4.png
1_0.2_4.png
1_0.3_4.png
1_0.4_4.png
1_0.5_4.png
1_0.6_4.png
1_0.7_4.png
1_0.8_4.png
1_0.9_4.png
1_1_4.png



After observing the performance of combinations of gamma_c and gamma_s, I decided to use **(gamma_c, gamma_s) = (0.9, 0.1)** for the following experiments of the 2 images.

---

4. **Perform Experiments on Different Cluster Size and Initialization Method of Kernel-Kmeans**

```
# Modified for running multiple task imgs
def plot_res(prd_res, name, clusters, img):
    prd_res = prd_res.reshape((100, 100))
    plt.imshow(prd_res)
    plt.savefig(os.path.join(f'./tmp/IMG{img}/{clusters}/{name}.png'))

    return

def kernel_kmeans(img, order):
    for gamma_c in Gamma_c:
        for gamma_s in Gamma_s:
            kernel = make_kernel(img, img, gamma_s, gamma_c)
            for k in K:
                tmp = list()
                for i in range(k-1):
                    tmp += [i]*int(10000/k)
                tmp += [k-1]*(10000-len(tmp))
                prd_res = np.array(tmp)
                # prd_res = np.random.randint(k, size=10000)
                prd_prev = np.zeros(10000)
                iteration = 0
                while not np.array_equal(prd_res, prd_prev) and iteration <
60:
                    dist = np.tile(np.diag(kernel_im1), k).reshape(-1, k)
                    prd_prev = np.copy(prd_res)
                    print ('iteration:', iteration)
                    num_points = np.array([len(prd_res[prd_res==c]) for c in
range(k)])
                    print (num_points)
                    for c in range(k):
                        # print (kernel_im1[:, prd_res==c].shape)
                        dist[:, c] -= 2/num_points[c] * np.sum(kernel[:,
prd_res==c], axis=1)
                        dist[:, c] += (1/num_points[c])**2 *
np.sum(kernel[prd_res==c][:, prd_res==c])
```

```
                    prd_res = np.argmin(dist, axis=1)
                    iteration += 1

                    plot_res(prd_res,
f"m1_{gamma_c}_{gamma_s}_{k}_ite{iteration}", k, order)

    return

if __name__ == "__main__":
    im1 = np.array(Image.open('./image1.png')).reshape((10000, 3)) / 255
    im2 = np.array(Image.open('./image2.png')).reshape((10000, 3)) / 255

    Gamma_s, Gamma_c = [0.1], [0.9]
    K = [2, 3, 4, 5]

    kernel_kmeans(im1, 1)
    kernel_kmeans(im2, 2)
```

Note that I choose 2 methods: **assign label in ascending order**(M1) and **randomly assign label**(M2)

I modified the ploting function to generate current prediction illustration at each iteration, and generate GIF using online GIF maker.

---

5. **Results and Discussion**
    - Number of Clusters：I tried 2, 3, 4, 5 clusters. **For IMG1, 3,4 are better**, and **for IMG2, the result of cluster 5 is best among all**. And is consistent with the original images.
    - Initialization Method and Converge Speed：Training with Method2 is faster than with Method1. And is especially obvious with K=4 or 5.

Spectual Clustering

1. **Extract data, normalize to 0~1 and transform to 10000X3 numpy array**

```
im1 = np.array(Image.open('./image1.png')).reshape((10000, 3)) / 255
im2 = np.array(Image.open('./image2.png')).reshape((10000, 3)) / 255
```

---

2. **Set up hyper parameters and main function, define kernel**

```
def kernel(x, y, gamma_s, gamma_c):
    dist_color = (distance.cdist(x, y, metric='euclidean'))**2
    coord_list = list()
    for i in range(100):
        for j in range(100):
            coord_list.append((i, j))
    coord_x = np.array(coord_list)
    coord_y = np.copy(coord_x)
    dist_spatial = (distance.cdist(coord_x, coord_y, metric='euclidean'))**2

    return np.dot(np.exp(-gamma_s * dist_spatial), np.exp(-gamma_c *
dist_color))
```

```
gamma_s, gamma_c = 0.1, 0.9
K = [2, 3, 4, 5]
for k in K:
    spectual_clustering(im1, k, 'normal', 1)
    spectual_clustering(im1, k, 'ratio', 1)

    spectual_clustering(im2, k, 'normal', 2)
    spectual_clustering(im2, k, 'ratio', 2)
```

Note that I directly use the same gamma_c, gamma_s as that of Kernel-kmeans.

---

3. **Spectual Clustering and Plotting Implementation**

```
def plot_data(U, name, prd_result, k):
    fig = plt.figure()
    ax = Axes3D(fig)
    if k == 2:
        for i in range(k):
            cluster_data = U[prd_result == i]
            ax.scatter(cluster_data[:, 0], cluster_data[:, 1], c=colors[i])
    else:
        for i in range(k):
            cluster_data = U[prd_result == i]
            ax.scatter(cluster_data[:, 0], cluster_data[:, 1],
cluster_data[:, 2], c=colors[i])

    plt.savefig(f'./{name}')
    return

def plot_res(prd_res, name, clusters, img):
    plt.close()
    prd_res = prd_res.reshape((100, 100))
    plt.imshow(prd_res)
    plt.savefig(os.path.join(f'./tmp/IMG{img}/sp/{clusters}/{name}.png'))

    return

def kmeans(U, k, method, img):
    prd_result = np.random.randint(k, size=10000)
    center = np.zeros((k, k))
    prev_center = np.ones((k, k))
    iteration = 0
    plot_res(prd_result, f'{method}_{k}_{iteration}', k, img)
    while not np.array_equal(center, prev_center) and iteration < 100:
        prev_center = np.copy(center)
        for i in range(k):
            center[i] = np.mean(U[prd_result == i], axis=0)
        print ([len(prd_result[prd_result == i]) for i in range(k)])
        prd_result = np.argmin(distance.cdist(U, center), axis=1)
        iteration += 1
        plot_res(prd_result, f'{method}_{k}_{iteration}', k, img)
    return prd_result

def spectual_clustering(img, k, method, img_count):
    kernel_img = kernel(img, img, gamma_s, gamma_c)
```

```
    D = np.diag(np.sum(kernel_img, axis=1))

    if method == 'normal':
        D_prime = np.linalg.inv(np.sqrt(D))
        L = np.dot(D_prime, np.dot(D - kernel_img, D_prime))
    else:
        L = D - kernel_img

    print ('Start eigen decomposition')
    eig_val, eig_vec = np.linalg.eig(L)
    eigen_dict = dict(zip(eig_val, eig_vec.T))
    U = np.zeros((10000, k))
    for i, val in enumerate(np.sort(eig_val)[1:]):
        print (i, val)
        if i == k:
            break
        else:
            U[:, i] = eigen_dict[val]
    for i in range(len(U)):
        U[i] /= np.linalg.norm(U[i])

    prd_result = kmeans(U, k, method, img_count)
    return
```

spectual_clustering implements the algorithm of both Ratio/Normal cuts.

After all data points are projected to eigenspace as **U**, **plot_data** is called to show the data points in eigenspace.

Note that if k>3, then only the first 3 dimension will be adopted.

kmeans is then called to perform a clustering to predict the result.

And **plot_res** save the current prediction at every iteration for creating GIF.

4. Experiment Result and Discussion
   Observing points in eigenspace, Normalized Cut looks more "cubic" than that of Ratio cut. As a result, Normalized Cut takes more time to converge.

Appendix

In this part I will explain my GIF naming principle and directory structure.

The dir-structure should look like as following

- GIF_RESULTS
  - Kernel-Kmeans
    - IMGS in format {IMG NO}-{INIT_Method}-{#Clusters}.png.
    - README recording Naming principles for TA's convenience.
  - Spectual-Clustering
    - EigenSpace
      - IMGS in format {Normal/Ratio}-{#Clusters}-{IMG NO}.png showing data in eigenspace.
      - README recording Naming principles for TA's convenience.
    - NormalCut
      - IMGS in format {IMG NO}-{Normal/Ratio}-{INIT_Method}-{#Clusters}.png.
      - README recording Naming principles for TA's convenience.

- RatioCut
    - IMGS in format {IMG NO}-{Normal/Ratio}-{INIT_Method}-{#Clusters}.png.
    - README recording Naming principles for TA's convenience.
- Note that I didn't run some case of testcase with INIT METHOD = M1 and k = 5 for time reason.