Machine Learning HW6

0510008 藍挺毓

I. Code

A. Kernel K-means

1. Main function

- load_png() is a self defined function, that simply load input image with PTL.

```
X_color = load_png() #(10000, 3)
X_spatial, fi_X = RBF_kernel(X_color, args.s, args.c) #spatial, color
cluster = initial_kernel_kmeans(args.k, fi_X)
kernel_kmeans(args.k, fi_X, cluster, iter=args.iter)
```

2. RBF_kernel(): self-defined kernel function that uses two RBF
   - I consider both spatial and color information into my kernel function.
   - X is input image, which contains color information
   - X_s contains coordinate for each pixel in an input image
   - RBF_s is RBF kernel made from spatial information
   - RBF_c is RFB kernel made from color information
   - My self defined kernel simply multiply RBF_s and RBF_c.

```
def RBF_kernel(X, gamma_s, gamma_c): #img, spatial, color
    dist_c = cdist(X,X,'sqeuclidean') #(10000,10000)

    seq = np.arange(0,100)
    c_coord = seq
    for i in range (99):
        c_coord = np.hstack((c_coord, seq))
    c_coord = c_coord.reshape(-1,1)
    r_coord = c_coord.reshape(100,100).T.reshape(-1,1)
    X_s = np.hstack((r_coord, c_coord))
    dist_s = cdist(X_s,X_s,'sqeuclidean')

    RBF_s = np.exp( -gamma_s * dist_s)
    RBF_c = np.exp( -gamma_c * dist_c) #(10000,10000)
    k = np.multiply(RBF_s, RBF_c) #(10000,10000)

    return X_s, k
```

3. Initial_kernel_kmeans()
   - Call initial_center function to find initial centers. (I will introduce this function later in this report)
   - Use found centers to initial clustering. I first calculate each pixels' distance to each center in feature space following the equation $||\phi(X_n) - \phi(\mu_n)|| = k(x_n, x_n) + k(\mu_k, \mu_k) - 2k(x_n, \mu_k)$. A pixels would be clustered to the center that has closest distance to it.

```
def initial_kernel_kmeans(K, data):
    centers_idx = initial_center(K, mode="kmean++")

    N = len(data)
    cluster = np.zeros(N, dtype=int)
    for n in range(N):
        dist = np.zeros(K)
        for k in range(K):
            dist[k] = data[n,n] + data[centers_idx[k],centers_idx[k]] - 2*data[n,centers_idx[k]]
        cluster[n] = np.argmin(dist)
    return cluster
```

4. kernel_kmeans
   - kernel_kmeans simply repeat clustering(). It would stop when cluster do not change anymore.
   - clustering follows the equation shown below.

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

Gram matrix!

```python
def kernel_kmeans(K, kernel, cluster, iter=1000):
    save_png(len(kernel), K, cluster, 0)
    for i in range(1, iter+1):
        print("iter", i)
        new_cluster = clustering(K, kernel, cluster)
        if(np.linalg.norm((new_cluster-cluster), ord=2)<1e-2):
            break
        cluster = new_cluster
        save_png(len(kernel), K, cluster, i)
```

```python
def clustering(K, kernel, cluster):
    N = len(kernel)
    new_cluster = np.zeros(N, dtype=int)
    C = construct_C(K, cluster)
    pq = construct_sigma_pq(C, K, kernel, cluster)
    for j in range(N):
        dist = np.zeros(K)
        for k in range(K):
            dist[k] += kernel[j,j] + pq[k]
            dist[k] -= 2/C[k] * construct_sigma_n(kernel[j,:], cluster, k)
        new_cluster[j] = np.argmin(dist)

    return new_cluster
```

B. Spectral clustering - ratio cut
   1. It can be proof that the result of ratio cut and unnormalized spectral clustering would be equal.
   2. main function
      - Follow the algorithm shown on the right.
      - Since it is time consuming to calculate eigen value, eigen vector, and Laplacian. I would save it as pickle or npy file for further use.

```python
# ****************** eigen ******************
X_spatial, W = RBF_kernel(X_color, 0.001, 0.01)
D, L = construct_Laplacian(W)
save_pkl(filename, isNorm=False)

# X_spatial, W, D, L = load_pkl(filename, isNorm=False)
# *******************************************

# ****************** eigen ******************
eigenvalue, eigenvector = np.linalg.eig(L)
eigenvector = eigenvector.T
np.save(data_name+"_eigenvalue.npy", eigenvalue)
np.save(data_name+"_eigenvector.npy", eigenvector)
# eigenvalue = np.load(filename+"_eigenvalue.npy")
# eigenvector = np.load(filename+"_eigenvector.npy")
# *******************************************
sort_idx = np.argsort(eigenvalue)
mask = eigenvalue[sort_idx]>0
sort_idx = sort_idx[mask]

U = eigenvector[sort_idx[0:args.k]].T
centers = initial_center(args.k, U, mode="random")
kmeans(args.k, U, centers, iter=args.iter)
```

Unnormalized spectral clustering

Input: Similarity matrix $S \in \mathbb{R}^{n \times n}$, number k of clusters to construct.
- Construct a similarity graph by one of the ways described in Section 2. Let W be its weighted adjacency matrix.
- Compute the unnormalized Laplacian L.
- **Compute the first k eigenvectors $u_1, \ldots, u_k$ of L.**
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \ldots, u_k$ as columns.
- For $i = 1, \ldots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of U.
- Cluster the points $(y_i)_{i=1,\ldots,n}$ in $\mathbb{R}^k$ with the k-means algorithm into clusters $C_1, \ldots, C_k$.

Output: Clusters $A_1, \ldots, A_k$ with $A_i = \{j | y_j \in C_i\}$.

3. construct_Laplacian()
   - First calculate Diagonal matrix with weight, then calculate Laplacian matrix.

```python
def construct_Laplacian(W):
    D = np.zeros((W.shape))
    L = np.zeros((W.shape))
    for r in range(len(W)):
        for c in range(len(W)):
            D[r,r] += W[r,c]

    L = D-W
    return D, L
```

4. kmeans()
   - There are two repeating step in kmeans, which are clustering and find new centers for new clusters.
   - clustering simply find nearest center for each data point. Then, cluster that data point to that nearest center.
   - find_centers() calculate mean for each cluster. These mean would become new centers.

```python
def kmeans(K, data, centers, iter=1000):
    for i in range(iter):
        print("iter", i)
        new_cluster = clustering(K, data, centers)
        if i!=0:
            if(np.linalg.norm((new_cluster-cluster), ord=2)<1e-2):
                break
        cluster = new_cluster
        save_png(len(data), cluster, centers, i)
        centers = find_centers(K, data, cluster, centers)
    cluster = clustering(K, data, centers)
    save_png(len(data), cluster, centers, iter)
```

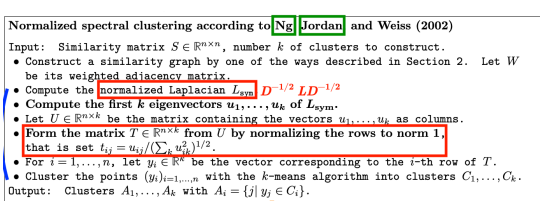
```python
def clustering(K, data, centers):
    N = len(data)
    cluster = np.zeros(N, dtype=int)
    for n in range(N):
        c = -1
        min_dist = np.Inf
        for k in range(K):
            dist = np.linalg.norm((data[n]-centers[k]), ord=2)
            if dist < min_dist:
                c = k
                min_dist = dist
        cluster[n] = c
    return cluster
```

```python
def find_centers(K, U, cluster, centers):
    new_centers = []
    for k in range(K):
        mask = cluster==k
        cluster_k = U[mask]
        new_center_k = np.sum(cluster_k, axis=0) / len(cluster_k)
        new_centers.append(new_center_k)

    return new_centers
```

5. save_png()
   - Use pre-defined colors to label different clusters with different colors.

```python
def save_png(N, cluster, centers, iter):
    colors = np.array([[255,0,0],[0,255,0],[0,0,255],[0,215,175],[95,0,135],[255,255,0],[255,175,0]])
    result = np.zeros((100*100, 3))
    for n in range(N):
        result[n,:] = colors[cluster[n],:]

    img = result.reshape(100, 100, 3)
    img = Image.fromarray(np.uint8(img))
    img.save(os.path.join(output_dir, '%06d.png'%iter))
```

C. Spectral clustering - ratio cut
1. It can be proof to have same result with normalized spectral clustering.
2. main function
   - Simply follow the algorithm shown below. There are many shared function with A. and B.



Normalized spectral clustering according to Ng, Jordan, and Weiss (2002)

Input: Similarity matrix $S \in \mathbb{R}^{n \times n}$, number k of clusters to construct.
- Construct a similarity graph by one of the ways described in Section 2. Let W be its weighted adjacency matrix.
- Compute the normalized Laplacian L_{sym} $D^{-1/2}LD^{-1/2}$
- Compute the first k eigenvectors $u_1, \ldots, u_k$ of L_{sym}.
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \ldots, u_k$ as columns.
- Form the matrix $T \in \mathbb{R}^{n \times k}$ from U by normalizing the rows to norm 1, that is set $t_{ij} = u_{ij}/(\sum_k u_{ik}^2)^{1/2}$.
- For $i = 1, \ldots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of T.
- Cluster the points $(y_i)_{i=1,\ldots,n}$ with the k-means algorithm into clusters $C_1, \ldots, C_k$.

Output: Clusters $A_1, \ldots, A_k$ with $A_i = \{j | y_j \in C_i\}$.

```python
# ***************** pkl *****************
X_spatial, W = RBF_kernel(X_color, 0.001, 0.01)
D, L = construct_Laplacian(W)
Dsym = D_minus_half_square_root(D)
Lsym = Dsym.dot(L).dot(Dsym)
save_pkl(filename, isNorm=True)

# X_spatial, W, D, L, Dsym, Lsym = load_pkl(filename, isNorm=True)
# ***************************************

# ***************** eigen *****************
eigenvalue, eigenvector = np.linalg.eig(Lsym)
eigenvector = eigenvector.T
```

```python
sort_idx = np.argsort(eigenvalue)
mask = eigenvalue[sort_idx]>0
sort_idx = sort_idx[mask]

U = eigenvector[sort_idx[0:args.k]].T
T = normalize_rows(U)
centers = initial_center(args.k, T, mode="random")
kmeans(args.k, T, centers, iter=args.iter)
```

D. Initialization of k-means clustering

1. Random from exist data points

```
if mode=="random":
    centers_idx = list(random.sample(range(0,10000), K))
```

2. k-means++

- Random find one data point.
- Find farthest data point.
- Repeat finding farthest data point till found k initial data points.

```
elif mode=="kmean++":
    centers_idx = []
    centers_idx = list(random.sample(range(0,10000), 1))
    found = 1
    while (found<K):
        dist = np.zeros(10000)
        for i in range(10000):
            min_dist = np.Inf
            for f in range(found):
                tmp = np.linalg.norm(X_spatial[i,:]-X_spatial[centers_idx[f],:])
                if tmp<min_dist:
                    min_dist = tmp
            dist[i] = min_dist
        dist = dist/np.sum(dist)
        idx = np.random.choice(np.arange(10000), 1, p=dist)
        centers_idx.append(idx[0])
        found += 1
```

E. Coordinate in the Eigen space of graph Laplacian

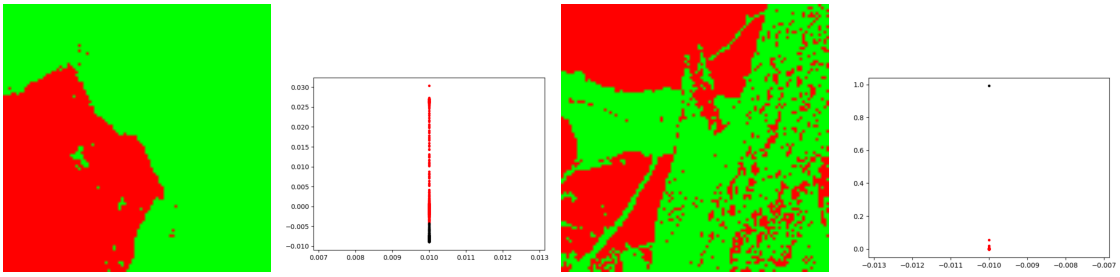
- Simply show Eigen space coordinate with different color for different cluster

```
def show_eigen(K, data, cluster):
    colors = ['b', 'r']
    plt.clf()
    for idx in range(len(data)):
        plt.scatter(data[idx,0], data[idx,1], c= colors[cluster[idx]])
    plt.savefig("eigen_"+args.filename+".png")
    plt.show()
```

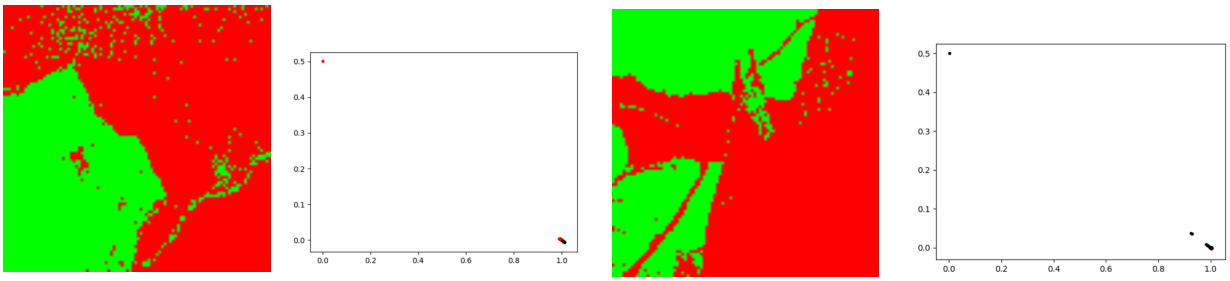
II. Results

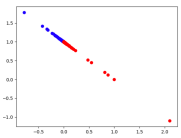
A. Kernel K-means

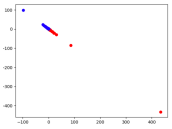
B. Ratio cut



C. Normalize cut

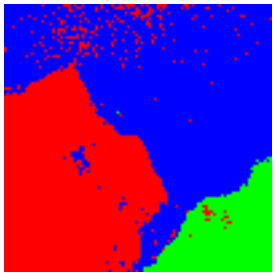






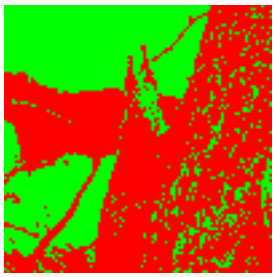
III. Discussion

A.

- There are some red dots within blue cluster. This is due to the reason that there are green dots, which is similar to green island, in ocean in input image. The algorithm shows great performance here.

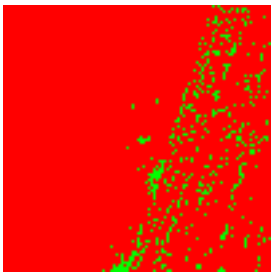
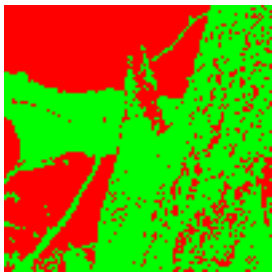
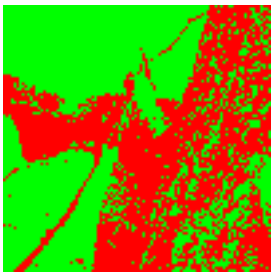
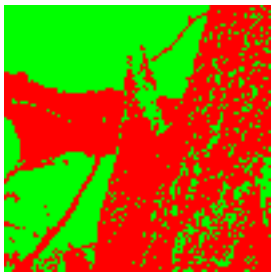
B.

- Some part of trunk is labeled as background. This is not what we expected to have. Nevertheless, it is reasonable. Some parts of trunk has similar color, light blue, with sky in the background. Since I consider color in my kernel, these blue dots would be considered similar to blue sky. Thus, they may be classify as same cluster.

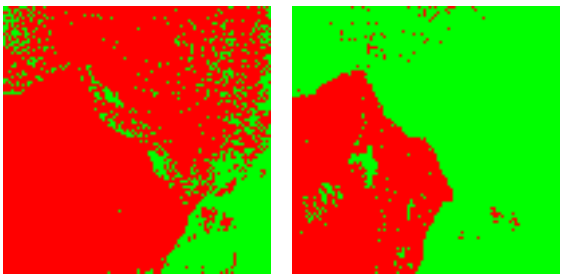
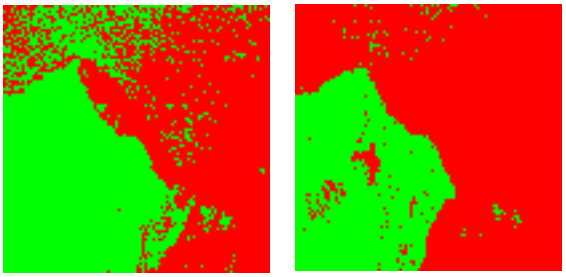
C. Random initial v.s. k-means++

- These two initial may lead to same result. However, there is higher chances for random initial to result in bad clustering.
- Random initial may get really close centers at the beginning of the program. Thus, it requires more time to converge to final clustering than k-means++.
- k-mean++ guarantee to find far initial centers. It is really important to have centers far from each others. It may have no huge difference in this assignment since the input data is not huge. The cost is still ok to have a worse initial. However, it would show great difference to have good initial in larger computational or data cases. Thus, it is good to have k-mean++ than simply random.
- Example 1:

random initial iteration 10 k-means++ initial iteration 8

- From above example, we verify k-means++ has better initial. Moreover, it use only 8 iteration to converge. As to random initial, it required 10 iteration to converge.
- Example 2:

random initial iteration 9 k-means++ initial iteration 8

D. How to choose k
   - Since it is a unsupervised learning. There is no correct k to be chosen. However, we can see from the result that there are better choices. Take image1.png as an example. It is a more reasonable result use clustering to separate the island and ocean. Thus, it is good to choose k as 2. Choosing k as 3 may still be reasonable, since there are significant 2 difference colors in ocean. We can label them with different class. However, it would be weird to choose k as 5 or 6 or even larger.
   - k is a data-dependent variable. There is no general solution to find one. Just try and see tendency from results.