Machine Learning HW6

Code

Kernel Kmeans

Kernel

The below kernel function was be asked for in this homework

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

```
def compute_kernel(image, gamma_s, gamma_c):
    row, col, color = image.shape
    ### the distance of each color
    color_distance = cdist(image.reshape(row * col, color), image.reshape(row * col, color), 'sqeuclidean')
    ### the indices vector
    grid = np.indices((row, col))
    row_indices, col_indices = grid[0], grid[1]
    indices = np.hstack((row_indices.reshape(-1, 1), col_indices.reshape(-1, 1)))

    ### the spatial distance
    spatial_distance = cdist(indices, indices, 'sqeuclidean')

    return np.multiply(np.exp(-gamma_s * spatial_distance), np.exp(-gamma_c * color_distance))
```

Initial Centers

There are two strategy to decide the center of each cluster at the begining. The first way is random choose a point as the center, it also mean the random initialize. The second way is choose the center by the kean++ strategy.

```
def choose_center(n_rows, n_cols, n_clusters, mode):
    if not mode:
        ### random strategy
        return np.random.choice(100, (n_clusters, 2))
    else:
        ### kmean++ strategy

        grid = np.indices((n_rows, n_cols))
        row_indices, col_indices = grid[0], grid[1]
        indices = np.hstack((row_indices.reshape(-1, 1), col_indices.reshape(-1, 1)))

        ### pick the init center randomly
        n_points = n_rows * n_cols
        centers = [indices[np.random.choice(n_points, 1)[0]].tolist()]

        ### find remaining centers
        for _ in range(n_clusters - 1):
            distance = np.zeros(n_points)
            for idx, point in enumerate(indices):
                min_distance = np.Inf
                for center in centers:
                    dist = np.linalg.norm(point - center)
                    min_distance = dist if dist < min_distance else min_distance
                distance[idx] = min_distance
            ### get the probability of the distance
            distance /= np.sum(distance)
            ### new center
            centers.append(indices[np.random.choice(n_points, 1, p=distance)[0]].tolist())

        return np.array(centers)
```

Initail Clustering

After found the center of each cluster, the next step is assign all points to some cluster by minimize the distance between its and the center point.

```
def init_clustering(n_rows, n_cols, n_clusters, kernel, mode):
    ### init centers
    centers = choose_center(n_rows, n_cols, n_clusters, mode)

    ### k-means
    n_points = n_rows * n_cols
    cluster = np.zeros(n_points, dtype=int)
    for p in range(n_points):
        ### calculate the distance between each center and each point
        distance = np.zeros(n_clusters)
        for idx, center in enumerate(centers):
            seq_center = center[0] * n_rows + center[1]
            distance[idx] = kernel[p, p] + kernel[seq_center, seq_center] - 2 * kernel[p, seq_center]

        ### put the point into the nearest cluster
        cluster[p] = np.argmin(distance)

    return cluster
```

Do The Kernel Kmeans

We use `capture_current_state` function to get the current clustering classification staus, and compute the new clustering for each point by `kernel_clustering` function follows the formula:

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

```
def get_sum_of_pairwise_distance(n_points, n_clusters, n_members, kernel, cluster):
    pairwise_distance = np.zeros(n_clusters)
    for c in range(n_clusters):
        tmp_kernel =  kernel.copy()
        for p in range(n_points):
            if cluster[p] != c:
                tmp_kernel[p, :]
                tmp_kernel[:, p]
        pairwise_distance = np.sum(tmp_kernel)

    ### if n_members == 0
    n_members[n_members == 0] = 1

    return pairwise_distance / n_members ** 2
```

It will repeat many times until the training converge or reach the maximun iteration time.

```python
def kernel_kmeans(n_rows, n_cols, n_clusters, cluster, kernel, mode, index):
    ### colors
    colors = np.array([[255, 0, 0], [0, 255, 0], [0, 0, 255]])
    if n_clusters > 3:
        colors = np.append(colors, np.random.choice(256, (n_clusters - 3, 3)), axis=0)

    ### list storing image of cluster state
    img = [capture_current_state(n_rows, n_cols, cluster, colors)]

    ### kernel kmeans
    current_cluster = cluster.copy()
    count = 0
    iteration = 100
    while True:
        ### get new cluster
        new_cluster = kernel_clustering(n_rows * n_cols, n_clusters, kernel, current_cluster)

        ### capture new state
        img.append(capture_current_state(n_rows, n_cols, new_cluster, colors))

        if np.linalg.norm((new_cluster - current_cluster), ord=2) < 0.001 or count >= iteration:
            break

        current_cluster = new_cluster.copy()
        count += 1

    ### save as gif
    filename = f'./gifs/kernel_kmeans/image{index}_cluster{n_clusters}_{"kmeans" if mode else "random"}.gif'
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    img[0].save(filename, save_all=True, append_images=img[1:], optimize=False, loop=0, duration=100)
```

```python
def kernel_clustering(n_points, n_clusters, kernel, cluster):
    ### number of members in each cluster
    n_members = np.array([np.sum(np.where(cluster == c, 1, 0)) for c in range(n_clusters)])

    ### sum of pairwise kernel distance of each cluster
    pairwise_distance = get_sum_of_pairwise_distance(n_points, n_clusters, n_members, kernel, cluster)

    new_cluster = np.zeros(n_points, dtype=int)
    for p in range(n_points):
        distance = np.zeros(n_clusters)
        for c in range(n_clusters):
            distance[c] += kernel[p, p] + pairwise_distance[c]

            ### the distance between others in the target cluster
            distance2others = np.sum(kernel[p, :][np.where(cluster == c)])
            distance[c] -= 2.0 / n_members[c] * distance2others
        new_cluster[p] = np.argmin(distance)

    return new_cluster
```

```python
def capture_current_state(n_rows, n_cols, cluster, colors):
    state = np.zeros((n_rows * n_cols, 3))
    for p in range(n_rows * n_cols):
        state[p, :] = colors[cluster[p], :]

    state = state.reshape((n_rows, n_cols, 3))

    return Image.fromarray(np.uint8(state))
```

After the training, save the result as a .gif file.

Spectral Clustering

Kernel

The way of calculate the kernel function is same as Kernel Kmeans.

```python
def compute_kernel(image, gamma_s, gamma_c):
    row, col, color = image.shape
    ### the distance of each color
    color_distance = cdist(image.reshape(row * col, color), image.reshape(row * col, color), 'sqeuclidean')
    ### the indices vector
    grid = np.indices((row, col))
    row_indices, col_indices = grid[0], grid[1]
    indices = np.hstack((row_indices.reshape(-1, 1), col_indices.reshape(-1, 1)))

    ### the spatial distance
    spatial_distance = cdist(indices, indices, 'sqeuclidean')

    return np.multiply(np.exp(-gamma_s * spatial_distance), np.exp(-gamma_c * color_distance))
```

Matrix U (Contain Eigenvectors)

U is the matrix which contains the first k eigenvectors of the Laplacian matrix L. The Laplacian matrix is made by the weight matrix W and the degree matrix D

- Unnormalized Laplacian $\boxed{\boldsymbol{L}=\boldsymbol{D}-\boldsymbol{W}}$ serve in the approximation of the minimization of RatioCut

- Normalized Laplacian $\boxed{D^{-1/2}\,\boldsymbol{L}D^{-1/2}}$ serve in the approximation of the minimization of NormalizedCut.

Then, we compute eigenvectors eigenvalues of the L, and sort eigenvector by the increasing order.

```
def compute_matrix_u(matrix_w, cut, n_clusters):
    ### get the laplacian matrix L and degree matrix D
    matrix_d = np.zeros_like(matrix_w)

    for idx, row in enumerate(matrix_w):
        matrix_d[idx, idx] += np.sum(row)

    matrix_l = matrix_d - matrix_w

    if cut:
        ### normalized cut
        ### compute the normalized laplacian
        for idx in range(len(matrix_d)):
            matrix_d[idx, idx] = 1.0 / np.sqrt(matrix_d[idx, idx])

        matrix_l = matrix_d.dot(matrix_l).dot(matrix_d)

    ### else is the ratio cut
    ### get eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(matrix_l)
    eigenvectors = eigenvectors.T

    ### sort eigenvalues and find indices of nonzero eigenvalues
    sort_idx = np.argsort(eigenvalues)
    sort_idx = sort_idx[eigenvalues[sort_idx] > 0]

    return eigenvectors[sort_idx[:n_clusters]].T
```

Spectral Clustering

Note that the matrix U is the composed by eigenvectors of the Laplacian matrix, with respect to the ordered eigenvalues. The spctral clustering is use the matrix U to initalize the centers of each point, and perform `kmeans` to get the result after converge. At the end, it will plot data points in the eigenspace.

```
def spectral_clustering(n_rows, n_cols, n_clusters, matrix_u, mode, cut, index):
    centers = init_centers(n_rows, n_cols, n_clusters, matrix_u, mode)

    ### k-means
    clusters = kmeans(n_rows, n_cols, n_clusters, matrix_u, centers, index, mode, cut)

    ### plot data point in eigenspace if number of clusters is 2
    if n_clusters == 2:
        plot_result(matrix_u, clusters, index, mode, cut)
```

Initial Centers

There are also have two mode of initialization: random and kmeans++ strategy. This function will return centers using coordinates in the eignespace.

```python
def init_centers(n_rows, n_cols, n_clusters, matrix_u, mode):
    if not mode:
        return matrix_u[np.random.choice(n_rows * n_cols, n_clusters)]
    else:
        grid = np.indices((n_rows, n_cols))
        row_indices, col_indices = grid[0], grid[1]

        indices = np.hstack((row_indices.reshape(-1, 1), col_indices.reshape(-1, 1)))

        n_points = n_rows * n_cols
        centers = [indices[np.random.choice(n_points, 1)[0]].tolist()]

        for _ in range(n_clusters - 1):
            distance = np.zeros(n_points)
            for idx, point in enumerate(indices):
                min_distance = np.Inf
                for center in centers:
                    dist = np.linalg.norm(point - center)
                    min_distance = dist if dist < min_distance else min_distance
                distance[idx] = min_distance

            distance /= np.sum(distance)
            centers.append(indices[np.random.choice(n_points, 1, p=distance)[0]].tolist())


        ### change from index to feature index
        for idx, center in enumerate(centers):
            centers[idx] = matrix_u[center[0] * n_rows + center[1], :]

        return np.array(centers)
```

Kmeans

And then, calculate the cluster of each point until the training converge or reach the maximun iteration times,

```python
def kmeans(n_rows, n_cols, n_clusters, matrix_u, centers, index, mode, cut):
    colors = np.array([[255, 0, 0], [0, 255, 0], [0, 0, 255]])

    if n_clusters > 3:
        colors = np.append(color, np.random.choice(256, (n_clusters - 3, 3)), axis=0)

    n_points = n_rows * n_cols
    img = []

    ### k means
    current_centers = centers.copy()
    new_cluster = np.zeros(n_points, dtype=int)
    count = 0
    iteration = 100
    while True:
        ### get new cluster
        new_cluster = kmeans_clustering(n_points, n_clusters, matrix_u, current_centers)

        ### get new centers
        new_centers = kmeans_recompute_centers(n_clusters, matrix_u, new_cluster)

        ### capture the new state
        img.append(capture_current_state(n_rows, n_cols, new_cluster, colors))

        if np.linalg.norm((new_centers - current_centers), ord=2) < 0.01 or count >= iteration:
            break

        current_centers = new_centers.copy()
        count += 1

    ### save as gif
    filename = f'./gifs/spectral_clustering/image{index}_cluster{n_clusters}_{"kmeans" if mode else "random"}_{"normalized" if cut else
"ratio"}.gif'
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    if len(img) > 1:
        img[0].save(filename, save_all=True, append_images=img[1:], optimize=False, loop=0, duration=100)
    else:
        img[0].save(filename)
```

```python
def kmeans_clustering(n_points, n_clusters, matrix_u, centers):
    new_clusters = np.zeros(n_points, dtype=int)

    for p in range(n_points):
        distance = np.zeros(n_clusters)
        for idx, center in enumerate(centers):
            distance[idx] = np.linalg.norm((matrix_u[p] - center), ord=2)

        new_clusters[p] = np.argmin(distance)

    return new_clusters


def kmeans_recompute_centers(n_clusters, matrix_u, current_cluster):
    new_centers = []
    for cluster in range(n_clusters):
        points_in_cluster = matrix_u[current_cluster == cluster]
        new_center = np.average(points_in_cluster, axis=0)
        new_centers.append(new_center)

    return np.array(new_centers)
```

Plot the Eignespace

This function is used to plot points into the eigenspace.

```python
def plot_result(matrix_u, clusters, index, mode, cut):
    color = ['r', 'b']
    plt.clf()

    for idx, point in enumerate(matrix_u):
        plt.scatter(point[0], point[1], c=color[clusters[idx]])

    ### save the plot
    filename = f'./gifs/spectral_clustering/eigenspace{index}_{"kmean" if mode else "random"}_{"normalized" if cut else "ratio"}.png'
    os.makedirs(os.path.dirname(filename), exist_ok=True)

    plt.savefig(filename)
```

Result

Kernel Kmeans

Image0

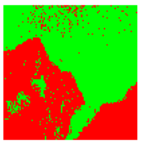
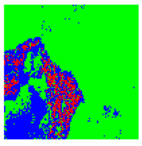
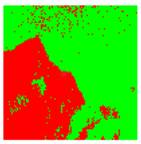
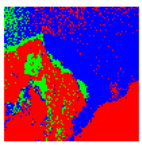
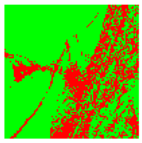
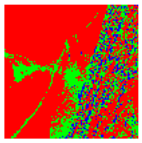
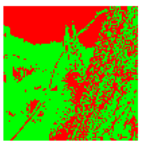
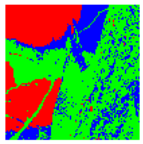
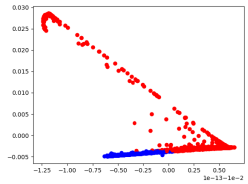
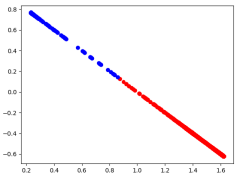
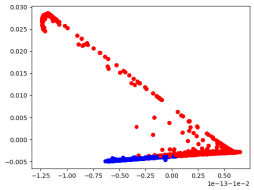
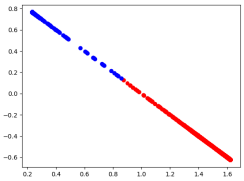
	2 clusters	Column 3
Random		
Kmeans++		

Image1

	2 clusters	Column 3
Random		
Kmeans++		

Spectral Clustering

IMAGE0

	Results	Eigenspace
2 clusters_Random_Ratio Cut		
2 clusters_Random_Normalized Cut		
2 clusters_Kmeans_Ratio Cut		
2 clusters_Kmeans_Normalized Cut		

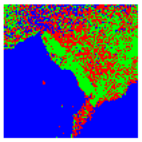
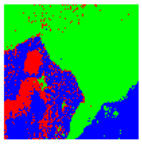
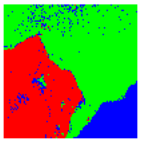
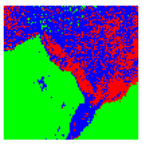
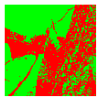
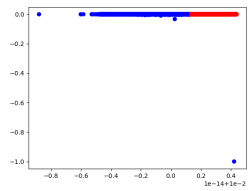
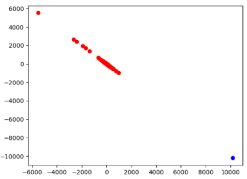
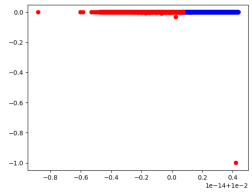
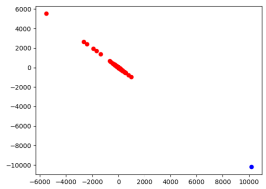
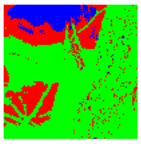
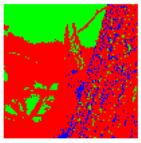
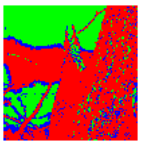
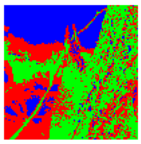
3 clusaters	Ratio cut	Normalized cut
Random		
Kmeans++		

IMAGE1

	Results	Eigenspace
2 clusters_Random_Ratio Cut		
2 clusters_Random_Normalized Cut		
2 clusters_Kmeans_Ratio Cut		
2 clusters_Kmeans_Normalized Cut		

3 clusaters	Ratio cut	Normalized cut
Random		
Kmeans++		

Observation

1. The classifiaction of these ways are not bad.
2. All of the used ways are unsupervised learning, the point will be clustered to some label randomly, therefore it may has different color at same point.
3. Kmeans++ is better than random