

Kernel K-means:

Goal: In this lab, we are supposed to implement two clustering methods

(1) Kernel K-means

(2) Spectral Clustering (including both normalized cut and ratio cut)

For both Kernel K-means and Spectral Clustering, use the defined kernel below to compute the Gram matrix:

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

This kernel multiplies two RBF kernels to simultaneously consider spatial similarity and color similarity, where:

(1) $S(x)$ represents the spatial information (i.e., the coordinate of the pixel).

(2) $C(x)$ represents the color information (i.e., the RGB values).

Code with detailed explanation:**Kernel**

```
def compute_kernel(image, gamma_s, gamma_c):
    n_rows, n_cols, _ = image.shape
    color_distance = cdist(image.reshape(n_rows * n_cols, 3), image.reshape(n_rows * n_cols, 3), 'sqeuclidean')
    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)))
    spatial_distance = cdist(indices, indices, 'sqeuclidean')
    return np.multiply(np.exp(-gamma_s * spatial_distance), np.exp(-gamma_c * color_distance))
```

The above function calculates the kernel matrix for an image by combining spatial and color similarities. It then computes the squared Euclidean distance between all pairs of pixels based on their color values to determine color similarity. Simultaneously, it creates a grid of pixel coordinates and computes the squared Euclidean distance between pixel pairs based on their spatial

coordinates to determine spatial similarity. The function then calculates the color similarity and spatial similarity using exponential functions scaled by factors γ_c and γ_s , respectively.

Centers Initialization

```
def choose_center(n_rows, n_cols, n_clusters, mode):
    if not mode:
        return np.random.choice(100, (n_clusters, 2))
    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())
        return np.array(centers)
```

In the center initialization part, there are two ways to determine the initial center of each cluster. The first method is random initialization, where a point is chosen randomly as the center. The second method uses the K-means++ strategy, which selects centers in a way that ensures they are spread out by choosing each new center with a preference for being further from the existing centers.

Clustering Initialization

```
def init_clustering(n_rows, n_cols, n_clusters, kernel, mode):
    centers = choose_center(n_rows, n_cols, n_clusters, mode)
    n_points = n_rows * n_cols
    cluster = np.zeros(n_points, dtype=int)
    for p in range(n_points):
        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]
        cluster[p] = np.argmin(distance)
    return cluster
```

After determining the center of each cluster, the next step is to assign all

points to a cluster by minimizing the distance between each point and the cluster center.

Kernel K-means

```
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, :] = 0
                tmp_kernel[:, p] = 0
        pairwise_distance[c] = np.sum(tmp_kernel)
    n_members[n_members == 0] = 1
    return pairwise_distance / n_members ** 2
```

The `get_sum_of_pairwise_distance` function calculates the sum of pairwise distances within each cluster using a kernel matrix, effectively setting distances to zero for points not in the current cluster. It will repeat many times until the training converges or reaches the maximum iteration time.

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

The `kernel_clustering` function assigns points to clusters by computing distances from each point to the cluster centers, utilizing the previously computed pairwise distances.

```

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))

def kernel_kmeans(n_rows, n_cols, n_clusters, cluster, kernel, mode, index):
    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)
    img = [capture_current_state(n_rows, n_cols, cluster, colors)]
    current_cluster = cluster.copy()
    count = 0
    iteration = 1000
    while True:
        new_cluster = kernel_clustering(n_rows * n_cols, n_clusters, kernel, current_cluster)
        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
    filename = f'./gifs/kernel_kmeans/image{index+1}_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)

```

The `capture_current_state` function captures the current clustering state and converts it to an image for visualization. After the training, it will save them as a .gif file.

Spectral Clustering:

Note that the calculation of kernel function is same as Kernel K-means.

Matrix U:

```

# Function to compute the Laplacian matrix and derive the eigenvectors (matrix U)
def compute_matrix_u(matrix_w, cut, n_clusters):
    # Compute the degree matrix D and Laplacian matrix L
    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
        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)

    # Compute eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(matrix_l)
    eigenvectors = eigenvectors.T

    # Sort eigenvalues and find indices of non-zero eigenvalues
    sort_idx = np.argsort(eigenvalues)
    sort_idx = sort_idx[eigenvalues[sort_idx] > 0]

    return eigenvectors[sort_idx[:n_clusters]].T

```

The `compute_matrix_u` function calculates the Laplacian matrix and derives the eigenvectors needed for spectral clustering. First, it constructs the degree matrix D and the Laplacian matrix L from the similarity matrix W .

Unnormalized Laplacian matrix:

$$L = D - S$$

Normalized spectral clustering:

$$L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$$

If the normalized cut is select, it adjusts L by computing the normalized Laplacian $L_{norm} = D^{-(1/2)} L D^{(1/2)}$. It then computes the eigenvalues and eigenvectors of L , transposes the eigenvectors, and sorts them based on the eigenvalues, excluding zero eigenvalues. Finally, it returns the top n eigenvectors, corresponding to the smallest non-zero eigenvalues, which form the matrix U for

clustering.

Spectral Clustering

```
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 clustering
    clusters = kmeans(n_rows, n_cols, n_clusters, matrix_u, centers, index, mode, cut)

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

Note that matrix U is composed of the eigenvectors of the Laplacian matrix, corresponding to the ordered eigenvalues. Spectral clustering uses matrix U to initialize the centers of each cluster and then performs K-means to obtain the final result after convergence. Finally, it plots the data points in the eigenspace for visualization.

Centers Initialization


```

def init_centers(n_rows, n_cols, n_clusters, matrix_u, mode):
    if not mode:
        # Random initialization
        return matrix_u[np.random.choice(n_rows * n_cols, n_clusters)]
    else:
        # K-means++ initialization
        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)

```

There also are two modes of initialization: random and K-means++ strategies. This function will return centers using coordinates in the eigenspace.

K-means

```

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(colors, np.random.choice(256, (n_clusters - 3, 3)), axis=0)

    n_points = n_rows * n_cols
    img = []

    # Initialize centers and cluster assignment
    current_centers = centers.copy()
    new_cluster = np.zeros(n_points, dtype=int)
    count = 0
    iteration = 100
    while True:
        # Assign clusters based on current centers
        new_cluster = kmeans_clustering(n_points, n_clusters, matrix_u, current_centers)
        # Recompute centers
        new_centers = kmeans_recompute_centers(n_clusters, matrix_u, new_cluster)
        # Capture the current state for visualization
        img.append(capture_current_state(n_rows, n_cols, new_cluster, colors))
        # Check for convergence
        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)

    return new_cluster

```

```

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)

```

It then calculates the cluster for each point until the training converges or reaches the maximum number of iterations.

Plot the eigenspace


```
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]])

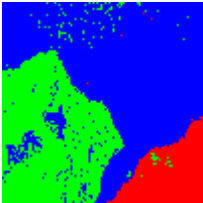
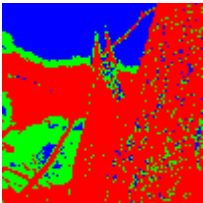
    # save the plot
    filename = f'./gifs/spectral_clustering/eigenspace{index}_{"kmean" if mode else "random"}_{"normalized" if cut else "ratio"}.png'
    os.makedirs(os.path.dirname(filename), exist_ok=True)

    plt.savefig(filename)
```

The function plots the points in the eigenspace for visualization.

Result:

Kernel K-means (n_cluster=3, random)

Image 1	
Image 2	

Spectral Clustering (n_cluster=2, random,)


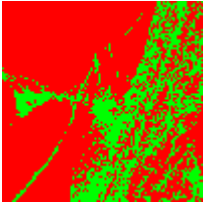
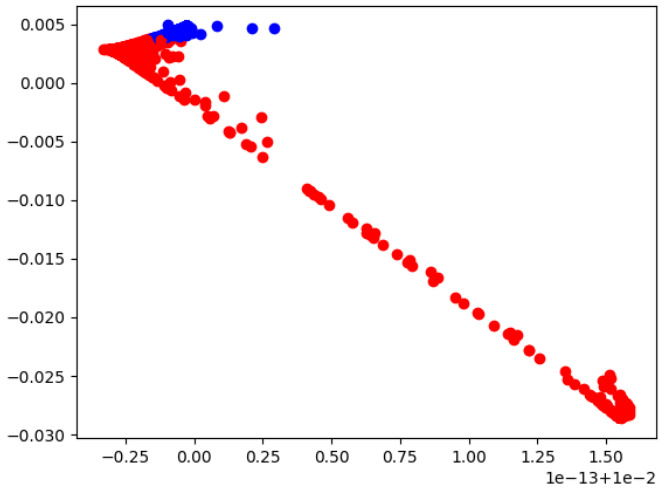
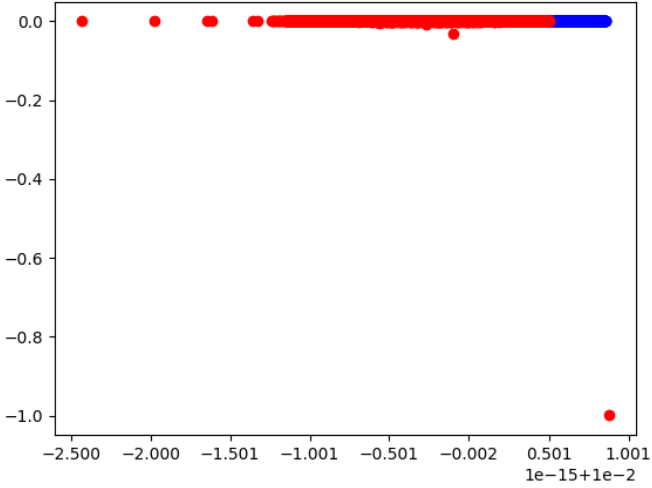
Image 1	
---------	--

Image 2	
---------	--

Eigenspace_random_ratio (k-means, ratio)

Image 1	
Image 2	

Observation

1. Classification of Methods: The classification of these clustering methods demonstrates a solid understanding of unsupervised learning techniques. The methods are well-categorized, highlighting their respective approaches and potential applications.

2. Unsupervised Learning: All the utilized methods are forms of unsupervised learning. In these approaches, data points are grouped into clusters based on their inherent similarities without the need for labeled training data. Consequently, the points are assigned to clusters with arbitrary labels, which means that the same point may be assigned different colors in different runs due to the randomness inherent in the initialization process.

3. Effectiveness of Initialization Strategies: Among the initialization strategies employed, K-means++ stands out as more effective compared to random initialization. K-means++ improves the convergence speed and the quality of the final clustering by spreading out the initial cluster centers more strategically, thereby avoiding poor clustering results that can arise from suboptimal initial conditions.