

ML HW6 Report

309552007 袁鈺勳

A. Code

a. Kernel K-means

1. Kernel

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

Based on the kernel function above, the code below builds the corresponding gram matrix.

```
# Get image shape
row, col, color = image.shape

# Compute color distance
color_distance = cdist(image.reshape(row * col, color), image.reshape(row * col, color), 'sqeuclidean')

# Get indices of a grid
grid = np.indices((row, col))
row_indices = grid[0]
col_indices = grid[1]

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

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

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

2. Initial Centers

Based on the value given by “-m” parameter from the command line, it will choose random strategy or kmeans++ strategy to get initial centers. The function will return centers using coordinates in the image.

```

if not mode:
    # Random strategy
    return np.random.choice(100, (num_of_clusters, 2))
else:
    # k-means++ strategy
    # Construct indices of a grid
    grid = np.indices((num_of_rows, num_of_cols))
    row_indices = grid[0]
    col_indices = grid[1]

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

    # Randomly pick first center
    num_of_points = num_of_rows * num_of_cols
    centers = [indices[np.random.choice(num_of_points, 1)[0]].tolist()]

    # Find remaining centers
    for _ in range(num_of_clusters - 1):
        # Compute minimum distance for each point from all found centers
        distance = np.zeros(num_of_points)
        for idx, point in enumerate(indices):
            min_distance = np.Inf
            for cen in centers:
                dist = np.linalg.norm(point - cen)

                min_distance = dist if dist < min_distance else min_distance
            distance[idx] = min_distance
        # Divide the distance by its sum to get probability
        distance /= np.sum(distance)
        # Get a new center
        centers.append(indices[np.random.choice(num_of_points, 1, p=distance)[0]].tolist())

    return np.array(centers)

```

3. Initial Clustering

After obtaining initial centers from “choose_center” function, it will classify each point according to the minimum distance in feature space from the point to the found centers.

```

# Get initial centers
centers = choose_center(num_of_rows, num_of_cols, num_of_clusters, mode)

# k-means
num_of_points = num_of_rows * num_of_cols
cluster = np.zeros(num_of_points, dtype=int)
for p in range(num_of_points):
    # Compute the distance of every point to all centers
    distance = np.zeros(num_of_clusters)
    for idx, cen in enumerate(centers):
        seq_of_cen = cen[0] * num_of_rows + cen[1]
        distance[idx] = kernel[p, p] + kernel[seq_of_cen, seq_of_cen] - 2 * kernel[p, seq_of_cen]
    # Pick the index of minimum distance as the cluster of the point
    cluster[p] = np.argmin(distance)

return cluster

```

4. Kernel K-means

Then, it can perform kernel k-means by using initial cluster. In each round, it use “kernel_clustering” function to get new cluster and

capture it. If the difference between new cluster and the old one is small enough or number of rounds reaches the bound, it will break the loop. That means it converges.

```
# Kernel k-means
current_cluster = cluster.copy()
count = 0
iteration = 100
while True:
    # Display progress
    progress_log(count, iteration)

    # Get new cluster
    new_cluster = kernel_clustering(num_of_rows * num_of_cols, num_of_clusters, kernel, current_cluster)

    # Capture new state
    img.append(capture_current_state(num_of_rows, num_of_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
```

$$\mathbf{k}(x_j, x_j) - \frac{2}{|C_k|} \sum_n \alpha_{kn} \mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2} \sum_p \sum_q \alpha_{kp} \alpha_{kq} \mathbf{k}(x_p, x_q)$$

The image below is “kernel_clustering” function. It is built upon the equation above.

```
# Get number of members in each cluster
num_of_members = np.array([np.sum(np.where(cluster == c, 1, 0)) for c in range(num_of_clusters)])

# Get sum of pairwise kernel distances of each cluster
pairwise = get_sum_of_pairwise_distance(num_of_points, num_of_clusters, num_of_members, kernel, cluster)

new_cluster = np.zeros(num_of_points, dtype=int)
for p in range(num_of_points):
    distance = np.zeros(num_of_clusters)
    for c in range(num_of_clusters):
        distance[c] += kernel[p, p] + pairwise[c]

    # Get distance from given data point to others in the target cluster
    dist_to_others = np.sum(kernel[p, :][np.where(cluster == c)])
    distance[c] -= 2.0 / num_of_members[c] * dist_to_others
    new_cluster[p] = np.argmin(distance)

return new_cluster
```

The image below is “get_sum_of_pairwise_distance” function which is the last term in the equation.

```

pairwise = np.zeros(num_of_clusters)
for c in range(num_of_clusters):
    tmp_kernel = kernel.copy()
    for p in range(num_of_points):
        # Set distance to 0 if the point doesn't belong to the cluster
        if cluster[p] != c:
            tmp_kernel[p, :] = 0
            tmp_kernel[:, p] = 0
    pairwise[c] = np.sum(tmp_kernel)

# Avoid division by 0
num_of_members[num_of_members == 0] = 1

return pairwise / num_of_members ** 2

```

5. Capture Current Clusters

In each round, the state of the current clusters will be recorded as an image. Each point will be marked with different colors according to its class.

```

state = np.zeros((num_of_rows * num_of_cols, 3))

# Give every point a color according to its cluster
for p in range(num_of_rows * num_of_cols):
    state[p, :] = colors[cluster[p], :]

state = state.reshape((num_of_rows, num_of_cols, 3))

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

```

6. Save gif

After the convergence, images will be saved as a gif file.

```

# Save gif
print()
filename = f'./output/kernel_kmeans/kernel_kmeans_{index}_' \
    f'cluster{num_of_clusters}_' \
    f'{"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)

```

b. Spectral Clustering

1. Kernel

Gram matrix is calculated by “compute_kernel” function from kernel kmeans.

2. Matrix U Containing Eigenvectors

Form the matrix $T \in \mathbb{R}^{n \times k}$ from U by normalizing the rows to norm 1,

Based on the image above, rows in the matrix U need to be normalized if normalized cut is used. The image below is the corresponding code.

```
# Get matrix U containing eigenvectors
info_log('≡ Calculate matrix U ≡')
m_u = compute_matrix_u(gram_matrix, cu, clu)
if cu:
    # Normalized cut
    sum_of_each_row = np.sum(m_u, axis=1)
    for j in range(len(m_u)):
        m_u[j, :] /= sum_of_each_row[j]
```

$$L = D - W \quad \text{normalized Laplacian } L_{\text{sym}} D^{-1/2} L D^{-1/2}$$

According to the images above, the Laplacian matrix L calculated from weight matrix W and degree matrix D will be transformed into normalized Laplacian matrix L_{sym} if normalized cut is used. Then, it finds the eigenvalues and eigenvectors of matrix L and returns matrix U containing first k eigenvectors with zero eigenvalues.

```

# Get 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 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 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[:num_of_clusters]].T

```

3. Spectral Clustering

It uses matrix U to get initial centers and perform k-means to get result cluster after convergence. Then, it plots the data points in eigenspace.

```

# Find initial centers
info_log('=== Find initial centers of each cluster ===')
centers = initial_centers(num_of_rows, num_of_cols, num_of_clusters, matrix_u, mode)

# K-means
info_log('=== K-means ===')
clusters = kmeans(num_of_rows, num_of_cols, num_of_clusters, matrix_u, centers, index, mode, cut)

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

```

4. Initial Centers

Based on the value given by “-m” parameter from the command line, it will choose random strategy or kmeans++ strategy to get initial centers. The function will return centers using coordinates in the eigenspace.

```

if not mode:
    # Random strategy
    return matrix_u[np.random.choice(num_of_rows * num_of_cols, num_of_clusters)]
else:
    # k-means++ strategy
    # Construct indices of a grid
    grid = np.indices((num_of_rows, num_of_cols))
    row_indices = grid[0]
    col_indices = grid[1]

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

    # Randomly pick first center
    num_of_points = num_of_rows * num_of_cols
    centers = [indices[np.random.choice(num_of_points, 1)[0]].tolist()]

    # Find remaining centers
    for _ in range(num_of_clusters - 1):
        # Compute minimum distance for each point from all found centers
        distance = np.zeros(num_of_points)
        for idx, point in enumerate(indices):
            min_distance = np.Inf
            for cen in centers:
                dist = np.linalg.norm(point - cen)
                min_distance = dist if dist < min_distance else min_distance
            distance[idx] = min_distance
        # Divide the distance by its sum to get probability
        distance /= np.sum(distance)
        # Get a new center
        centers.append(indices[np.random.choice(num_of_points, 1, p=distance)[0]].tolist())

    # Change from index to feature index
    for idx, cen in enumerate(centers):
        centers[idx] = matrix_u[cen[0] * num_of_rows + cen[1], :]

    return np.array(centers)

```

5. K-means

Then, it can perform k-means by using initial centers. In each round, it uses “kmeans_clustering” function to get new cluster and capture it. It also uses “kmeans_recompute_centers” function to update centers. If the difference between new center and the old one is small enough or number of rounds reaches the bound, it will break the loop. That means it converges.

```
# K-means
current_centers = centers.copy()
new_cluster = np.zeros(num_of_points, dtype=int)
count = 0
iteration = 100
while True:
    # Display progress
    progress_log(count, iteration)

    # Get new cluster
    new_cluster = kmeans_clustering(num_of_points, num_of_clusters, matrix_u, current_centers)

    # Get new centers
    new_centers = kmeans_recompute_centers(num_of_clusters, matrix_u, new_cluster)

    # Capture new state
    img.append(capture_current_state(num_of_rows, num_of_cols, new_cluster, colors))

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

    # Update current parameters
    current_centers = new_centers.copy()
    count += 1
```

In “kmeans_clustering”, it classifies data point into clusters based on the minimum distance in eigenspace from the point to the current centers.

```
new_cluster = np.zeros(num_of_points, dtype=int)
for p in range(num_of_points):
    # Find minimum distance from data point to centers
    distance = np.zeros(num_of_clusters)
    for idx, cen in enumerate(centers):
        distance[idx] = np.linalg.norm((matrix_u[p] - cen), ord=2)
    # Classify data point into cluster according to the closest center
    new_cluster[p] = np.argmin(distance)

return new_cluster
```

In “kmeans_recompute_centers”, it computes the mean in each cluster as new center.

```
new_centers = []
for c in range(num_of_clusters):
    points_in_c = matrix_u[current_cluster == c]
    new_center = np.average(points_in_c, axis=0)
    new_centers.append(new_center)

return np.array(new_centers)
```

6. Capture Current Clusters & Save gif

Current clusters are captured and gif file is saved in the same way as

kernel k-means.

7. Plot Data Points in the Eigenspace

Data points are scattered in the eigenspace and marked with different colors according to their colors.

```
colors = ['r', 'b']
plt.clf()

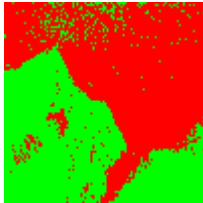
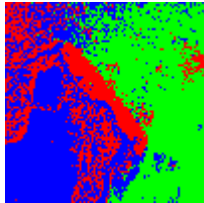
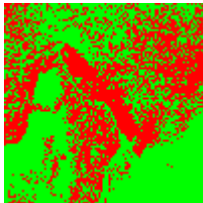
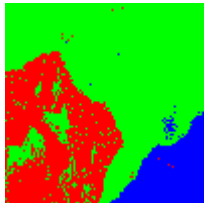
for idx, point in enumerate(matrix_u):
    plt.scatter(point[0], point[1], c=colors[clusters[idx]])

# Save the figure
filename = f'./output/spectral_clustering/eigenspace_{index}_' \
    f'{"kmeans++" if mode else "random"}_' \
    f'{"normalized" if cut else "ratio"}.png'
os.makedirs(os.path.dirname(filename), exist_ok=True)
plt.savefig(filename)
```

B. Results

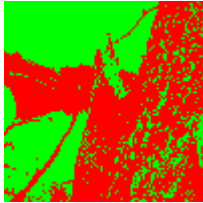
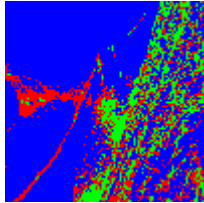
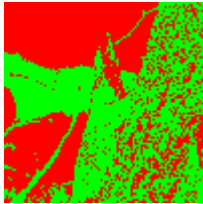
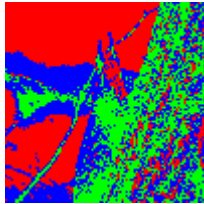
a. Kernel K-means

1. Image 1

	2 clusters	3 cluster
Random		
Kmeans++		

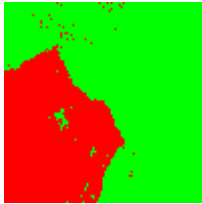
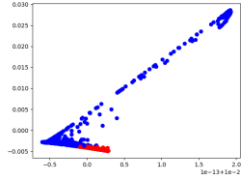
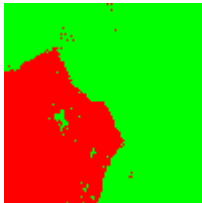
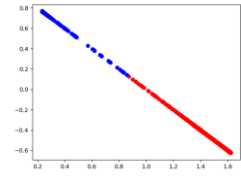
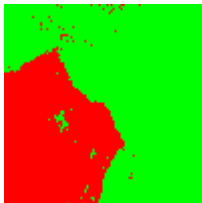
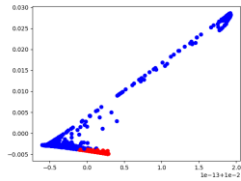
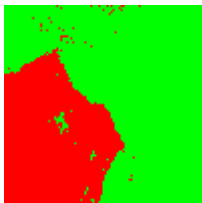
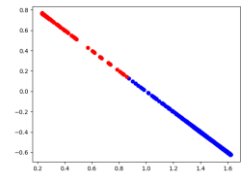
2. Image 2

	2 clusters	3 clusters
--	------------	------------

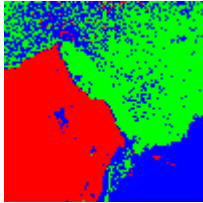
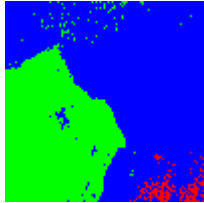
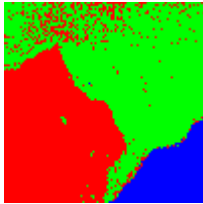
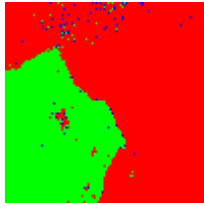
Random		
Kmeans++		

b. Spectral Clustering

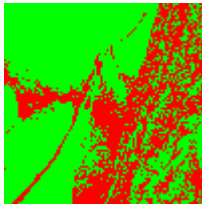
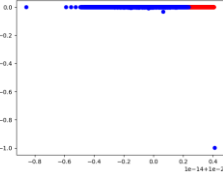
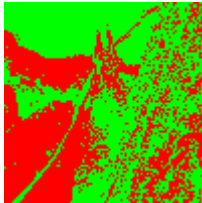
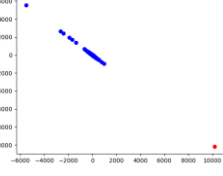
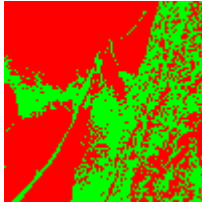
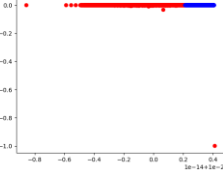
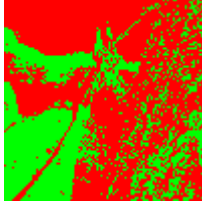
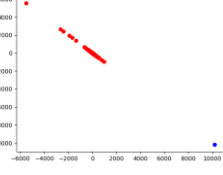
1. Image 1

2 clusters Random Ratio cut		
2 clusters Random Normalized cut		
2 clusters Kmeans++ Ratio cut		
2 clusters Kmeans++ Normalzied cut		

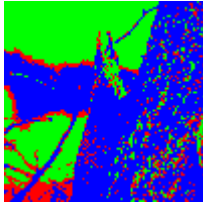
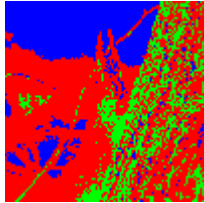
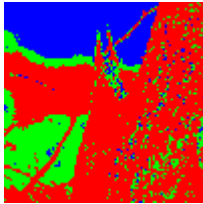
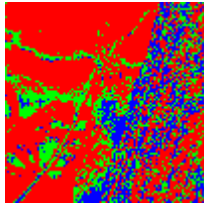
3 clusters	Ratio cut	Normalized cut
------------	-----------	----------------

Random		
Kmeans++		

2. Image 2

2 clusters Random Ratio cut		
2 clusters Random Normalized cut		
2 clusters Kmeans++ Ratio cut		
2 clusters Kmeans++ Normalized cut		

3 clusters	Ratio cut	Normalized cut
------------	-----------	----------------

Random		
Kmeans++		

C. Observations

- Overall, the classification is not bad. Although there are some misclassifications.
- Because this is unsupervised learning, it doesn't have correct labels. The classifier has to use similarity to classify each points into clusters. Therefore, it sometimes misclassifies data into the cluster to which we don't think it should belong. Take image 1 with 2 clusters and random initialization for example. The button-right corner of the image is classified in the same cluster as the land part in the image, but that region is is part of the ocean. The reason behind misclassification might be the similarity of color.
- Kmeans++ strategy can get better initial clustering than those using random strategy.
- It seems the classifier hardly classifies data points using normalized cut. Most data points are classified into the same cluster using normalized cut.