ML Homework 6 Report

- Student Info
  1. Student ID: 310555024
  2. Student Name: 林廷翰

- Code
  1. Kernel K-means
     - Common Parts
       1. Load Data

```python
def readImagesInNpArray():
    # Read the image files.
    images = [Image.open(filePathOfImage1), Image.open(filePathOfImage2)]

    # Convert image to numpy array.
    images[0] = np.asarray(images[0])
    images[1] = np.asarray(images[1])

    return images
```

The objective of the `readImagesINpArray` function is to load data from the input file, it will be called in the main function.

       2. Compute Kernel

```python
def computeKernel(image, gammaS, gammaC):
    # Get image shape.
    rows, cols, colors = image.shape

    # Compute the color distance.
    numOfPixels = rows * cols
    colorDist = cdist(image.reshape(numOfPixels, colors), image.reshape(numOfPixels, colors), 'sqeuclidean')

    # Compute the indices of a grid.
    indices = np.indices((rows, cols))
    indicesOfRow = indices[0]
    indicesOfCol = indices[1]

    # Compute the indices vector.
    indicesVector = np.hstack((indicesOfRow.reshape(-1, 1), indicesOfCol.reshape(-1, 1)))

    # Compute the spatial distance.
    spatialDist = cdist(indicesVector, indicesVector, 'sqeuclidean')

    # The kernel formula in spec.
    return np.multiply(np.exp(-gammaS * spatialDist), np.exp(-gammaC * colorDist))
```

The function is to compute the gram matrix based on the kernel function defined in spec, and the formula is following pic.

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

- Part 1 (2-clusters / randomly init)

  1. Init Center

```python
def initCenters(numOfRows, numOfCols, numOfClusters, initMode):
    if initMode == 0:
        # Random strategy.
        return np.random.choice(100, (numOfClusters, 2))
```

For the init center code of part 1, I just initialize the center randomly according to the number of number of clusters.

  2. Init Clusters

```python
def initClusters(numOfRows, numOfCols, numOfClusters, kernel, initMode):
    # Init centers.
    centers = initCenters(numOfRows, numOfCols, numOfClusters, initMode)

    # K-means.
    numOfPixels = numOfRows * numOfCols
    clusters = np.zeros(numOfPixels, dtype=int)

    for pixel in range(numOfPixels):
        # Compute the distance of every pixel to all centers.
        distance = np.zeros(numOfClusters)

        for index, center in enumerate(centers):
            seqOfCenter = center[0] * numOfRows + center[1]
            distance[index] = kernel[pixel, pixel] + kernel[seqOfCenter, seqOfCenter] - 2 * kernel[pixel, seqOfCenter]
        # Pick the index of minimum distance as the cluster of the point
        clusters[pixel] = np.argmin(distance)

    return clusters
```

After get the init center from `initCenter` function, We will classify all the pixels based on the min distance in feature space (each point to the center).

  3. Kernel K-means

```python
    # Kernel k-means.
    currentClusters = clusters.copy()
    count = 0
    iteration = 100

    while True:
        # Compute new clusters.
        numOfPixels = numOfRows * numOfCols
        newClusters = kernelClustering(numOfPixels, numOfClusters, kernel, currentClusters)

        # Get the image state.
        imageState = getCurrentImageState(numOfRows, numOfCols, newClusters)
        imageStates.append(imageState)

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

        currentClusters = newClusters.copy()
        count += 1
```

After getting the init clusters, we start to perform kernel k-means. In each round, we will perform `kernelCluster` function to get new clusters and calculate the difference between the `currentClusters` and `newClusterss` to check if it is already converged.

4. Kernel Clustering

```python
def kernelClustering(numOfPixels, numOfClusters, kernel, clusters):
    # Get number of members in each cluster
    numOfMembers = np.array([np.sum(np.where(clusters == c, 1, 0)) for c in range(numOfClusters)])

    # Get sum of pairwise kernel distances of each cluster
    pairwise = getSumOfPairwiseDistance(numOfPixels, numOfClusters, numOfMembers, kernel, clusters)

    newClusters = np.zeros(numOfPixels, dtype=int)
    for p in range(numOfPixels):
        distance = np.zeros(numOfClusters)
        for c in range(numOfClusters):
            distance[c] += kernel[p, p] + pairwise[c]

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

    return newClusters
```

We perform the `kernelClustering` based on the following pic formula. It the end, it will return the new clusters in array form.

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

- Part 2 (3-clusters / randomly init)

  1. Init Center

```python
def initCenters(numOfRows, numOfCols, numOfClusters, initMode):
    if initMode == 0:
        # Random strategy.
        return np.random.choice(100, (numOfClusters, 2))
```

It is same to the init center code of part 1.

  2. Init Clusters

```python
def initClusters(numOfRows, numOfCols, numOfClusters, kernel, initMode):
    # Init centers.
    centers = initCenters(numOfRows, numOfCols, numOfClusters, initMode)

    # K-means.
    numOfPixels = numOfRows * numOfCols
    clusters = np.zeros(numOfPixels, dtype=int)

    for pixel in range(numOfPixels):
        # Compute the distance of every pixel to all centers.
        distance = np.zeros(numOfClusters)

        for index, center in enumerate(centers):
            seqOfCenter = center[0] * numOfRows + center[1]
            distance[index] = kernel[pixel, pixel] + kernel[seqOfCenter, seqOfCenter] - 2 * kernel[pixel, seqOfCenter]
        # Pick the index of minimum distance as the cluster of the point
        clusters[pixel] = np.argmin(distance)

    return clusters
```

It is same to the init cluster of part 1.

  3. Kernel K-means

```python
    # Kernel k-means.
    currentClusters = clusters.copy()
    count = 0
    iteration = 100

    while True:
        # Compute new clusters.
        numOfPixels = numOfRows * numOfCols
        newClusters = kernelClustering(numOfPixels, numOfClusters, kernel, currentClusters)

        # Get the image state.
        imageState = getCurrentImageState(numOfRows, numOfCols, newClusters)
        imageStates.append(imageState)

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

        currentClusters = newClusters.copy()
        count += 1
```

It is same to the kernel k-means part of part 1.

  4. Kernel Clustering

```python
def kernelClustering(numOfPixels, numOfClusters, kernel, clusters):
    # Get number of members in each cluster
    numOfMembers = np.array([np.sum(np.where(clusters == c, 1, 0)) for c in range(numOfClusters)])

    # Get sum of pairwise kernel distances of each cluster
    pairwise = getSumOfPairwiseDistance(numOfPixels, numOfClusters, numOfMembers, kernel, clusters)

    newClusters = np.zeros(numOfPixels, dtype=int)
    for p in range(numOfPixels):
        distance = np.zeros(numOfClusters)
        for c in range(numOfClusters):
            distance[c] += kernel[p, p] + pairwise[c]

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

    return newClusters
```

It is same to the kernel clustering of part 1.

- Part 3 (2,3-clusters / k-means++)

1. Init Center

```python
    else:
        # k-means++ strategy.
        # Compute indices of a grid.
        indices = np.indices((numOfRows, numOfCols))
        indicesOfRow = indices[0]
        indicesOfCol = indices[1]

        # Compute the indices vector.
        indicesVector = np.hstack((indicesOfRow.reshape(-1, 1), indicesOfCol.reshape(-1, 1)))

        # Randomly pick first center.
        numOfPixels = numOfRows * numOfCols
        centers = [indicesVector[np.random.choice(numOfPixels, 1)[0]].tolist()]

        # Find remaining centers.
        for _ in range(numOfClusters - 1):
            # Compute the min distance for each point to all found centers.
            distance = np.zeros(numOfPixels)

            for index, indice in enumerate(indicesVector):
                minDistance = np.Inf

                for center in centers:
                    dist = np.linalg.norm(indice - center)
                    minDistance = dist if dist < minDistance else minDistance
                distance[index] = minDistance

            # Divide the distance by its sum.
            distance /= np.sum(distance)
            # Compute the new center and append to centers array.
            centers.append(indicesVector[np.random.choice(numOfPixels, 1, p=distance)[0]].tolist())

        return np.array(centers)
```

For the init center code of part 3, I choose the init center based on kmeans++ strategy. The function will return centers in array form.

2. Init Clusters

```python
def initClusters(numOfRows, numOfCols, numOfClusters, kernel, initMode):
    # Init centers.
    centers = initCenters(numOfRows, numOfCols, numOfClusters, initMode)

    # K-means.
    numOfPixels = numOfRows * numOfCols
    clusters = np.zeros(numOfPixels, dtype=int)

    for pixel in range(numOfPixels):
        # Compute the distance of every pixel to all centers.
        distance = np.zeros(numOfClusters)

        for index, center in enumerate(centers):
            seqOfCenter = center[0] * numOfRows + center[1]
            distance[index] = kernel[pixel, pixel] + kernel[seqOfCenter, seqOfCenter] - 2 * kernel[pixel, seqOfCenter]
        # Pick the index of minimum distance as the cluster of the point
        clusters[pixel] = np.argmin(distance)

    return clusters
```

It is same to the init cluster of part 1.

3. Kernel K-means

```python
    # Kernel k-means.
    currentClusters = clusters.copy()
    count = 0
    iteration = 100

    while True:
        # Compute new clusters.
        numOfPixels = numOfRows * numOfCols
        newClusters = kernelClustering(numOfPixels, numOfClusters, kernel, currentClusters)

        # Get the image state.
        imageState = getCurrentImageState(numOfRows, numOfCols, newClusters)
        imageStates.append(imageState)

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

        currentClusters = newClusters.copy()
        count += 1
```

It is same to the kernel k-means part of part 1.

4. Kernel Clustering

```python
def kernelClustering(numOfPixels, numOfClusters, kernel, clusters):
    # Get number of members in each cluster
    numOfMembers = np.array([np.sum(np.where(clusters == c, 1, 0)) for c in range(numOfClusters)])

    # Get sum of pairwise kernel distances of each cluster
    pairwise = getSumOfPairwiseDistance(numOfPixels, numOfClusters, numOfMembers, kernel, clusters)

    newClusters = np.zeros(numOfPixels, dtype=int)
    for p in range(numOfPixels):
        distance = np.zeros(numOfClusters)
        for c in range(numOfClusters):
            distance[c] += kernel[p, p] + pairwise[c]

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

    return newClusters
```

It is same to the kernel clustering of part 1.

- Common Parts

  1. Output the GIF Results

```python
    # Output the gif result.
    filename = f'./output/kernel_kmeans/kernel_kmeans_{index}_' \
               f'cluster{numOfClusters}_' \
               f'{"kmeans++" if initMode else "random"}.gif'
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    imageStates[0].save(filename, save_all=True, append_images=imageStates[1:], optimize=False, loop=0, duration=100)
```

After converging, we will output the result with gif pics.

2. Spectral Clustering

  - Common Parts

    1. Load Data

```python
def readImagesInNpArray():
    # Read the image files.
    images = [Image.open(filePathOfImage1), Image.open(filePathOfImage2)]

    # Convert image to numpy array.
    images[0] = np.asarray(images[0])
    images[1] = np.asarray(images[1])

    return images
```

The objective of the `readImagesINpArray` function is to load data from the input file, it will be called in the main function.

    2. Compute Kernel

```python
def computeKernel(image, gammaS, gammaC):
    # Get image shape.
    rows, cols, colors = image.shape

    # Compute the color distance.
    numOfPixels = rows * cols
    colorDist = cdist(image.reshape(numOfPixels, colors), image.reshape(numOfPixels, colors), 'sqeuclidean')

    # Compute the indices of a grid.
    indices = np.indices((rows, cols))
    indicesOfRow = indices[0]
    indicesOfCol = indices[1]

    # Compute the indices vector.
    indicesVector = np.hstack((indicesOfRow.reshape(-1, 1), indicesOfCol.reshape(-1, 1)))

    # Compute the spatial distance.
    spatialDist = cdist(indicesVector, indicesVector, 'sqeuclidean')

    # The kernel formula in spec.
    return np.multiply(np.exp(-gammaS * spatialDist), np.exp(-gammaC * colorDist))
```

We reuse the `computeKernel` function in kernel k-means program, so the explanation is same to the content in kernel k-means section.

- Part 1 (2-clusters / randomly init)

  1. Compute Matrix U

```python
# Compute matrixU which containing eigenvectors.
def computeMatrixU(matrixW, cutMode, numOfClusters):
    # Compute degree matrixD and Laplacian matrixL.
    matrixD = np.zeros_like(matrixW)
    for index, row in enumerate(matrixW):
        matrixD[index, index] += np.sum(row)
    matrixL = matrixD - matrixW

    # Normalized cut.
    if cutMode == 1:
        # Compute the normalized Laplacian matrixL.
        for idx in range(len(matrixD)):
            matrixD[idx, idx] = 1.0 / np.sqrt(matrixD[idx, idx])
        matrixL = matrixD.dot(matrixL).dot(matrixD)

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

    # Sort the eigenvalues and find indices of nonzero eigenvalues.
    sortedIdx = np.argsort(eigenvalues)
    sortedIdx = sortedIdx[eigenvalues[sortedIdx] > 0]

    return eigenvectors[sortedIdx[:numOfClusters]].T
```

According to the lecture PPT, we know that L = D - W. So we compute the `matrixD` first, and then compute the `matrixL` based on D and W. After obtaining the `matrixL` , we will based on the input parameter `cutMode` to determine if we need to normalize the `matrixL` . The we will find the eigenvectors (with to zero eigenvalues) as the return value (`matrixU`).

  2. Spectral Clustering

```python
def spectralClustering(numOfRows, numOfCols, numOfClusters, matrixU, initMode, cutMode, index):
    # Init centers.
    centers = initCenters(numOfRows, numOfCols, numOfClusters, matrixU, initMode)

    # K-means.
    clusters = kmeans(numOfRows, numOfCols, numOfClusters, matrixU, centers, index, initMode, cutMode)
```

We will use the `matrixU` to get the initial centers, and perform k-means to get the result cluster.

  3. Init Center

```python
def initCenters(numOfRows, numOfCols, numOfClusters, matrixU, initMode):
    if initMode == 1:
        # Random strategy.
        numOfPixels = numOfRows * numOfCols
        return matrixU[np.random.choice(numOfPixels, numOfClusters)]
```

For the init center code of part 1, I just initialize the center randomly according to the number of number of clusters and also the eigenspace.

4. K-means

```python
    # Kernel k-means.
    currentCenters = centers.copy()
    newClusters = np.zeros(numOfPixels, dtype=int)
    count = 0
    iteration = 100

    while True:
        # Compute new cluster.
        newClusters = kmeansClustering(numOfPixels, numOfClusters, matrixU, currentCenters)

        # Compute new centers.
        newCenters = kmeansRecomputeCenters(numOfClusters, matrixU, newClusters)

        # Get new state.
        imageStates.append(getCurrentImageState(numOfRows, numOfCols, newClusters))

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

        # Update current parameters.
        currentCenters = newCenters.copy()
        count += 1
```

We perform k-means based on initial centers. In each round, we use `kmeansClustering` to get new cluster, we also use the `kMeansRecomputeCenters` to update the centers. We will calculate the difference between the `currentCenters` and `newCenters` to check if it is already converged.

5. Output the GIF Results

```python
    # Output the gif result.
    filename = f'./output/spectral_clustering/spectral_clustering_{index}_' \
               f'cluster{numOfClusters}_' \
               f'{"kmeans++" if initMode else "random"}_' \
               f'{"normalized" if cutMode else "ratio"}.gif'
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    if len(imageStates) > 1:
        imageStates[0].save(filename, save_all=True, append_images=imageStates[1:], optimize=False, loop=0,
                            duration=100)
    else:
        imageStates[0].save(filename)
```

After converging, we will output the result with gif pics.

- Part 2 (3-clusters / randomly init)

1. Compute Matrix U

```python
# Compute matrixU which containing eigenvectors.
def computeMatrixU(matrixW, cutMode, numOfClusters):
    # Compute degree matrixD and Laplacian matrixL.
    matrixD = np.zeros_like(matrixW)
    for index, row in enumerate(matrixW):
        matrixD[index, index] += np.sum(row)
    matrixL = matrixD - matrixW

    # Normalized cut.
    if cutMode == 1:
        # Compute the normalized Laplacian matrixL.
        for idx in range(len(matrixD)):
            matrixD[idx, idx] = 1.0 / np.sqrt(matrixD[idx, idx])
        matrixL = matrixD.dot(matrixL).dot(matrixD)

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

    # Sort the eigenvalues and find indices of nonzero eigenvalues.
    sortedIdx = np.argsort(eigenvalues)
    sortedIdx = sortedIdx[eigenvalues[sortedIdx] > 0]

    return eigenvectors[sortedIdx[:numOfClusters]].T
```

It is same to the compute matrix U of part 1.

2. Spectral Clustering

```python
def spectralClustering(numOfRows, numOfCols, numOfClusters, matrixU, initMode, cutMode, index):
    # Init centers.
    centers = initCenters(numOfRows, numOfCols, numOfClusters, matrixU, initMode)

    # K-means.
    clusters = kmeans(numOfRows, numOfCols, numOfClusters, matrixU, centers, index, initMode, cutMode)
```

It is same to the spectral clustering of part 1.

3. Init Center

```python
def initCenters(numOfRows, numOfCols, numOfClusters, matrixU, initMode):
    if initMode == 1:
        # Random strategy.
        numOfPixels = numOfRows * numOfCols
        return matrixU[np.random.choice(numOfPixels, numOfClusters)]
```

It is same to the init center of part 1.

4. K-means

```python
    # Kernel k-means.
    currentCenters = centers.copy()
    newClusters = np.zeros(numOfPixels, dtype=int)
    count = 0
    iteration = 100

    while True:
        # Compute new cluster.
        newClusters = kmeansClustering(numOfPixels, numOfClusters, matrixU, currentCenters)

        # Compute new centers.
        newCenters = kmeansRecomputeCenters(numOfClusters, matrixU, newClusters)

        # Get new state.
        imageStates.append(getCurrentImageState(numOfRows, numOfCols, newClusters))

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

        # Update current parameters.
        currentCenters = newCenters.copy()
        count += 1
```

It is same to the k-means of part 1.

5. Output the GIF Results

```python
    # Output the gif result.
    filename = f'./output/spectral_clustering/spectral_clustering_{index}_' \
               f'cluster{numOfClusters}_' \
               f'{"kmeans++" if initMode else "random"}_' \
               f'{"normalized" if cutMode else "ratio"}.gif'
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    if len(imageStates) > 1:
        imageStates[0].save(filename, save_all=True, append_images=imageStates[1:], optimize=False, loop=0,
                            duration=100)
    else:
        imageStates[0].save(filename)
```

It is same to the output results of part 1.

- Part 3 (2,3-clusters / k-means++)

1. Compute Matrix U

```python
# Compute matrixU which containing eigenvectors.
def computeMatrixU(matrixW, cutMode, numOfClusters):
    # Compute degree matrixD and Laplacian matrixL.
    matrixD = np.zeros_like(matrixW)
    for index, row in enumerate(matrixW):
        matrixD[index, index] += np.sum(row)
    matrixL = matrixD - matrixW

    # Normalized cut.
    if cutMode == 1:
        # Compute the normalized Laplacian matrixL.
        for idx in range(len(matrixD)):
            matrixD[idx, idx] = 1.0 / np.sqrt(matrixD[idx, idx])
        matrixL = matrixD.dot(matrixL).dot(matrixD)

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

    # Sort the eigenvalues and find indices of nonzero eigenvalues.
    sortedIdx = np.argsort(eigenvalues)
    sortedIdx = sortedIdx[eigenvalues[sortedIdx] > 0]

    return eigenvectors[sortedIdx[:numOfClusters]].T
```

It is same to the compute matrix U of part 1.

2. Spectral Clustering

```python
def spectralClustering(numOfRows, numOfCols, numOfClusters, matrixU, initMode, cutMode, index):
    # Init centers.
    centers = initCenters(numOfRows, numOfCols, numOfClusters, matrixU, initMode)

    # K-means.
    clusters = kmeans(numOfRows, numOfCols, numOfClusters, matrixU, centers, index, initMode, cutMode)
```

It is same to the spectral clustering of part 1.

3. Init Center

```python
else:
    # k-means++ strategy.
    # Compute indices of a grid.
    indices = np.indices((numOfRows, numOfCols))
    indicesOfRow = indices[0]
    indicesOfCol = indices[1]

    # Compute the indices vector.
    indicesVector = np.hstack((indicesOfRow.reshape(-1, 1), indicesOfCol.reshape(-1, 1)))

    # Randomly pick first center.
    numOfPixels = numOfRows * numOfCols
    centers = [indices[np.random.choice(numOfPixels, 1)[0]].tolist()]

    # Find remaining centers.
    for _ in range(numOfClusters - 1):
        # Compute min distance for each point to all found centers.
        distance = np.zeros(numOfPixels)

        for index, indice in enumerate(indicesVector):
            minDistance = np.Inf

            for center in centers:
                dist = np.linalg.norm(indice - center)
                minDistance = dist if dist < minDistance else minDistance
            distance[index] = minDistance

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

    # Change from index to feature index.
    for index, center in enumerate(centers):
        centers[index] = matrixU[center[0] * numOfRows + center[1], :]

    return np.array(centers)
```

For the init center code of part 3, I choose the init center based on kmeans++ strategy. The function will return centers in array form based on the eigenspace (`matrixU`).

4. K-means

```python
# Kernel k-means.
currentCenters = centers.copy()
newClusters = np.zeros(numOfPixels, dtype=int)
count = 0
iteration = 100

while True:
    # Compute new cluster.
    newClusters = kmeansClustering(numOfPixels, numOfClusters, matrixU, currentCenters)

    # Compute new centers.
    newCenters = kmeansRecomputeCenters(numOfClusters, matrixU, newClusters)

    # Get new state.
    imageStates.append(getCurrentImageState(numOfRows, numOfCols, newClusters))

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

    # Update current parameters.
    currentCenters = newCenters.copy()
    count += 1
```

It is same to the k-means of part 1.

5. Output the GIF Results

```python
    # Output the gif result.
    filename = f'./output/spectral_clustering/spectral_clustering_{index}_' \
               f'cluster{numOfClusters}_' \
               f'{"kmeans++" if initMode else "random"}_' \
               f'{"normalized" if cutMode else "ratio"}.gif'
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    if len(imageStates) > 1:
        imageStates[0].save(filename, save_all=True, append_images=imageStates[1:], optimize=False, loop=0,
                            duration=100)
    else:
        imageStates[0].save(filename)
```

It is same to the output results of part 1.

- Part 4 (examine points)

1. Plot Result

```python
def plotResult(matrixU, clusters, index, initMode, cutMode):
    colors = ['r', 'b']
    plt.clf()

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

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

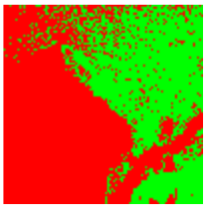
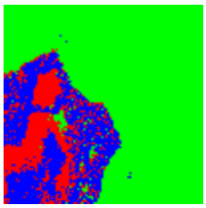
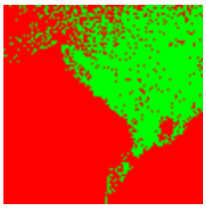
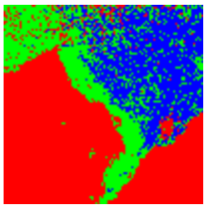
Capture the result to examine whether the data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian or not.

- Experiments & Discussion

1. Kernel K-means

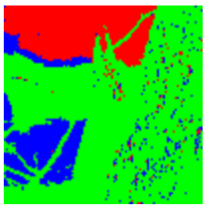
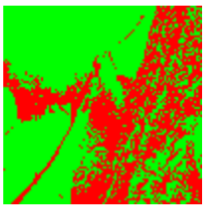
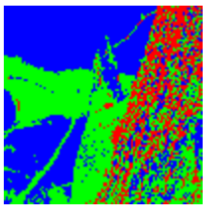
- Image 1

1. Result

	2 Clusters	3 Clusters
Randomly		
Kmeans++		

2. Discussion
    - Kmeans++ strategy can get better initial clustering.
    - For image 1, the better value of k is 2 (sea and island).

- Image 2
    1. Result

	2 Clusters	3 Clusters
Randomly		
Kmeans++		

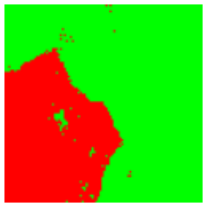
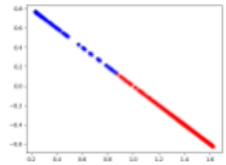
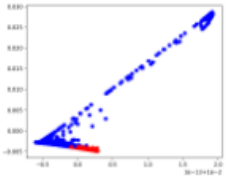
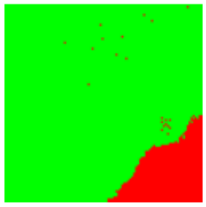
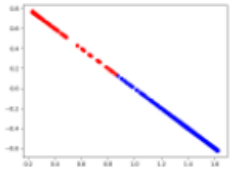
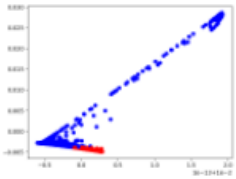
    2. Discussion

- For image 2, the better value of k is 3 (tree, rabbit, and background).

2. Spectral Clustering

- Image 1 / 2 Clusters

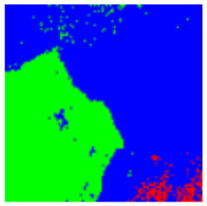
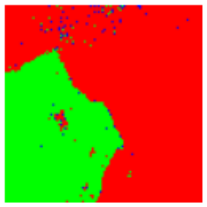
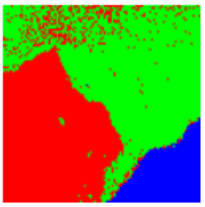
1. Result

	Result	Examine Points
2 Cluster Randomly Normalized Cut		
2 Cluster Randomly Ratio Cut		
2 Cluster Kmeans++ Normalized Cut		
2 Cluster Kmeans++ Ratio Cut		

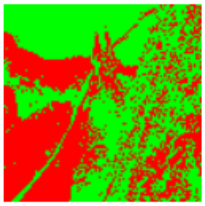
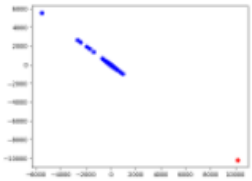
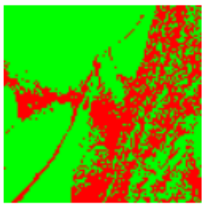
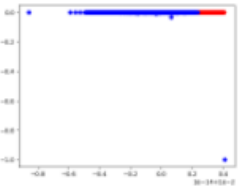
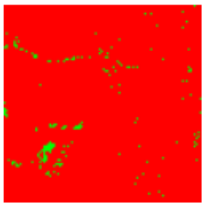
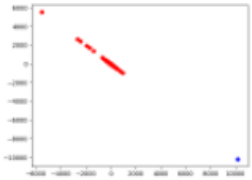
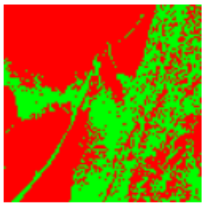
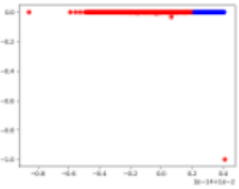
2. Discussion

- Kmeans++ strategy can get better initial clustering.

- Image 1 / 3 Clusters
  1. Result

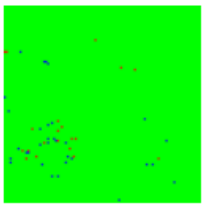
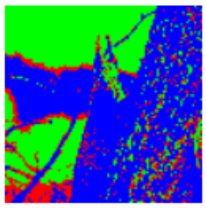
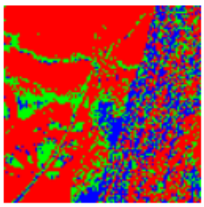
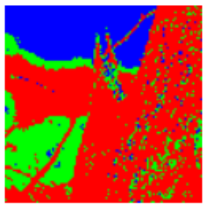
	Normalized Cut	Ratio Cut
Randomly		
Kmeans++		

  2. Discussion
     - For image 1, the better value of k is 2 (sea and island). In randomly, normalized cut, we can see that there are only some of parts with red color.
- Image 2 / 2 Clusters
  1. Result

	Result	Examine Points
2 Cluster Randomly Normalized Cut		
2 Cluster Randomly Ratio Cut		
2 Cluster Kmeans++ Normalized Cut		
2 Cluster Kmeans++ Ratio Cut		

2. Discussion
    - All of experiments result show that we can well separate the point in eigenspace.

- Image 2 / 3 Clusters
    1. Result

	Normalized Cut	Ratio Cut
Randomly		
Kmeans++		

2. Discussion
    - The randomly init is not good initially (we can get the result in the randomly init, normalized cut).

- Observations
    1. How to choose k for supervised learning?

       We may try the see the result first, and then decide the k value it should be. Because there is no general solution to find one.
    2. Kmeans++ strategy is better than randomly initializing (the initial result).
    3. We can see that it hardly classifies data points using normalized cut. Most of points are classified into same cluster.