Report

A. Code Explanation

- **Kernel K-Means**
  - **Load Image**

```python
IMAGE = "2"
image = np.asarray(Image.open(f"img/image{IMAGE}.png"))
```

I read image with Pillow package and convert it into a numpy array.

  - **Big Picture**

```python
def kernel_kmeans(
    image: np.ndarray,
    gamma_spatial: float,
    gamma_color: float,
    num_cluster: int,
    init_cluster: str,
    output_name: str
) -> None:

    # compute gram matrix
    kernel = compute_kernel(image, gamma_spatial, gamma_color)

    # clustering initialization
    initial_cluster = initialize_clustering(num_cluster, init_cluster, kernel)

    # clustering
    clustering(num_cluster, initial_cluster, kernel, output_name)
```

The kernel k-means can be broken down into three steps. The first step is computing gram matrix, which is defined in compute_kernel function. The second step is initializing the center of all clusters, which is defined in initialize_clustering function. The last step is executing kernel k-means algorithm, which is defined in clustering function.

  - **Compute Gram Matrix**

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

The computation of gram matrix is based on the above formula. The first part is RBF kernel of spatial information, and the second part is RBF kernel of color information.

```python
    # spatial similarity
    coordinate = []
    for row_pos in range(100):
        for col_pos in range(100):
            coordinate.append([row_pos, col_pos])
    coordinate = np.array(coordinate)
    spatial_similarity = cdist(coordinate, coordinate, "sqeuclidean")

    # color similarity
    color = np.reshape(image, (image.shape[0] * image.shape[1], image.shape[2]))
    color_similarity = cdist(color, color, 'sqeuclidean')

    # kernel function
    result = np.multiply(np.exp(-gamma_spatial * spatial_similarity), np.exp(-gamma_color * color_similarity))
```

Therefore, in the above code implementation, I store the coordinates and pixel values of all data points, calculate similarity (distance), and multiply two RBF kernels to get gram matrix.

- **Initialize Cluster (Random or K-Means++)**

  There are two ways to initialize center clusters, random initialization or k-means++. In k-means++, the center of first cluster is randomly initialized.

```python
        # choose first center randomly
        idx = np.random.choice(a=10000, size=1)[0]
        cluster_center.append(coordinate[idx])
```

After that, the remaining center of clusters will be determined based on the distance of each data points to its nearest cluster. If the data point is far away from the existing clusters, it is likely to selected as center of new cluster.

```python
        # find remaining centers
        for _ in range(num_cluster-1):

            # distance of each data point to its nearest cluster
            distance = np.zeros(10000)

            for idx, coord in enumerate(coordinate):

                min_distance = np.Inf
                for center in cluster_center:
                    dist = np.linalg.norm(np.array(coord) - np.array(center))
                    if dist < min_distance:
                        min_distance = dist

                distance[idx] = min_distance

            # convert distance into probability
            distance /= np.sum(distance)

            # choose the data point with higher probability as new center of
            # the data point is far away from all of existing centers
            idx = np.random.choice(10000, 1, p=distance)[0]
            cluster_center.append(coordinate[idx])

        cluster_center = np.array(cluster_center)
```

After determining the center of all clusters, each data point will be assigned to nearest cluster.

```
    for point_idx in range(10000):

        # compute the distance of current data point to all clusters
        distance = np.zeros(num_cluster)
        for idx, center in enumerate(cluster_center):
            center_idx = center[0] * 100 + center[1]
            distance[idx] = kernel[point_idx, point_idx] \
                + kernel[center_idx, center_idx] \
                - 2 * kernel[point_idx, center_idx]

        # Pick the index of minimum distance as the cluster of the point
        initial_cluster[point_idx] = np.argmin(distance)
```

The distance of data point between center of cluster in feature space follows this formula, ||ϕ(Xn) − ϕ(μn)|| = k(xn, xn) + k(μk, μk) − 2k(xn, μk).

- **Core Algorithm**

  In the core part of kernel k-means algorithm, the clustering process is executed up to 100 iterations. In each iteration, I calculate the distance between each data points and all center of clusters, and the data points are assigned to its nearest cluster.

```
        # compute distance of each data point to all cluster
        new_cluster = np.zeros(10000, dtype=np.int32)

        for point_idx in range(10000):
            distance = np.zeros(num_cluster)
            for cluster_idx in range(num_cluster):
                first_item = kernel[point_idx, point_idx]
                mask = np.where(current_cluster == cluster_idx)
                second_item = np.sum(kernel[point_idx, mask])
                second_item *= (2.0 / cluster_size[cluster_idx])
                third_item = cluster_pairwise[cluster_idx]
                distance[cluster_idx] = first_item - second_item + third_item

            new_cluster[point_idx] = np.argmin(distance)
```

The distance follows this formula:

$$d(x_i, m_j) = \underline{\kappa(x_i, x_i)} + \underline{\frac{2}{C_j}\sum_{x_l \in C_j} \kappa(x_i, x_l)} + \underline{\frac{1}{|C_j|^2}\sum_{x_l \in C_j}\sum_{x_s \in C_j} \kappa(x_l, x_s)}$$

The third item is the sum of distance between all data points in the cluster.

```python
        # calculate sum of pairwise kernel distance of each cluster
        cluster_pairwise = np.zeros(num_cluster)
        for cluster_idx in range(num_cluster):
            tmp_kernel = kernel.copy()
            for point_idx in range(10000):

                # if the data point not in this cluster, zero out its kernel distance
                if current_cluster[point_idx] != cluster_idx:
                    tmp_kernel[point_idx, :] = 0
                    tmp_kernel[:, point_idx] = 0

            cluster_pairwise[cluster_idx] = np.sum(tmp_kernel)
```

- **Convert Cluster Information into Image**

```python
def cluster2image(initial_cluster: np.ndarray, cluster_color: list) -> Image:
    image = np.zeros((100*100, 3))
    for pixel_idx in range(10000):
        pixel_cluster = initial_cluster[pixel_idx]
        pixel_color = cluster_color[pixel_cluster]
        image[pixel_idx, :] = pixel_color[:]
    image = np.reshape(image, (100, 100, 3))
    image = np.uint8(image)
    image = Image.fromarray(image)
    return image
```

In each iteration of kernel k-means, I save the cluster information as a single image. Each data point has its own color based on its cluster.

- **Convert List of Images into GIF**

```python
    # save list of images as a gif
    cluster_image[0].save(
        f"result/kernel_kmeans/{output_name}.gif",
        save_all=True,
        append_images=cluster_image[1:],
        duration=150,
        loop=0
    )
```

With Pillow package, it is easy to convert list of images into a single gif file.

- **Spectral Clustering**
  - **Compute Gram Matrix**
    The approach to computing gram matrix in spectral clustering is same as it in kernel k-means.

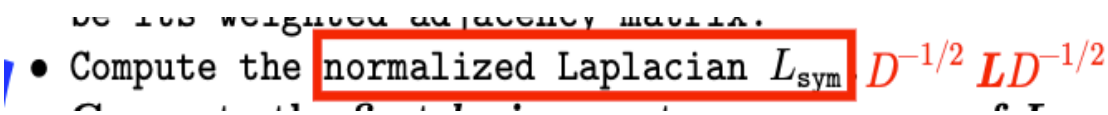
  - **Compute Laplacian Matrix**

```python
def compute_laplacian(matrix_w: np.ndarray) -> np.ndarray:

    # compute 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

    # compute matrix L (Laplacian)
    matrix_l = matrix_d - matrix_w

    return matrix_l, matrix_d
```

    With the weight matrix, I am able to compute diagonal matrix and Laplacian matrix.

  - **Normalize Laplacian Matrix (Normalized Cut Only)**

```python
    # normalize laplacian matrix if normalized cut
    if cut_way == "normalized":
        for r in range(matrix_d.shape[0]):
            matrix_d[r][r] = 1.0 / np.sqrt(matrix_d[r, r])
        matrix_l = np.matmul(np.matmul(matrix_d, matrix_l), matrix_d)
```



    It must be noted that if we use normalized cut in spectral clustering, we have to normalize Laplacian matrix. Therefore, I calculate the diagonal matrix to the power of -1/2, and multiply it with Laplacian matrix.

- **Eigen-Decomposition of Laplacian Matrix**

```
    # calculate eigenvalues and eigenvectors of laplacian matrix
    eigenvalues, eigenvectors = np.linalg.eig(matrix_l)
    eigenvectors = eigenvectors.T

    # find non-zero eigenvalues and sort it
    sort_idx = np.argsort(eigenvalues)
    mask = eigenvalues[sort_idx] > 0
    sort_idx = sort_idx[mask]

    # only keep first k eigenvectors
    sort_idx = sort_idx[:num_cluster]

    # get corresponding eigenvectors
    matrix_u = eigenvectors[sort_idx].T
```

I use numpy package to get eigenvalues and eigenvectors from Laplacian matrix, and only keep the first k non-zero eigenvalues with corresponding eigenvectors.

- **Normalize Eigenvectors (Normalized Cut Only)**
After the previous step, the matrix U will contain eigenvectors of Laplacian matrix. If we use normalized cut in spectral clustering, we have to normalize each row in matrix U.

```
    if cut_way == "normalized":
        for row_idx in range(matrix_u.shape[0]):
            matrix_u[row_idx, :] /= np.sum(matrix_u[row_idx, :])
```

- **Find Initial Center of Clusters**
There are two approaches to find the initial center of clusters, random and k-means++. The implementation of these two methods is same as it in kernel k-means. However, instead of returning the coordinate information, I transform the coordinate to row index, and return specific rows of matrix U (eigenspace).

```
        matrix_u_part = np.zeros((len(cluster_center), matrix_u.shape[1]))
        for idx, center in enumerate(cluster_center):
            center_idx = center[0] * 100 + center[1]
            matrix_u_part[idx][:] = matrix_u[center_idx][:]
```

- **Core Algorithm**
  In the core part of spectral clustering, the clustering process is executed up to 100 iterations. In each iteration, I calculate the distance between each data points and all center of clusters, and the data points are assigned to its nearest cluster.

```
        # caculate current cluster
        for point_idx in range(10000):

            # compute the distance of current data point to all clusters
            distance = np.zeros(num_cluster)
            for idx, center in enumerate(current_center):
                distance[idx] = np.linalg.norm((matrix_u[point_idx] - center), ord=2)

            # current data point belongs to the nearest cluster
            current_cluster[point_idx] = np.argmin(distance)
```

  After that, I calculate new center of all clusters.

```
        # update each cluster's center
        new_center = np.zeros_like(current_center)
        for cluster_idx in range(num_cluster):
            mask = (current_cluster == cluster_idx)
            cluster_point = matrix_u[mask]
            cluster_center = np.average(cluster_point, axis=0)
            new_center[cluster_idx][:] = cluster_center[:]
```

  If the centers do not change a lot, the clustering process ends in this iteration.

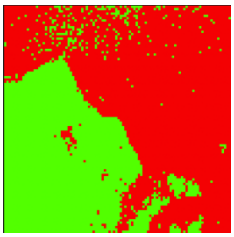
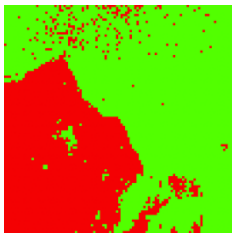
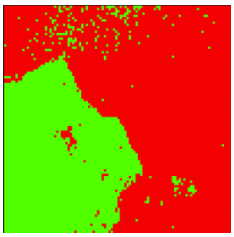
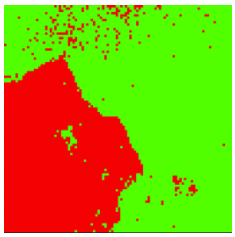
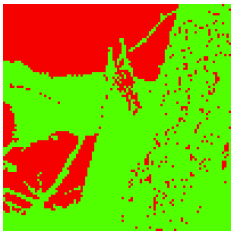
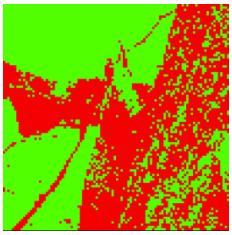
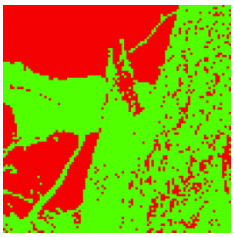
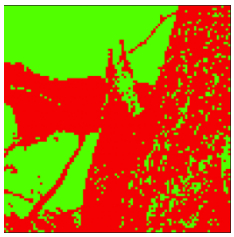
```
        if np.linalg.norm((new_center - current_center), ord=2) < 0.01:
            break
```

- **Convert Cluster Information into Image**
  In each iteration of spectral clustering, I save the cluster information as a single image. Each data point has its own color based on its cluster. The implementation is same as it in kernel k-means.

- **Convert List of Images into a Single GIF**
  The implementation is same in kernel k-means.

B. Experiment Result and Discussion

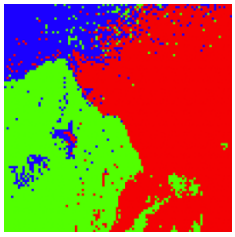
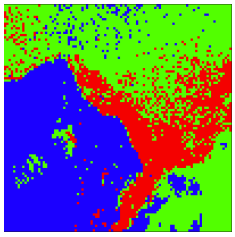
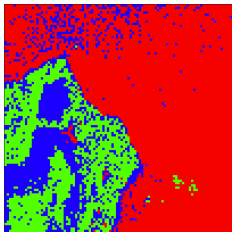
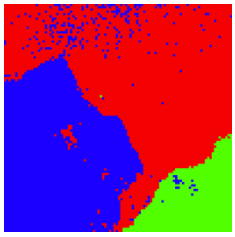
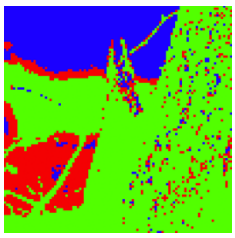
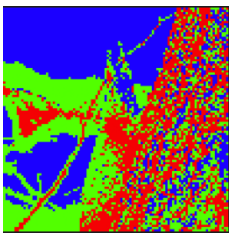
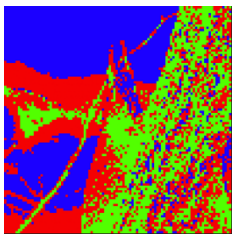
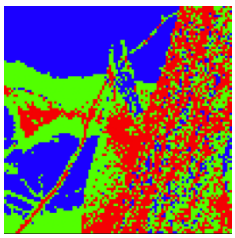
- **Kernel K-Means with 2 Clusters (Random vs K-Means++)**

		Random	K-Means++
Image 1	Iteration 0		
	Final		
Image 2	Iteration 0		
	Final		

From the experiment, I find something interesting:

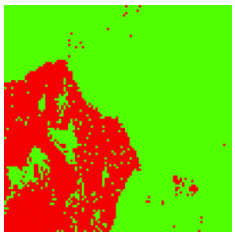
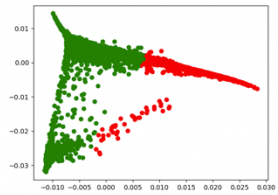
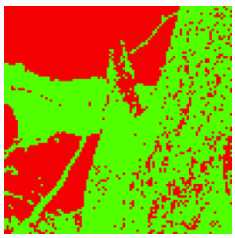
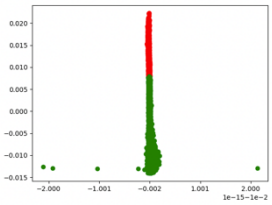
- Random initialization needs more iterations than kernel k-means++
- Random initialization tends to converge to bad result like the example in image 2

- **Kernel K-Means with 3 Clusters (Random vs K-Means++)**

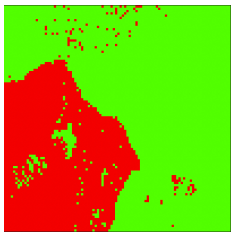
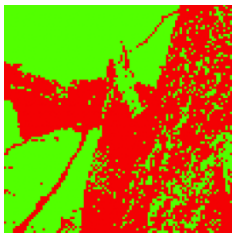
		Random	K-Means++
Image 1	Iteration 0		
	Final		
Image 2	Iteration 0		
	Final		

From the experiment, I find that when the number of clusters increases, the advantage of k-means++ is more significant. For example, k-means++ initialization make the model converge in better result.

- **Spectral Clustering with 2 Cluster, Ratio Cut, Random Initialization**

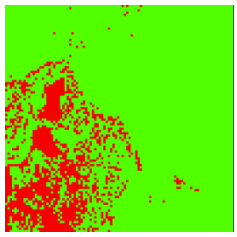
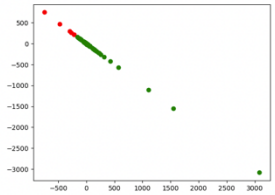
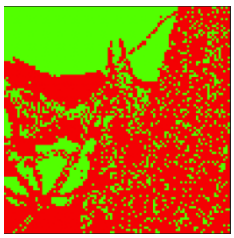
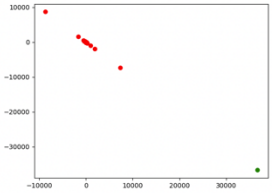
		Clustering Result	Eigenspace
Image 1	Final		
Image 2	Final		

- **Spectral Clustering with 2 Cluster, Ratio Cut, K-Means++ Initialization**

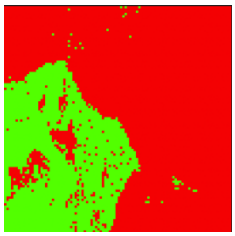
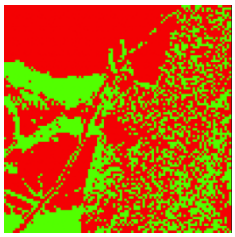
		Clustering Result	Eigenspace
Image 1	Final		Same as Above
Image 2	Final		Same as Above

When using spectral clustering with 2 clusters and ratio cut, I find that the k-means initialization is also better than random.

- **Spectral Clustering with 2 Cluster, Normalized Cut, Random Initialization**

		Clustering Result	Eigenspace
Image 1	Final		
Image 2	Final		

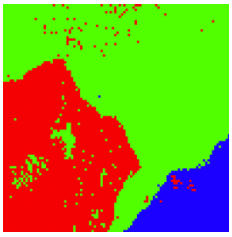
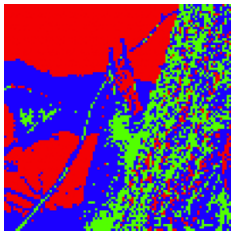
- **Spectral Clustering with 2 Cluster, Normalized Cut, K-Means++ Initialization**

		Clustering Result	Eigenspace
Image 1	Final		Same as Above
Image 2	Final		Same as Above

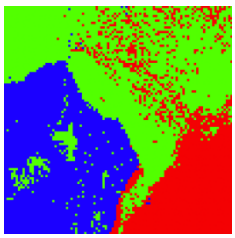
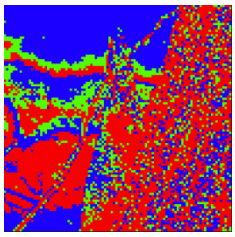
When using spectral clustering with 2 clusters and normalized cut, I find that the k-means initialization is still better than random. However, in terms of convergence, I

think ratio cut is better that normalized cut in this case because normalized cut tends to classify all many data points in one same cluster.

- **Spectral Clustering with 3 Cluster, Ratio Cut, K-Means++ Initialization**

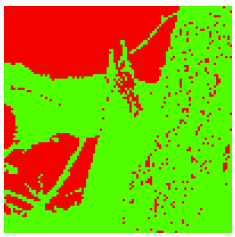
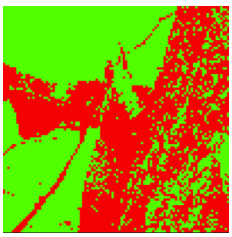
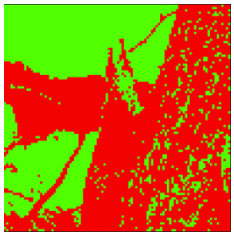
		Clustering Result	Eigenspace
Image 1	Final		Only for 2 Clsuters
Image 2	Final		Only for 2 Clsuters

- **Spectral Clustering with 3 Cluster, Normalized Cut, K-Means++ Initialization**

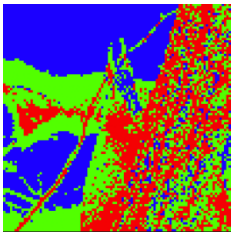
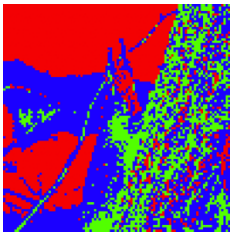
		Clustering Result	Eigenspace
Image 1	Final		Only for 2 Clsuters
Image 2	Final		Only for 2 Clsuters

C. Experiment Result and Discussion

- **Performance: Initialization Method**
  From experiments, I discover that the initialization method plays an important role in both kernel k-means and spectral clustering. For example, in the experiment, Kernel K-Means with 2 Clusters (Random vs K-Means++), k-means++ has better result than random initialization on image 2.

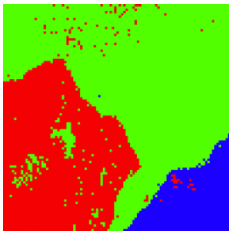
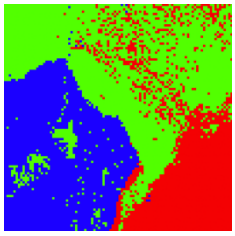
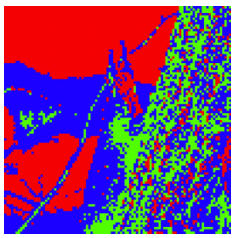
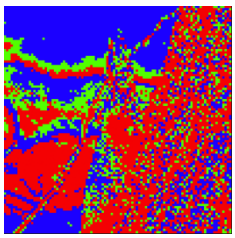
		Random	K-Means++
Image 2	Iteration 0		
	Final		

- **Performance: Clustering Algorithm**
  The left image is result of kernel k-means on image 2, and the right image is result of spectral clustering with ratio cut on same image. Both uses k-means++ as initialization method.

I find that it is hard to determine one approach outperforms the other because sometimes one way is better and sometimes the other way is better.

- **Performance: Different Cut in Spectral Clustering**

		Ratio Cut	Normalized Cut
Image 1	Final		
Image 2	Final		

In terms of different cut in spectral clustering, I find that ratio cut usually outperforms normalized cut because normalized cut is prone to generate “big” cluster. In other words, normalized cut usually includes more data points in one cluster.

- **Execution Time of Different Setting**
  From above experiments and implementation, I find:
  - Initialization: random < k-means++ (because k-means++ needs additional operations)
  - Clustering algorithm: kernel k-means < spectral clustering (because eigen-decomposition takes a lot of time)
  - Cut Method in Spectral Clustering: ratio cut < normalized cut (because normalized cut needs additional operations)