HW6 Kernel K-means and Spectral Clustering

1. Kernel K-means

In this part, I need to implement kernel k-means. To achieve this goal, I implement it with the following architecture:

1. Choose initial point:

   ```
   initial_points = choose_initial_points(args,img)
   ```

   We have two kinds of strategies to choose the initial point. The first one is to randomly choose the initial point in the image.

   ```
   if(args.choose_mode == "random"):
       return np.random.randint(low=0, high=10000, size=(args.n))
   ```

   The second one is referred to kmeans++. We hope that our initial point is as far apart as possible. Thus, after we randomly choose the random point.

   ```
   center = [ np.random.randint(low=0, high=10000) ]
   ```

   We would calculate the distance of each point in the grid between the center and we use $P = \frac{dist(center,point)}{\sum dist(center,point)}$ as our probability that each point can be selected as center points.

   ```
   # We want to choose the centroid the far the better
   grid = np.indices((img.shape[0],img.shape[1]))
   grid = np.stack(grid, axis=-1)
   indices = np.array([ i for i in range(10000) ])
   center = [ np.random.randint(low=0, high=10000) ]
   for i in range(args.n-1):
       dis = np.zeros((img.shape[0],img.shape[1]))
       min_distance = np.Inf
       arr = np.full((100, 100), 100000,dtype=float)
       for c in center:
           x = c //img.shape[0]
           y = c %img.shape[1]
           dis = np.sum((grid - np.array([x,y]))*(grid - np.array([x,y])),axis=2)
           dis = np.sqrt(dis)
           arr[dis<arr] = dis[dis<arr]
       arr = arr.reshape(100*100)
       arr /= np.sum(arr)

       center.append(indices[np.random.choice(100*100, 1, p=arr)[0]].tolist())
   ```

2. Construct **Gram matrix**

   After selecting initial points, We need to construct a gram matrix. We follow the instructions from spec. We use the following equation as our kernel.

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

```python
def calculate_gramm_matrix(args,img):
    gramm_matrix = np.zeros((img.shape[0]*img.shape[1],img.shape[0]*img.shape[1]))
    img_copy = img.copy()
    img_copy = img_copy.reshape((-1,3))
    color_distance = cdist(img_copy, img_copy,'sqeuclidean')

    grid = np.indices((img.shape[0],img.shape[1]))
    row_indices,col_indices = grid[0],grid[1]
    final_indices = np.hstack((row_indices.reshape(-1,1),col_indices.reshape(-1,1)))
    location_distance = cdist(final_indices,final_indices,'sqeuclidean')

    # return (np.exp(location_distance*(-args.y_s)) * np.exp(color_distance*(-args.y_c)))
    return (np.exp(location_distance*(-args.y_s)) * np.exp(color_distance*(-args.y_c)))
```

3. Give each data point a label
   After having a gram matrix and initial points, we can give each point a corresponding label. We calculate the distance from each point to each center, then select the nearest center as belonging to the same category.

```python
'''
Need to output clustering (number of points) -> each put the label of each points
'''
calculate_distance = np.zeros((gramm_matrix.shape[0],initial_points.shape[0]))

for idx,center_point in enumerate(initial_points):
    calculate_distance[:,idx] = np.diag(gramm_matrix)
    calculate_distance[:,idx] += gramm_matrix[center_point,center_point]
    calculate_distance[:,idx] -= 2*gramm_matrix[:,center_point]
result = np.argmin(calculate_distance,axis=1)

return result
```

   We would recalculate the new label based on the previous clustering result.

```python
# number of cluster in each class
num_cluster = np.array([ np.sum(cluster == c) for c in range(args.n) ])
new_cluster = np.zeros(gramm_matrix.shape[0],dtype=int)
pair_distance = calculate_pair_distance(cluster,gramm_matrix,args,num_cluster)

for p in range(gramm_matrix.shape[0]):
    dis = np.zeros(args.n)
    for idx in range(args.n):
        dis[idx] = gramm_matrix[p,p] + pair_distance[idx]
        tmp = gramm_matrix[p,:].copy()
        tmp[cluster != idx] = 0
        dis[idx] -= ( 2/num_cluster[idx] ) * ( np.sum(tmp) )

    new_cluster[p] = np.argmin(dis)
return new_cluster
```

   The above code is to implement the following equation.

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

Gram matrix!

We would repeat step 3 until a specific condition is met. Our condition is that if the new cluster is nearly the same as the previous cluster then we stop.

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

2. Spectral Clustering

To implement spectral clustering, we follow the bellowed architecure.

1. Construct the gram matrix
   This step is exactly the same as the step 2 in kernel kmeans. We follow the instructions from spec. We use the following equation as our kernel.

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

```python
def calculate_gramm_matrix(args,img):
    gramm_matrix = np.zeros((img.shape[0]*img.shape[1],img.shape[0]*img.shape[1]))
    img_copy = img.copy()
    img_copy = img_copy.reshape((-1,3))
    color_distance = cdist(img_copy, img_copy,'sqeuclidean')

    grid = np.indices((img.shape[0],img.shape[1]))
    row_indices,col_indices = grid[0],grid[1]
    final_indices = np.hstack((row_indices.reshape(-1,1),col_indices.reshape(-1,1)))
    location_distance = cdist(final_indices,final_indices,'sqeuclidean')

    # return (np.exp(location_distance*(-args.y_s)) * np.exp(color_distance*(-args.y_c)))
    return (np.exp(location_distance*(-args.y_s)) * np.exp(color_distance*(-args.y_c)))
```

2. Construct **U matrix**
   After having a gram matrix, we start to construct a U matrix. First, we construct a degree matrix which indicates the summation of degree of each data point.

```python
matrix_w = gramm_matrix.copy()
matrix_d = np.zeros_like(matrix_w)
for idx,row in enumerate(matrix_w):
    matrix_d[idx,idx] = np.sum(row)
```

   After constructing a degree matrix, we would construct matrix L which is the result of D-Gram_matrix.

```python
matrix_l = matrix_d - matrix_w
```

   Then, if we normalize the cut, we should normalize it.

```python
if args.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)
```

   We find the eigenvalue and eigenvector of L. We construct matrix U as

$$\mathbf{U} = \begin{bmatrix} \boldsymbol{u}_2(v_1) & \cdots & \boldsymbol{u}_{k+1}(v_1) \\ \vdots & & \vdots \\ \boldsymbol{u}_2(v_n) & \cdots & \boldsymbol{u}_{k+1}(v_n) \end{bmatrix}$$

We choose k non-zero eigenvectors to construct the U matrix. (K means the number of clusters.)

```
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[:args.n]].T
```

3. Use kmeans to calculate the labels
   First, we select initial points. In here, we also have two kinds of strategies. For one, randomly choose the points.

```
if args.choose_mode == "random":
    # Random strategy
    return matrix_u[np.random.choice(num_of_rows * num_of_cols, args.n)]
```

The other is to choose the points which is as far apart as possible.

```
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(args.n - 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)
```

Second, we calculate the label of each data point. We choose the closest center as our label.

```python
def calculate_normal_kmeans(current_center,matrix_u,args,img):
    new_cluster = np.zeros(matrix_u.shape[0], dtype=int)
    for p in range(matrix_u.shape[0]):
        # Find minimum distance from data point to centers
        distance = np.zeros(args.n)
        for idx, cen in enumerate(current_center):
            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
```

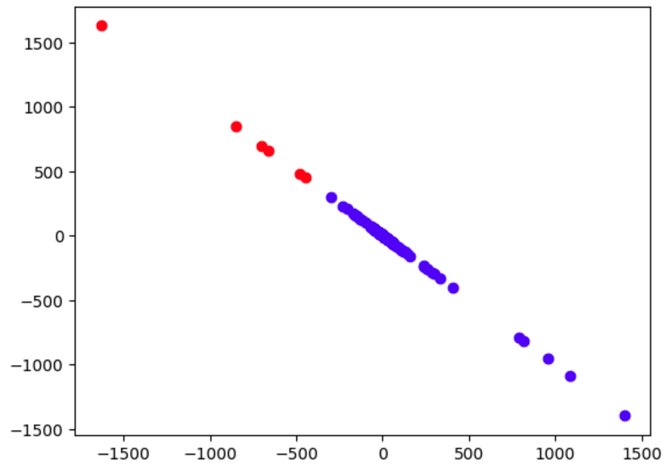
Based on the result, we need to recalculate the center of each cluster.

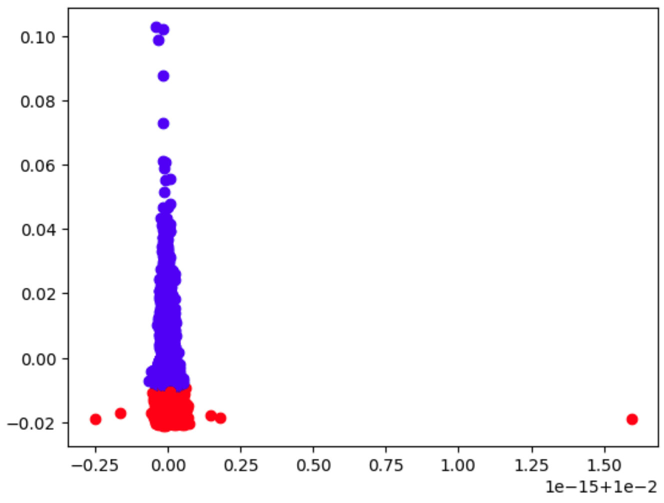
```python
def calculate_new_center(num_of_clusters: int, matrix_u: np.ndarray, current_cluster: np.ndarray) -> np.ndarray:
    """
    Recompute centers according to current cluster
    :param num_of_clusters: number of clusters
    :param matrix_u: matrix U containing eigenvectors
    :param current_cluster: current cluster
    :return: new centers
    """
    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)
```

We would repeat step 3 until a specific condition is met. Our condition is that if the new cluster is nearly the same as the previous cluster then we stop.

```python
if np.linalg.norm((new_center - current_center), ord=2) < 0.001 :
    break
```

This is the result of plotting data points in eigenvector space.





It is obvious that the same cluster would have the same coordinates. Compare with ration and normalized cut we can find that normalized cut can separate data points more clearly.