Github @angel870326

Programming Assignment 2: Spectral Clustering on Star War

1. Ratio-Cut Laplacian (L)

Given the adjacent matrix (A), we can construct the diagonal degree matrix (D) using the NumPy library. `np.diag(np.sum(A, axis=1))` construct a diagonal matrix whose diagonal elements are the sums of the rows of A. After creating D, we can compute the Ratio-Cut Laplacian (L) by subtracting A from D.

A	D	L = D - A
([[0., 1., 1., ..., 0., 0., 0.], [1., 0., 1., ..., 0., 0., 0.], [1., 1., 0., ..., 0., 0., 0.], ..., [0., 0., 0., ..., 0., 1., 1.], [0., 0., 0., ..., 1., 0., 1.], [0., 0., 0., ..., 1., 1., 0.]])	([[17., 0., 0., ..., 0., 0., 0.], [0., 17., 0., ..., 0., 0., 0.], [0., 0., 10., ..., 0., 0., 0.], ..., [0., 0., 0., ..., 4., 0., 0.], [0., 0., 0., ..., 0., 5., 0.], [0., 0., 0., ..., 0., 0., 4.]])	([[17., -1., -1., ..., 0., 0., 0.], [-1., 17., -1., ..., 0., 0., 0.], [-1., -1., 10., ..., 0., 0., 0.], ..., [0., 0., 0., ..., 4., -1., -1.], [0., 0., 0., ..., -1., 5., -1.], [0., 0., 0., ..., -1., -1., 4.]])

2. Top k Eigenvectors

(1) Compute eigenvalues and eigenvectors

We can simply compute the eigenvalues and eigenvectors by calling the NumPy function linalg.eig().

```
[ ]  e, v = np.linalg.eig(L)
```

(2) Sort eigenvectors by eigenvalues (ascending)

The method .argsort() returns the indices that would sort the eigenvalues (e) in ascending order, so that we can sort both the eigenvalues (e) and the eigenvectors (v) by that order. Since v[:, i] is the eigenvector corresponding to the eigenvalue e[i], we transpose the array v first.

```
[ ]  ind = e.argsort()
     ind
```

```
array([29, 33, 34, 35, 36, 38, 39, 40, 41, 64, 37, 42, 60, 61, 63, 65, 62,
       56, 59, 58, 57, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43,
       68, 67, 66, 32, 31, 30, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18,
       17, 15, 16, 14, 12, 13, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,
        0])
```

```
[ ]  sorted_e = e[ind[::1]]
     sorted_v = v.T[ind[::1]]
```

(3) Find top 3 eigenvectors

After sorting the eigenvalues and eigenvectors, we select the top 3 eigenvalues and their corresponding eigenvectors, which are the 3 smallest eigenvalues in the sorted list. The smallest eigenvalue is equal to 0, and its corresponding eigenvector is a constant vector.

```
[ ]  top_k = 3

     # smallest k eigenvalues
     top_e = sorted_e[:top_k]
     print(np.round(top_e, 10))

     # top k eigenvectors
     top_v = sorted_v[:top_k]
     eigen = pd.DataFrame(top_v, columns = id)
     eigen
```

```
[0.         0.4727275  0.65491922]
```

	0	1	2	3	4	5	6	7	8	9	...	102	103
0	0.120386	0.120386	0.120386	0.120386	0.120386	0.120386	0.120386	0.120386	0.120386	0.120386	...	0.120386	0.120386
1	−0.030371	0.021461	0.033133	−0.059563	−0.080287	−0.094750	−0.089101	−0.055608	−0.113178	−0.101323	...	0.035563	0.310957
2	−0.007693	0.013670	0.019272	−0.034807	−0.057147	−0.074879	−0.068011	−0.032708	−0.095971	−0.081160	...	0.021925	−0.050964

3 rows × 69 columns

(4) Create $\hat{X}$

We construct $\hat{X}$ by removing the constant vector from top_v and transposing the selected eigenvectors.

```
[ ]  X = top_v[1:].T
     X
```

```
array([[-0.0303707 , -0.00769338],
       [ 0.02146138,  0.01367009],
       [ 0.03313276,  0.01927221],
       [-0.05956305, -0.03480719],
       [-0.08028724, -0.05714747],
       [-0.09475033, -0.0748792 ],
```

3. K-means

(1) Algorithm

We define a function called **Kmeans()** to implement K-means clustering algorithm with an additional convergence check. A numpy array X containing the data points, an integer k indicating the number of clusters to form, and an optional integer max_iter indicating the maximum number of iterations to run.
We first initialize k centroids randomly from X, and then iteratively performs the following steps until convergence:
Step 1: Compute the distance between each data point and each centroid.
Step 2: Assign each data point to the cluster with the closest centroid.
Step 3: Recalculate the centroids by taking the mean of all the data points assigned to each cluster.
Step 4: Check for convergence by comparing the new centroids to the old centroids. If they are the same, increment the convergence variable. If the centroids have not changed by more than a certain threshold (10 iterations), terminate the algorithm.
Step 5: If the centroids have not converged, update the centroids and repeat the above steps.

```
[ ]  def Kmeans(X: np.array, k: int, max_iter: int = 1000):
         n, d = X.shape
         # Randomly initialize centroids
         centroids = X[np.random.choice(n, k, replace=False), :]
         for i in range(max_iter):
             # Compute the distance between each data point and each centroid
             distances = np.linalg.norm(X[:, np.newaxis, :] - centroids, axis=2)
             # Assign cluster (find the index of the centroid with the smallest distance)
             labels = np.argmin(distances, axis=1)
             # Recalculate centroids
             centroids_new = np.zeros((k, d))
             for j in range(k):
                 centroids_new[j, :] = np.mean(X[labels == j, :], axis=0)
             # Check for convergence (if the new centroids are different from the old centroids)
             convergence = 0
             if np.allclose(centroids, centroids_new):
                 convergence += 1
             if convergence >= 10:
                 break
             # Update centroids
             centroids = centroids_new
         return centroids, labels
```

(2) Result

Finally, we can simply call Kmeans(X, k, max_iter) to get the clustering results.

```
[ ]  k = 3
     max_iter = 1000
     centroids, labels = Kmeans(X, k, max_iter)
     labels
```

```
array([1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2,
       1, 2, 2, 2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 0, 0, 2, 0, 2, 2, 2, 2, 0,
       2, 2, 2])
```