2019/06/3
0780828
Alisher Mukashev

HOMEWORK #6

1. You need to make videos or GIF images to show the clustering procedure (visualize the cluster assignments of data points in each iteration, colorize each cluster with different colors) of your k-means, kernel k-means, spectral clustering and DBSCAN program.

2. In addition to cluster data into 2 clusters, try more clusters (e.g. 3 or 4) and show your results.

K-MEANS (homework_6_1_kmeans.py)

Explanation of code.

First step: set initial data. There are four values which you can change (must select only one of them):
- *dataset = circle* OR *moon* (dataset type)
- *k = 2* OR *3* OR *4* (number of clusters)
- *initial_mean = -1 + 2*np.random.random((k, dataset.shape[1]))* OR *km.k_means_plus_plus(dataset, k)* (The first one is the uniform data generator. The last one is the k-means++ algorithm, let's talk about it later)
- *error = 7* OR any number (convergence condition).

Second step: run k-means algorithm. Here are the steps summary for doing k-means clustering:
- initiate means for each clusters (already done).
- assign each data input into each cluster. Each data input belongs to cluster, whose the mean is the nearest one. This is done by using Euclidean norm.

*weights = np.matmul(dataset**2, np.ones((dataset.shape[1], mean.shape[0]))) *

```
+ np.matmul(np.ones((dataset.shape[0], dataset.shape[1])), (mean**2).T) \
- 2*np.dot(dataset, mean.T)
classification = np.zeros(weights.shape)
ind_min = np.argmin(weights, axis=1)
classification[np.arange(weights.shape[0]), ind_min] = 1
```

- Re-calculate the means of each these clusters. Let's use the equation:

$$\mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}} \tag{1}$$

```
mean = np.matmul(classification.T, dataset) / np.sum(classification, axis=0)[:, None]
```

- check convergence. This is done by comparing current classified matrix with previous classified matrix.

```
error = np.count_nonzero(np.abs(classification - previous_classification))
if error < err:
    break
```

Third step: save obtained results.

```
img = af.save_cluster(dataset, set_classification, k)
```

Discussion.

We need to specify how many clusters we want to make, and we need to specify means of each clusters. Different means initialization may give different final result clusters. All obtained results you can find in the folder 'kmeans_gifs'.

KERNEL K-MEANS (homework_6_1_kernel_kmeans.py)

Explanation of code.

First step: set initial data. There are four values which you can change (must select only one of them):

- *dataset = circle* OR *moon* (dataset type)
- *k = 2* OR *3* OR *4* (number of clusters)
- *gamma = 7* OR any number (for RBF kernel)
- *error = 7* OR any number (convergence condition).

Second step: run kernel k-means algorithm. Here are the steps summary for doing kernel k-means clustering:

- calculate Gram matrix by using RBF kernel.
- assign each data input into each cluster. Here is the difference with previous one – this is using of RBF kernel or Gram matrix. Let's use the equation:

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

```
weights = Kernel(previous_classification, gram_matrix)
classification = np.zeros(weights.shape)
ind_min = np.argmin(weights, axis=1)
classification[np.arange(weights.shape[0]), ind_min] = 1
```

- check convergence. This is done by comparing current classified matrix with previous classified matrix.

```
error = np.count_nonzero(np.abs(classification - previous_classification))
if error < err:
    break
```

Third step: save obtained results.

```
img = af.save_cluster(dataset, set_classification, k)
```

Discussion.

We able to identify the non-linear structure. The mean doesn't have to be computed because computations only depend on kernel evaluation. All obtained results you can find in the folder 'kernel-kmeans_gifs'.

SPECTRAL CLUSTERING (homework_6_1_spectral_clustering.py)

Explanation of code.

First step: set initial data. There are five values which you can change (must select only one of them):
- *dataset = circle* OR *moon* (dataset type)
- *k = 2* OR *3* OR *4* (number of clusters)
- *gamma = 7* OR any number (for RBF kernel)
- *error = 7* OR any number (convergence condition)

- *initial_mean = None* OR *km.k_means_plus_plus(U, k)* (The last one is the k-means++ algorithm, let's talk about it later).

If *initial_mean = None* then it's necessary to commute mean by using equation (1). But the thing is, that it uses the datapoints from eigenspace (U).

```
classification = np.zeros((U.shape[0], k))
ind_min = np.random.randint(k, size=U.shape[0])
classification[np.arange(U.shape[0]), ind_min] = 1
mean = np.matmul(classification.T, U) / np.sum(classification, axis=0)[:, None]
```

Second step: move datapoints from original space to eigenspace. Here are the steps summary for doing it:

- consider given datapoints as a graph
- represent graph as an adjacency (similarity) matrix. It could be done by using RBF kernel.
- find degree (diagonal) matrix.

```
D = np.sum(W, axis=1)*np.eye(W.shape[0])
```

- calculate the graph Laplacian (just subtract the adjacency matrix from our degree matrix)

```
L = D - W
```

- get eigenvalues and eigenvectors from graph Laplacian

```
eigenvalues, eigenvectors = np.linalg.eig(L)
```

- project the data onto eigenspace that can be easily separated by using k-means clustering. It could be done by using beautiful properties of eigenvalues and eigenvectors.

```
sorted_idx = np.argsort(eigenvalues)
for i in range(k):
        evc = eigenvectors[:, sorted_idx[i]]
        U.append(evc[:, None])
U = np.concatenate(U, axis=1)
```

Third step: run k-means by using obtained datapoints in eigenspace.

Fourth step: save obtained results.

```
img = af.save_cluster(dataset, set_classification, k)
```

Discussion.

Compared to kernel k-means clustering and k-means clustering, this method gives us the best results. All obtained results you can find in the folder 'spectral-clustering_gifs'.

DBSCAN (homework_6_1_dbscan.py)

Explanation of code.

First step: set initial data. There are three values which you can change (must select only one of them):

- *dataset = circle* OR *moon* (dataset type)
- *min_points = 3* OR any number ()
- *eps = 0.1* OR any number (the minimum distance between two points. It means that if the distance between two points is lower or equal to this value (eps), these points are considered neighbors)

Second step: run dbscan algorithm. Here are the steps summary for doing dbscan clustering:

- find the points in the *eps* neighborhood of every point, and identify the core points with more than *min_points* neighbors. Assign each non-core point to a nearby cluster if the cluster is an ε (eps) neighbor, otherwise assign it to noise.

```
n = np.sum((dataset - dataset[i])**2, axis=1) <= eps
if sum(n) >= min_points:
        n[i] = False
        n = np.logical_and(n, np.logical_or(Y==0, Y==-1))
        Y[n] = C
        return np.where(n)[0]
else:
        Y[n] = -1
        return np.ndarray(0)
```

Third step: save obtained results.

```
imageio.mimsave('dbscan_gifs/dbscan.gif', img, fps=1)
```

Discussion.

The advantage of this method - doesn't need to set number of clusters. This algorithm may take some time. You can just find all obtained results in the folder 'dbscan_gifs'.

3. For the initialization of k-means clustering used in k-means, kernel k-means and spectral clustering, try different ways and show corresponding results, e.g. k-means++.

K-MEANS++ (for k-means and spectral clustering)

Explanation of code.

This algorithm has already been mentioned. Let's look deeper…

k-means++ is an algorithm for choosing the initial values for the k-means clustering algorithm. Here are the steps summary for doing k-means++ clustering:

- choose one center uniformly at random from among the data points

mean[0, :] = dataset[np.random.choice(range(dataset.shape[0]), 1), :]

- for each data point x, compute weights, the distance between x and the nearest center that has already been chosen.

*weights = np.sum((dataset-mean[j-1])**2, axis=1)*

- choose one new data point at random as a new center, using a weighted probability distribution.

prob = np.cumsum(weights/np.sum(weights))

c = prob.searchsorted(np.random.rand(), 'right')

mean[j, :] = dataset[c, :]

- now that the initial centers have been chosen, proceed using standard k-means clustering or spectral clustering.

Discussion.

This is one way to improve k-means clustering. As I told before that different initialization may give different final cluster result. All obtained results you can find in the folder ‘kmeans++’.

Gamma parameter of the RBF kernel (for kernel k-means clustering)

As I told before the RBF kernel really matters for kernel k-means clustering. That’s why let’s take a look an influence of Gamma parameter of RBF kernel on the clustering performance. Note that before starting you must switch the value *change_gamma* to *True*.

Discussion.

The gamma parameter is the inverse of the standard deviation of the RBF kernel (Gaussian function), which is used as similarity measure between two points. Intuitively, a small gamma value defines a Gaussian function with a large variance. In this case, two points can be considered similar even if are far from each other. In the other hand, a large gamma value means defines a Gaussian function with a small variance and in this case, two points are considered similar just if they are close to each other. So to choose right gamma we should consider input datapoints. All obtained results you can find in the folder ‘change_gamma’.

4. For spectral clustering, you can see if data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian. You should plot the result and discuss it in the report.

Explanation of code.

To prove that spectral clustering really works I can show obtained datapoints in eigenspace. This can make everything clearer.

From the previous task you already got datapoints in eigenspace. Now you need to display this data. Let's do this!

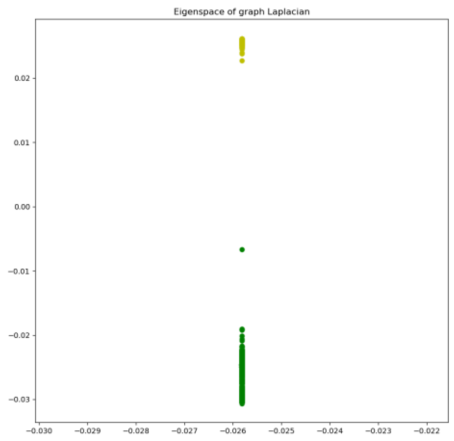
k – number of clusters (0, 1)

classification – already got after k-means clustering

dataset – new datapoints (U)

```
for i in range(0, k):
        for j in range(0, dataset.shape[0]):
            if classification[j] == i:
            ax.scatter(dataset[j][0], dataset[j][1], c=colors[i])
```

Here is the figure which show us the datapoints in eigenspace.



Discussion.

As you can see the data separation is very clear.