University of California, Berkeley Fall 2022, EECS 182/282

The power of the graph perspective in clustering

Dependencies

In [15]:
```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"

import numpy as np
import matplotlib.pyplot as plt

from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

from scipy.linalg import svd
from scipy.spatial import distance
from sklearn.preprocessing import normalize

import math
```

Helper Functions

In [16]:
```
def show_data_results(X, num_plot=2, y_pred=None, cmap='jet'):
    if num_plot==1:
        plt.scatter(X[:, 0], X[:, 1])
        plt.title ("input data")
    elif num_plot==2:
        try:
            assert y_pred is not None
            fig = plt.figure(figsize=(10, 3))
            ax1 = fig.add_subplot(121)
            ax1.scatter(X[:, 0], X[:, 1])
            ax1.set(xticks=[],yticks=[],title ="input data")

            ax2 = fig.add_subplot(122)
            ax2.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=cmap)
            ax2.set(xticks=[], yticks=[], title ="clustered data")
        except:
            print('y_pred is required for 2 plots')
```

Q.1. (Given)

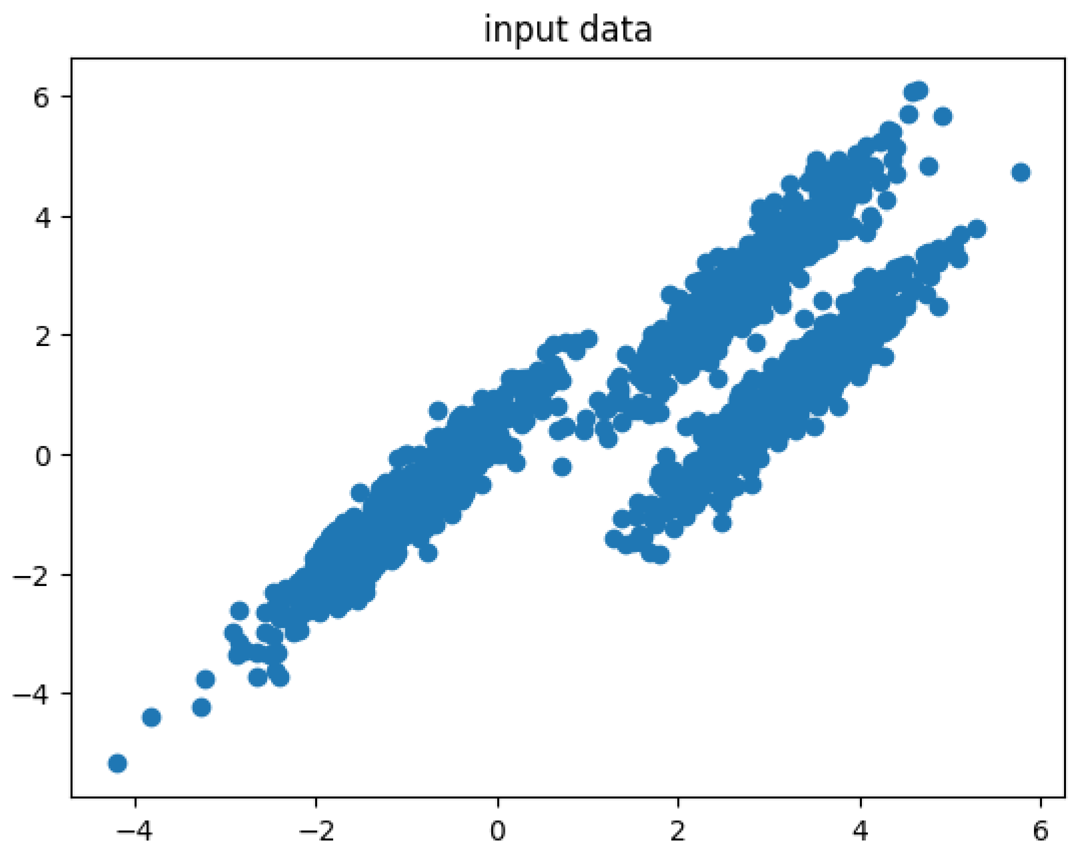
Please, read https://en.wikipedia.org/wiki/K-means_clustering (https://en.wikipedia.org/wiki/K-means_clustering) about the Kmeans algorithm.

In this problem, we will show how interperting a dataset as a graph may result is obtaining an elegant clustering solution. We have an input dataset that we wish to cluster in 3 aparant classes.

We provide the synthetic dataset of 2000 points described below where the T_matrix is just a 2D transformation matrix:

```
In [17]: T_matrix, seed = [[-0.60834549, -0.63667341], [0.40887718, 0.85253229]], 170
         X_orig, y_orig = make_blobs(n_samples=2000, random_state=seed)
         X = np.dot(X_orig, T_matrix)
```

```
In [18]: show_data_results(X, num_plot=1)
```

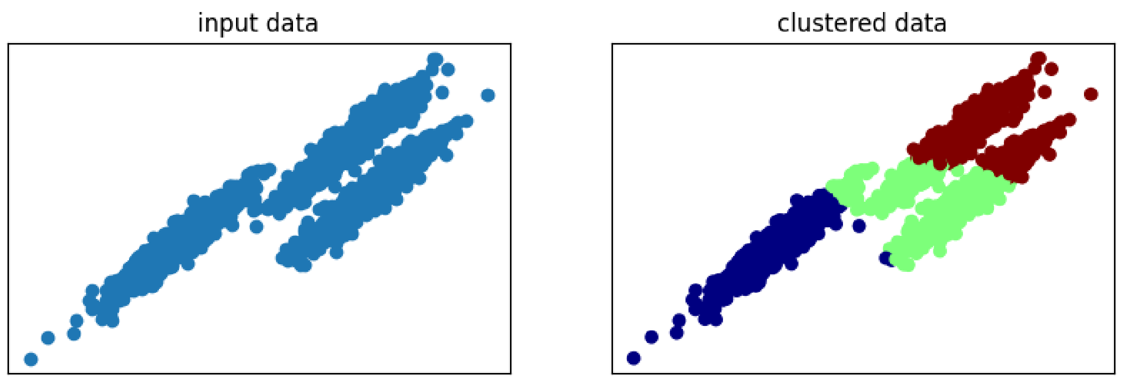


Using the the Kmeans algorithm implementation of sklearn, show your attempt to cluster this dataset into 3 classes in one luine of code.

Solution (Given)

```
In [19]: y_pred = KMeans(n_clusters=3, random_state=seed).fit_predict(X)
```

In [20]:
```
show_data_results(X, 2, y_pred)
```



Q.2.

Comment on the output the the KMeans algorithm? Did it work? If so explain why, if not, explain not not.

Solution

Type *Markdown* and LaTeX: α^2

Q.3.

Let's now interprete every single point in the provided dataset as a node in a graph. Our goal is to find a way to relate every node in the graph is such way that they points that closer together maintain that relationship while points that are far are explicitely identified.

lots of points points are closed top each other and kmaeans is missing it. representing as the graph unveils the relation.

One way to capture such relationship between points (nodes) in a graph is through the Adjacency matrix. Typically, a simple adjacency matrix between nodes of and indiredted graph is given by:

$$A_{i,j} = \begin{cases} 1 : \text{if there is an edge between node i and node j,} \\ 0 : \text{otherwise.} \end{cases}$$

In this probem, we will use the weighted distances betweeen points instead as a similary measure. Write a function that takes in the input dataset and some coeficient gamma which returns the adjacency matrix A.

$$A_{i,j} = e^{\gamma ||x_i - x_j||^2}$$

where x_i and x_j represent each point in the provided dataset. You may find the $dictance$ module from $scipy.spatial$ useful.

In [21]:
```
print(X.shape)
```
```
(2000, 2)
```

Solution

In [40]:
```
def get_adjacency_matrix(gamma, X):
    # fill in your code here
    # adjacency_matrix = ?

    #num = X.shape[0]
    #adjacency_matrix = np.zeros((num,num))

    #for i in range(num):
    #    for j in range(i, num):
    #        dis = math.exp(-gamma * np.square(distance.euclidean(X[i],X[j])))
    #        adjacency_matrix[i][j] = dis
    #        adjacency_matrix[j][i] = dis

    adjacency_matrix = np.exp(-gamma * distance.cdist(X,X,metric='sqeuclidean'))

    return adjacency_matrix
```

In [41]:
```
adj = get_adjacency_matrix(gamma=1, X=X)
print(adj)
```
```
[[1.00000000e+00 1.16130639e-02 5.62548235e-10 ... 6.92521915e-09
  3.80974378e-15 6.19953663e-02]
 [1.16130639e-02 1.00000000e+00 4.22486809e-16 ... 3.40414939e-15
  1.31605674e-22 1.36491384e-02]
 [5.62548235e-10 4.22486809e-16 1.00000000e+00 ... 8.49932392e-01
  2.60842144e-01 7.64214433e-18]
 ...
 [6.92521915e-09 3.40414939e-15 8.49932392e-01 ... 1.00000000e+00
  1.26977418e-01 2.67051116e-16]
 [3.80974378e-15 1.31605674e-22 2.60842144e-01 ... 1.26977418e-01
  1.00000000e+00 1.22127633e-24]
 [6.19953663e-02 1.36491384e-02 7.64214433e-18 ... 2.67051116e-16
  1.22127633e-24 1.00000000e+00]]
```

In [26]:
```
adj.shape
```
Out[26]:
```
(2000, 2000)
```

In [32]:
```
compare = np.exp(-1 * distance.cdist(X,X,metric='sqeuclidean'))
compare.shape
```
Out[32]:
```
(2000, 2000)
```

```
In [33]: compare
```

```
Out[33]: array([[1.00000000e+00, 1.16130639e-02, 5.62548235e-10, ...,
                 6.92521915e-09, 3.80974378e-15, 6.19953663e-02],
                [1.16130639e-02, 1.00000000e+00, 4.22486809e-16, ...,
                 3.40414939e-15, 1.31605674e-22, 1.36491384e-02],
                [5.62548235e-10, 4.22486809e-16, 1.00000000e+00, ...,
                 8.49932392e-01, 2.60842144e-01, 7.64214433e-18],
                ...,
                [6.92521915e-09, 3.40414939e-15, 8.49932392e-01, ...,
                 1.00000000e+00, 1.26977418e-01, 2.67051116e-16],
                [3.80974378e-15, 1.31605674e-22, 2.60842144e-01, ...,
                 1.26977418e-01, 1.00000000e+00, 1.22127633e-24],
                [6.19953663e-02, 1.36491384e-02, 7.64214433e-18, ...,
                 2.67051116e-16, 1.22127633e-24, 1.00000000e+00]])
```

Q. 4.

The degree matrix of an undirected graph is a diagonal matrix which contains information about the degree of each vertex. In other word, it contains the number of edges attached to each vertex and it is given by:

$$D_{i,j} = \begin{cases} deg(v_i) : \text{if i == j}, \\ 0 : \text{otherwise.} \end{cases}$$

where the degree $\deg(v_i)$ of a vertex counts the number of times an edge terminates at that vertex. Note that in the traditional definition of the adjacency matrix, this boils down to the diagonal matrix in which element along the diagonals are column-wise sum of the adjacency matrix. Using the same idea, write a function that takes in the adjacency matrix as argument and returns the inverse square root of degree matrix.

Solution

```
In [27]: def get_degree_matrix(adjacency_matrix):
             # fill in your code here
             # degree_matrix = ?
             degree_matrix = np.diag(np.sum(adjacency_matrix, axis=0))
             return degree_matrix
```

Type *Markdown* and LaTeX: α^2

Q. 5.

Using γ = 7.5, compute the symmetrically normalized adjacency matrix A, degree matrix D and the matrix $M = D^{-1/2} A D^{-1/2}$

Solution

In [46]:
```
# adjacency_matrix = ?
# degree_matrix = ?
# M = ?
adjacency_matrix = get_adjacency_matrix(7.5, X)
degree_matrix = get_degree_matrix(adjacency_matrix)
inv = np.linalg.inv(np.sqrt(degree_matrix))
M = inv @ adjacency_matrix @ inv
```

In [47]:
```
M
```

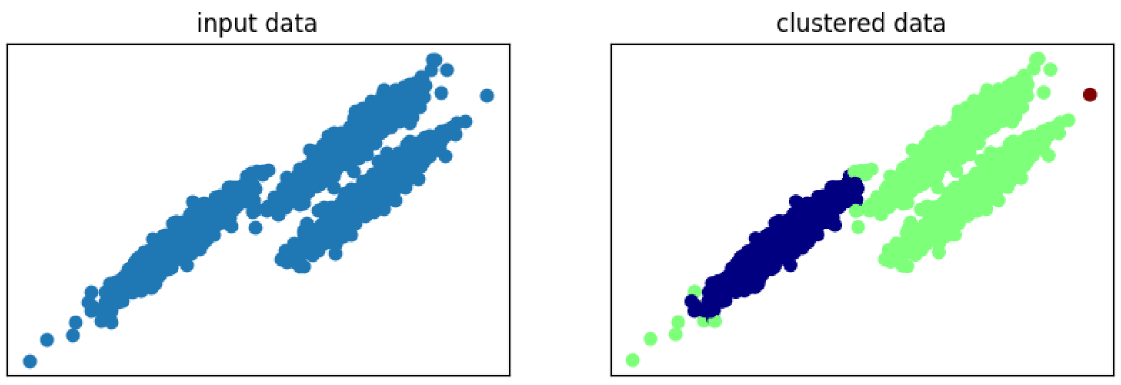
Out[47]:
```
array([[1.56324335e-002, 5.95177042e-017, 5.28958398e-072, ...,
        8.42495130e-064, 1.55115214e-110, 1.17671220e-011],
       [5.95177042e-017, 2.40471459e-002, 7.66196907e-118, ...,
        5.08041331e-111, 2.09900856e-166, 1.71705334e-016],
       [5.28958398e-072, 7.66196907e-118, 1.00096744e-002, ...,
        3.13253209e-003, 7.24379279e-007, 4.52154623e-131],
       ...,
       [8.42495130e-064, 5.08041331e-111, 3.13253209e-003, ...,
        1.12358359e-002, 3.46875976e-009, 1.80186978e-119],
       [1.55115214e-110, 2.09900856e-166, 7.24379279e-007, ...,
        3.46875976e-009, 2.97745821e-002, 8.29831903e-182],
       [1.17671220e-011, 1.71705334e-016, 4.52154623e-131, ...,
        1.80186978e-119, 8.29831903e-182, 1.15325532e-002]])
```

Q. 6.

Using SVD decomposition, select the first 3 vectors in the matrix U and perform the same KMeans clustering used above on them. What do you observe? Did it work? If so explain why, if not, explain not not.

Solution

In [48]:
```
u, s, vh = np.linalg.svd(M)
svd_y_pred = KMeans(n_clusters=3, random_state=seed).fit_predict(u[:, :3])
show_data_results(X, 2, svd_y_pred)
```



In [50]:
```
u[:, :3].shape
```

Out[50]:
```
(2000, 3)
```

Q.7.

Now lets think of the Adjacency obtained above as the transition Matrix in of a Markov Chain.To do so, A needs to be a proper stochastic matrix which means that the sum of the element in each column must add up to 1. Write a function that takes in the matrix M and returns M_stachastic, the stochastic version of M; compute the stochastic matrix

Solution

In [51]:
```
def stochastic_matrix_converter(M):
    # fill in your code here
    # degree_matrix = ?
    M_stoch = M / np.sum(M, axis=0)
    return M_stoch
```

In [52]:
```
M_stoch = stochastic_matrix_converter(M)
M_stoch
```

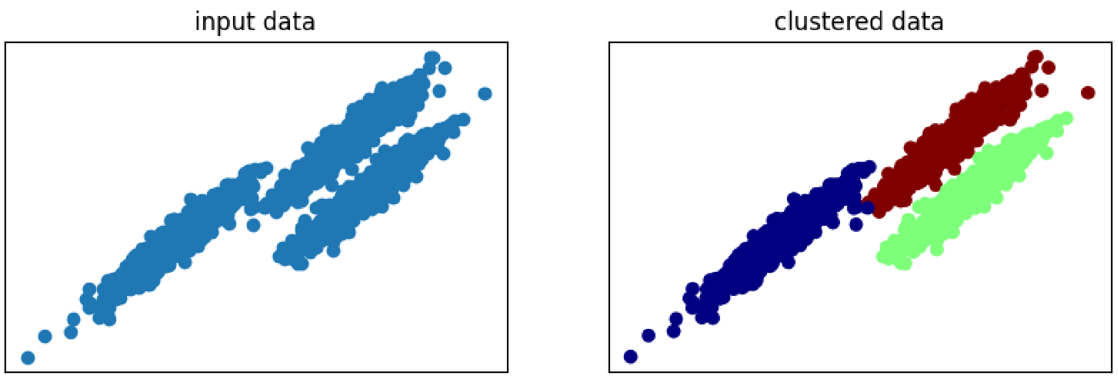
Out[52]:
```
array([[1.63135023e-002, 6.90398945e-017, 4.96294146e-072, ...,
        8.26245010e-064, 1.79736241e-110, 1.14711464e-011],
       [6.21107523e-017, 2.78944298e-002, 7.18882696e-118, ...,
        4.98242185e-111, 2.43217864e-166, 1.67386472e-016],
       [5.52003887e-072, 8.88780144e-118, 9.39155672e-003, ...,
        3.07211154e-003, 8.39358087e-007, 4.40781688e-131],
       ...,
       [8.79200687e-064, 5.89322461e-111, 2.93909189e-003, ...,
        1.10191181e-002, 4.01934683e-009, 1.75654779e-119],
       [1.61873223e-110, 2.43482727e-166, 6.79647391e-007, ...,
        3.40185402e-009, 3.45006229e-002, 8.08959343e-182],
       [1.22797882e-011, 1.99176335e-016, 4.24233159e-131, ...,
        1.76711516e-119, 9.61548928e-182, 1.12424777e-002]])
```

Q.8

Now, let’s investigate how could we have made the matrix M work directly in our original interpretation. To do this, normalize those 3 vectors first before performing the clustering.

Solution

In [53]:
```
u, s, vh = np.linalg.svd(M_stoch)
svd_y_pred = KMeans(n_clusters=3, random_state=seed).fit_predict(u[:, :3])
show_data_results(X, 2, svd_y_pred)
```



In [ ]:
```
```