1. Directed and Undirected Graphs

(a) **Show that $L^{SymNorm}$ could also be written as**:

$$L^{SymNorm} = D^{-1/2} L D^{-1/2}$$

where D is the dregree matrix, and L is the unnormalized Laplacian.

$$L^{symNorm} = I - A^{symNorm} = I - D^{-1/2} A D^{-1/2}$$
$$= I - D^{-1/2} (D - L) D^{-1/2}$$
$$= I - I + D^{-1/2} L D^{-1/2}$$
$$= D^{-1/2} L D^{-1/2}$$

(b) **Write the unnormalized adjacency A, the degree matrix, D, and the symmetrically normalized adjacency matrix, $A^{SymNorm}$, of the graph in Figure. 1.**

	1	2	3	4	5
1		1		1	1
2	1			1	1
3					1
4	1	1			
5	1	1	1		

A

	1	2	3	4	5
1	3	0	0	0	0
2	0	3	0	0	0
3	0	0	1	0	0
4	0	0	0	2	0
5	0	0	0	0	3

D

	1	2	3	4	5
1	$\frac{\sqrt{3}}{3}$				
2		$\frac{\sqrt{3}}{3}$			
3			1		
4				$\frac{\sqrt{2}}{2}$	
5					$\frac{\sqrt{3}}{3}$

$D^{-1/2}$

$$A^{symNorm} = D^{-1/2} A D^{-1/2} = \begin{bmatrix} \frac{\sqrt{3}}{3} & & & & \\ & \frac{\sqrt{2}}{3} & & & \\ & & 1 & & \\ & & & \frac{\sqrt{2}}{2} & \\ & & & & \frac{\sqrt{3}}{3} \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{3} & & & & \\ & \frac{\sqrt{3}}{3} & & & \\ & & 1 & & \\ & & & \frac{\sqrt{2}}{2} & \\ & & & & \frac{\sqrt{3}}{3} \end{bmatrix}$$

$$= \begin{bmatrix} 0 & \frac{\sqrt{3}}{3} & 0 & \frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} \\ \frac{\sqrt{3}}{3} & 0 & 0 & \frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} \\ 0 & 0 & 0 & 0 & 1 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 & 0 \\ \frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{3} & & & & \\ & \frac{\sqrt{3}}{3} & & & \\ & & 1 & & \\ & & & \frac{\sqrt{2}}{2} & \\ & & & & \frac{\sqrt{3}}{3} \end{bmatrix}$$

	1	2	3	4	5
1	0	$\frac{1}{3}$	0	$\frac{\sqrt{6}}{6}$	$\frac{1}{3}$
2	$\frac{1}{3}$	0	0	$\frac{\sqrt{6}}{6}$	$\frac{1}{3}$
3	0	0	0	0	$\frac{\sqrt{3}}{3}$
4	$\frac{\sqrt{6}}{6}$	$\frac{\sqrt{6}}{6}$	0	0	0
5	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{\sqrt{3}}{3}$	0	0

$A^{symNorm}$

(c) **Write the symmetrically normalized Laplacian matrix of the graph in Figure. 1.**

	1	2	3	4	5
1	1	$\frac{1}{3}$	1	$-\frac{\sqrt{6}}{6}$	$\frac{2}{3}$
2	$\frac{2}{3}$	1	1	$-\frac{\sqrt{6}}{6}$	$\frac{2}{3}$
3	1	1	1	1	$-\frac{\sqrt{3}}{3}$
4	$-\frac{\sqrt{6}}{6}$	$-\frac{\sqrt{6}}{6}$	1	1	1
5	$\frac{2}{3}$	$\frac{2}{3}$	$1-\frac{\sqrt{3}}{3}$	1	1

(d) **Compute A^2, A^3**

$$A^2 = \begin{bmatrix} 3 & 2 & 1 & 1 & 1 \\ 2 & 3 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 2 & 2 \\ 1 & 1 & 0 & 2 & 3 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 4 & 5 & 1 & 5 & 6 \\ 5 & 4 & 1 & 5 & 6 \\ 1 & 1 & 0 & 2 & 3 \\ 5 & 5 & 2 & 2 & 2 \\ 6 & 6 & 3 & 2 & 2 \end{bmatrix}$$

(e) **Write the unnormalized adjacency matrix of the graph in Figure 2.**

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

(f) **Write the In-degree D_{in} and Out-degree D_{out} matrix of the graph in Figure. 2.**

$$D_{in} = \begin{bmatrix} 1 & & & & & \\ & 2 & & & & \\ & & 1 & & & \\ & & & 2 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

$$D_{out} = \begin{bmatrix} 1 & & & & & \\ & 2 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

(g) **Write both of the symmetrically normalized In-degree and Out-degree Laplacian matrix graph in Figure. 2.**

$$D_{in}^{-1/2} = \begin{bmatrix} 1 & & & & & \\ & \frac{1}{2} & & & & \\ & & 1 & & & \\ & & & \frac{1}{2} & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

$$D_{out}^{-1/2} = \begin{bmatrix} 1 & & & & & \\ & \frac{1}{2} & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

$$L_{in} = D_{in} - A = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

$$L_{out} = D_{out} - A = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

$$L_{in}^{symNorm} = D_{in}^{-1/2} L_{in} D_{in}^{-1/2} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & +1 & 0 & 0 & +1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & & & & \\ & \frac{1}{2} & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & \frac{1}{2} & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \end{bmatrix}$$

$$= \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -\frac{1}{2} & 0 & 0 & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & -\frac{1}{2} & 0 & 0 & -\frac{1}{2} & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

$$L_{out}^{symNorm} = D_{out}^{-1/2} L_{out} D_{out}^{-1/2} = \begin{bmatrix} 1 & & & & & & & \\ & \frac{1}{2} & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \end{bmatrix} \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 & +1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & & & & \\ & \frac{1}{2} & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \end{bmatrix}$$

$$= \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -\frac{1}{2} & 0 & 0 & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & & & & \\ & \frac{1}{2} & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \end{bmatrix}$$

$$= \begin{bmatrix} 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

(h) The adjancy matrix can capture the connectivity of the graph in a concise and computationally efficient way.

The Laplacian matrix is derived from adjacent matrix, which provides a measure of the "smothness" of the graph, and it can be used to compute various properties such as the number of connected components, and it's useful for clustering and partitioning the nodes of graph

2. Graph Dynamics

(a) Let A be the $n \times n$ size adjacency matrix for the underlying graph where the entry $A_{i,j} = 1$ if vertices i and j are connected in the graph and 0 otherwise. **Write the output of the j-th node at layer k in this network in terms of the matrix A.**

(Hint: This output is an n-dimensional vector since there are n output channels at each layer.)

Let $X_i^{(k)}$ denotes the ith node of layer k, then

$$X_j^{(k)} = \sum_{i=1}^{n} A_{i,j} X_i^{(k-1)}$$

(b) Here is some helpful notation: Let $V(i)$ be the set of vertices that are connected to vertex i in the graph. Let $L_k(i,j)$ be the number of distinct paths that go from vertex i to vertex j in the graph where the number of edges traversed in the path is exactly k. Recall that a path from i to j in a graph is a sequence of vertices that starts with i, ends with j, and for which every successive vertex in the sequence is connected by an edge in the graph. The length of the path is 1 less than the number of vertices in the corresponding sequence. **Show that the i-th output of node j at layer k in the network above is the count of how many paths there are from i to j of length k**, where by convention there is exactly 1 path of length 0 that starts at each node and ends up at itself.

(Hint: Can applying induction on k help?)

in the layth 0 when k = 0, the path count is 1, which meets the initialization of the graph

when k=1, the path count is

(c) The structure of the neural network in this problem is compatible with a straightforward linear graph neural network since the operations done (just summing) are locally permutation-invariant at the level of each node and can be viewed as essentially doing the exact same thing at each vertex in the graph based on inputs coming from its neighbors. This is called "aggregation" in the language of graph neural nets. In the case of the computations in previous parts, **what is the update function that takes the aggregated inputs from neighbors and results in the output for this node?**

$$X_j^{k+1} \leftarrow \sum_{i=1}^{n} A_{ij} X_i^k$$

(d) The simple GNN described in the previous parts counts paths in the graph. If we were to replace sum aggregation with max aggregation, **what is the interpretation of the outputs of node j at layer k?**

shortest path

3. The power of the graph perspective in clustering (Coding)

(a) We used the KMeans algorithm implementation of sklearn, and showed our attempt to cluster this dataset into 3 classes. **Comment on the output the KMeans algorithm? Did it work? If so explain why, if not, explain not.**

It works, the kMeans classifies the data points into 3 parts because the algorithm can find the nearest class for every datapoints

(b) As given, the data points in our dataset are represented simply with their 2D Cartesian coordinates. Let's now interpret every single point as a node in a graph. Our goal is to find a way to relate every node in the graph in such way that the points that are closer together and points that are far apart maintain that relationship explicitly.

That is, we will choose to look at every point in the dataset as a vertex in a graph where the edge connection between two vertexes is determined by the weighted distances between them. **Write a function that takes in the input dataset and some coefficient gamma and returns the adjacency matrix A. Is this a directed or an undirected graph?**

$$A_{i,j} = e^{-\gamma||x_i - x_j||^2} \tag{10}$$

where x_i and x_j represent each point in the provided dataset, γ is positive. You may find the $distance$ module from $scipy.spatial$ useful.

It is an undirected graph

```
def get_adjacency_matrix(gamma, X):
    # TODO - fill in your code here
    x, y = X.shape
    print(X.shape)
    adjacency_matrix = np.zeros((x, x))
    for i in range(x):
      for j in range(x):
        adjacency_matrix[i][j] = np.exp(-gamma * distance.euclidean(X[i], X[j])**2)
    return adjacency_matrix
```

(c) The degree matrix of an undirected graph is a diagonal matrix that contains information about the degree of each vertex. In other words, it contains the number of edges attached to each vertex and it is given by Eq (4) in problem 3. Note that in the traditional definition of the adjacency matrix, this boils down to the diagonal matrix in which elements along the diagonals are the column-wise sum of the elements in the adjacency matrix. Using the same idea, **write a function that takes in the adjacency matrix as an argument and returns the degree matrix.**

```
def get_degree_matrix(adjacency_matrix):
    # TODO - fill in your code here
    degree_matrix = np.zeros_like(adjacency_matrix)
    X, Y = adjacency_matrix.shape
    for i in range(X):
      for j in range(Y):
        if i != j:
          degree_matrix[i][i] = degree_matrix[i][i] + adjacency_matrix[i][j]

    return degree_matrix
```

(d) Using $\gamma = 7.5$, **compute the adjacency matrix A, degree matrix D and the symmetrically normalized adjacency matrix matrix** M,

$$M = A^{SymNorm} = D^{-1/2}AD^{-1/2} \tag{11}$$

Note that another interpretation of the matrix M is that it shows the probability of moving/jumping from one node to another.

```
D_12_inv =  [[0.1260186  0.         0.         ... 0.         0.         0.        ]
 [0.         0.15697025 0.         ... 0.         0.         0.        ]
 [0.         0.         0.10055288 ... 0.         0.         0.        ]
 ...
 [0.         0.         0.         ... 0.10659979 0.         0.        ]
 [0.         0.         0.         ... 0.         0.1751808  0.        ]
 [0.         0.         0.         ... 0.         0.         0.18801437]]
A =  [[1.00000000e+000 3.06973656e-015 4.22861598e-070 ... 6.35698827e-062
  7.18982405e-109 8.76383571e-010]
 [3.06973656e-015 1.00000000e+000 4.93854085e-116 ... 3.09075501e-109
  7.84439767e-165 1.03107269e-014]
 [4.22861598e-070 4.93854085e-116 1.00000000e+000 ... 2.95381206e-001
  4.19597803e-005 4.20837341e-129]
 ...
 [6.35698827e-062 3.09075501e-109 2.95381206e-001 ... 1.00000000e+000
  1.89648185e-007 1.58291638e-117]
 [7.18982405e-109 7.84439767e-165 4.19597803e-005 ... 1.89648185e-007
  1.00000000e+000 4.47821104e-180]
 [8.76383571e-010 1.03107269e-014 4.20837341e-129 ... 1.58291638e-117
  4.47821104e-180 1.00000000e+000]]
D =  [[62.96956692  0.          0.         ...  0.          0.        ]
 [ 0.         ]
 [ 0.         40.58497658  0.         ...  0.          0.        ]
 [ 0.         ]
 [ 0.          0.         98.90334959 ...  0.          0.        ]
 [ 0.         ]
 ...
 [ 0.          0.          0.         ... 88.00094375  0.        ]
 [ 0.         ]
 [ 0.          0.          0.         ...  0.         32.58569385]
 [ 0.         ]
 [ 0.          0.          0.         ...  0.          0.        ]
 85.71106768]]
M =  [[1.58806873e-002 6.07229837e-017 5.35830454e-072 ... 8.53969564e-064
  1.58722839e-110 1.19291753e-011]
 [6.07229837e-017 2.46396594e-002 7.79489914e-118 ... 5.17175886e-111
  2.15706616e-166 1.74818819e-016]
 [5.35830454e-072 7.79489914e-118 1.01108810e-002 ... 3.16616610e-003
  7.39118749e-007 4.57077996e-131]
 ...
 [8.53969564e-064 5.17175886e-111 3.16616610e-003 ... 1.13635145e-002
  3.54153501e-009 1.82261885e-119]
 [1.58722839e-110 2.15706616e-166 7.39118749e-007 ... 3.54153501e-009
  3.06883139e-002 8.47369089e-182]
 [1.19291753e-011 1.74818819e-016 4.57077996e-131 ... 1.82261885e-119
  8.47369089e-182 1.16671047e-002]]
```

(e) Applying SVD decomposition on M, **write a function that selects the top 3 vectors (corresponding to the highest singular values) in the matrix U and performs the same KMeans clustering used above on them ; show the plots. What do you observe? Did it work? If so explain why, if not, explain not.**

Intuition: By selecting the top 3 vectors of the U matrix, we are selecting a new representation of the data points which could be seen as a construction of a low dimension embedding of the data points as mentioned in problem 3.

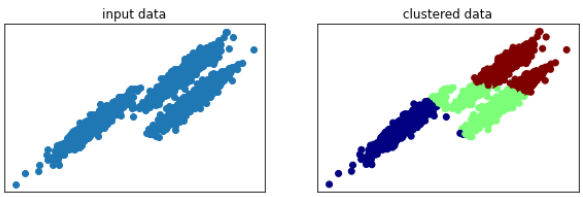
```
u, s, vh = np.linalg.svd(M)
print(u)
print(u.shape)
```

```
[[ 9.26984011e-18 -1.30790085e-17  3.84301385e-17 ...  1.06172766e-02
   2.05521108e-02 -3.50041922e-02]
 [ 1.19354373e-18  1.78850873e-19  2.08654741e-17 ...  7.48160916e-04
  -4.17357973e-03 -2.69947825e-03]
 [-4.29596594e-18 -9.70571095e-20 -5.39289044e-16 ... -3.45336946e-04
  -8.77242830e-04  2.80050100e-03]
 ...
 [-8.18496311e-23 -5.61058643e-20 -3.96947565e-17 ...  1.53410875e-03
  -1.12469028e-03 -4.93893347e-03]
 [-8.26133198e-20  5.32503587e-22 -8.45321972e-12 ... -7.12596828e-05
   4.66852945e-05  1.46498892e-04]
 [ 3.08911247e-20 -2.34916204e-18  1.32040827e-17 ... -5.88075906e-02
   3.85766169e-02  6.78202070e-02]]
(2000, 2000)
```

```
top3 = u[:, -3:]
print(s)
print(top3.shape)
```

```
[1.53702497e+03 3.56887304e+02 1.08405781e+01 ... 3.93046805e-14
 2.18420518e-14 4.13803147e-15]
(2000, 3)
```

```
y_pred = KMeans(n_clusters=3, random_state=seed).fit_predict(top3)
show_data_results(X, 2, y_pred)
# show_data_results(top3_vectors, num_plot=1)
```



It works, and it can classify the datapoint as before.

(f) Now let's think of the symmetrically normalized adjacency matrix obtained above as the transition matrix in of a Markov Chain. That is, it represents the probability of jumping from one node to another. In order to fully interpret M in such way, it needs to be a proper stochastic matrix which means that the sum of the elements in each column must add up to 1. **Write a function that takes in the matrix M and returns M_{stoch}, the stochastic version of M; compute the stochastic matrix.**
Using SVD decomposition on the newly obtained stochastic matrix M_{stoch}, **use your function in part (e) to select the top 3 vectors of the matrix U_{stoch} and perform the same KMeans clustering used above on them and show the plots. What do you observe? Did it work?**

Solution

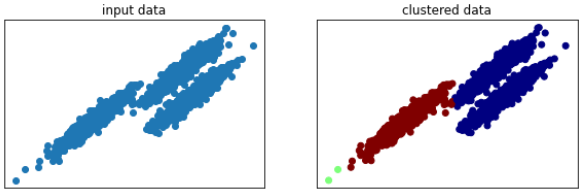
```
def stochastic_matrix_converter(M):
    # TODO - fill in your code here
    # degree_matrix =
    M_stoch = M / np.sum(M, axis = 0)
    return M_stoch
```

```
M_stoch = stochastic_matrix_converter(M)
print(M_stoch)
print(np.sum(M_stoch, axis=0))
```

```
[[1.63170918e-002 6.88941150e-017 4.97266299e-072 ... 8.27718837e-064
  1.78954874e-110 1.14085014e-011]
 [6.23916637e-017 2.79552720e-002 7.23389388e-118 ... 5.01277595e-111
  2.43282242e-166 1.68360862e-016]
 [5.50555184e-072 8.84381242e-118 9.38319264e-003 ... 3.06883629e-003
  8.33332516e-007 4.40193143e-131]
 ...
 [8.77436821e-064 5.86769173e-111 2.93829454e-003 ... 1.10141934e-002
  3.99296633e-009 1.75520974e-119]
 [1.63084575e-110 2.44732974e-166 6.85923769e-007 ... 3.43266614e-009
  3.46000826e-002 8.16866548e-102]
 [1.22569915e-011 1.98343149e-016 4.24181719e-131 ... 1.76659047e-119
  9.55381277e-102 1.12361118e-002]]
[1. 1. 1. ... 1. 1. 1.]
```

Using the method from Q.5 above, compute the top 3 U vectors on `M_stoch`, perform K Means clustering on it and plot your answers. What do you observe?

```
# TODO - compute y_pred_spectral_stoch below
u, sigma, vt = np.linalg.svd(M_stoch)
top3 = u[:, :3]
y_pred_spectral_stoch = KMeans(n_clusters=3, random_state=seed).fit_predict(top3)
show_data_results(X, 2, y_pred_spectral_stoch)
```



It can classify the data points well.

4. Graph Neural Networks

(a) **Tell which of these are valid functions for this node's computation of the next self-message $\mathbf{s}_i^\ell$. For any choices that are not valid, briefly point out why.**
Note: we are *not* asking you to judge whether these are useful or will have well behaved gradients. Validity means that they respect the invariances and equivariances that we need to be able to deploy as a GNN on an undirected graph.

(i) $\mathbf{s}_i^\ell = w_1 \mathbf{s}_i^{\ell-1} + w_2 \frac{1}{n_i} \sum_{j=1}^{n_i} \mathbf{m}_{i,j}^{\ell-1}$
(ii) $\mathbf{s}_i^\ell = \max(w_1^\ell \mathbf{s}_i^{\ell-1}, w_2 \mathbf{m}_{i,1}^{\ell-1}, w_3 \mathbf{m}_{i,2}^{\ell-1}, \ldots, w_{n_i-1} \mathbf{m}_{i,n_i}^{\ell-1})$ where the max acts component-wise on the vectors.
(iii) $\mathbf{s}_i^\ell = \max(w_1^\ell \mathbf{s}_i^{\ell-1}, w_2 \mathbf{m}_{i,1}^{\ell-1}, w_2 \mathbf{m}_{i,2}^{\ell-1}, \ldots, w_2 \mathbf{m}_{i,n_i}^{\ell-1})$ where the max acts component-wise on the vectors.

(i)(iii) are valid
(ii) is not valid, because for each node in GNN, the number of its neighbors are not deterministic so we can not know what the w_{n_i-1} exactly is.

(b) We are given the following simple graph on which we want to train a GNN. The goal is binary node classification (i.e. classifying the nodes as belonging to type 1 or 0) and we want to hold back nodes 1 and 4 to evaluate performance at the end while using the rest for training. We decide that the surrogate loss to be used for training is the average binary cross-entropy loss.

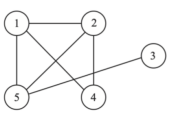


Figure 3: Simple Undirected Graph

nodes	1	2	3	4	5
y_i	0	1	1	1	0
$\hat{y}_i$	a	b	c	d	e

Table 1: y_i is the ground truth label, while $\hat{y}_i$ is the predicted probability of node i belonging to class 1 after training.

Table 1 gives you relevant information about the situation.
Compute the training loss at the end of training.

$-\frac{1}{3}(-\log b - \log c - \log(1-e))$

(c) Suppose we decide to use the following update rule for the internal state of the nodes at layer ℓ.

$$\mathbf{s}_i^\ell = \mathbf{s}_i^{\ell-1} + W_1 \frac{\sum_{j=1}^{n_i} \tanh\left(W_2 \mathbf{m}_{i,j}^{\ell-1}\right)}{n_i} \quad (12$$

where the tanh nonlinearity acts element-wise.
For a given node i in the graph, let $\mathbf{s}_i^{\ell-1}$ be the self-message for this node from the preceeding laye while the preceeding layer messages from the n_i neighbors of node i are denoted by $\mathbf{m}_{i,j}^{\ell-1}$ where ranges from 1 to n_i. We will use W with subscripts and superscripts to denote learnable weights i matrix form. If there's no superscript, the weights are shared across layers.

(i) **Which of the following design patterns does this update rule have?**
☐ Residual connection
☑ Batch normalization

BN

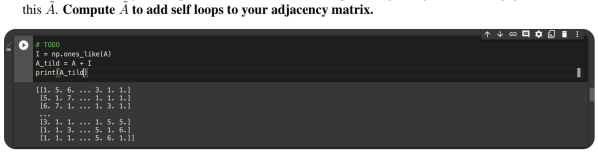
(ii) **If the dimension of the state s is d-dimensional and W_2 has k rows, what are the dimensions of the matrix W_1?**

$d \times k$

(iii) If we choose to use the state $\mathbf{s}_i^{\ell-1}$ itself as the message $\mathbf{m}^{\ell-1}$ going to all of node i's neighbors, **please write out the update rules corresponding to (12) giving $\mathbf{s}_i^\ell$ for the graph in Figure 3 for nodes $i = 2$ and $i = 3$ in terms of information from earlier layers.** Expand out all sums.

5. Zachary's Karate Club (Coding)

(a) Go through `q_zkc.ipynb`. We want our network to be aware of information about the nodes themselves instead of only the neighborhood, so we add self loops our adjacency matrix. The paper called this $\tilde{A}$. **Compute $\tilde{A}$ to add self loops to your adjacency matrix.**



(b) **Write the unnormalized adjacency A, the degree matrix, D, and the symmetrically normalized adjacency matrix, $A^{SymNorm}$, of the graph in Figure. 1.**

```python
def get_degree_matrix(adjacency_matrix):
    # TODO - fill in your code here
    degree_matrix = np.zeros_like(adjacency_matrix)
    X, Y = adjacency_matrix.shape
    for i in range(X):
      for j in range(Y):
        if i != j:
          degree_matrix[i][i] = degree_matrix[i][i] + adjacency_matrix[i][j]

    return degree_matrix
```

```python
[23] def get_adjacency_matrix(A_tild):
    D = get_degree_matrix(A_tild)
    A_hat = np.linalg.inv(sqrtm(D)) @ A_tild @ np.linalg.inv(sqrtm(D))
    return A_hat
```

```python
A_hat = get_adjacency_matrix(A_tild)
A_hat
```