Data Mining - Handin 2 - Graph mining

This handin corresponds to the topics in Week 10-15 in the course.

The handin is

- done in the chosen handin groups
- worth 10% of the grade

For the handin, you will prepare a report in PDF format, by exporting the Jupyter notebook. Please submit

1. The jupyter notebook file with your answers
2. The PDF obtained by exporting the jupyter notebook

Submit both files on Brightspace no later than **April 21 kl. 11.59PM**.

The grading system: Tasks are assigned a number of points based on the difficulty and time to solve it. The sum of the number of points is **100**. For the maximum grade you need to get at least *80 points*. The minimum grade (02 in the Danish scale) requires **at least** 30 points, with at least 8 points on of the first three Parts (Part 1,2,3) and 6 points in the last part (Part 4). Good luck!

The exercise types: There are three different types of exercises

1. **[Compute by hand]** means that you should provide **NO code**, but show the main steps to reach the result (not all).
2. **[Motivate]** means to provide a short answer of 1-2 lines indicating the main reasoning, e.g., the PageRank of a complete graph is 1/n in all nodes as all nodes are symmetric and are connected one another.
3. **[Describe]** means to provide a potentially longer answer of 1-5 lines indicating the analysis of the data and the results.
4. **[Prove]** means to provide a formal argument and NO code.
5. **[Implement]** means to provide an implementation. Unless otherwise specified, you are allowed to use helper functions (e.g., `np.mean`, `itertools.combinations`, and so on). However, if the task is to implement an algorithm, by no means a call to a library that implements the same algorithm will be deemed as sufficient!

In [ ]:
```
### BEGIN IMPORTS - DO NOT TOUCH!
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
import sys
sys.path.append('..')
#!{sys.executable} -m pip install matplotlib
#!{sys.executable} -m pip install networkx
#!{sys.executable} -m pip install torchvision
import random
import scipy.io as sio
import time
```

```
import networkx as nx
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt

import csv
from itertools import count


import torch
import torch.optim
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms


from utilities.load_data import load_mnist
import utilities.email as email
from utilities.mnist import *

from utilities.make_graphs import read_edge_list, read_list, load_data

### END IMPORTS - DO NOT TOUCH!
```

Task 1.1 Random walks and PageRank (12 points)

In this exercise recall that the PageRank is defined as

$$\mathbf{r} = \alpha \mathbf{M} \mathbf{r} + (1 - \alpha)\mathbf{p}$$

where $\mathbf{r} \in \mathbb{R}^n$ is the PageRank vector, α is the restart probability, $\mathbf{M} = A\Delta^{-1}$, and $\mathbf{p}$ is the restart (or personalization) vector.

Task 1.1.1 (4 points)

What is the PageRank of a d**-regular** graph with n nodes and $\alpha = 1$?

[Motivate] your answer without showing the exact computation.

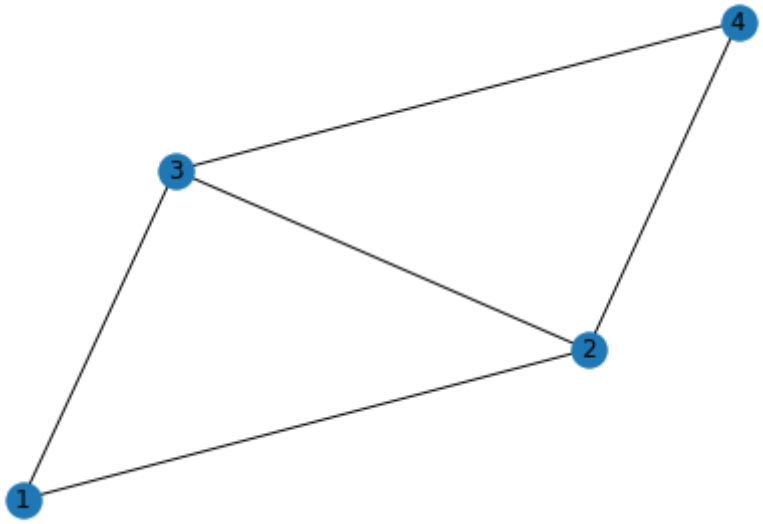
Answer:

As we have that $\alpha = 1$, we get that $r = Mr$. Therefore we can find the PageRank of the graph by finding its eigenvector corresponding the the eigenvalue of 1. As we have a d-regular graph, we will notice that all nodes become symmetrical. And therefore, we will find that the eigenvector will be the same for all nodes. This means that the PageRank of all nodes will be the same. And as we have n nodes, we will find that the PageRank of all nodes will be $1/n$.

Task 1.1.2 (6 points)

Look at the graph below (run the code) and try make a guess about the PageRank values of each node only by reasoning on the graph's connections.

In [ ]:
```
G = nx.Graph()
G.add_edges_from([(1,2), (2,3), (2,4), (3,4), (1,3)])
nx.draw(G, with_labels=True, )
```



A) [Implement] the PageRank for $\alpha = 1$ for the graph using the Power Iteration method (use $\epsilon = 1e - 16$ to stop the iteration).

B) [Implement] Plot the norm square difference of the r vector (between any two consecutive iterations) for each iteration.

C) [Motivate] Do you observe a constant decrease of the norm square difference as iterations are increasing, and is this decrease expected or not?

D) [Implement] the PageRank for $\alpha = 1$ using the eigenvector method.

E) [Motivate] Are solutions of both methods the same? Why don't we only use the eigenvector method that optimally solves the problem?

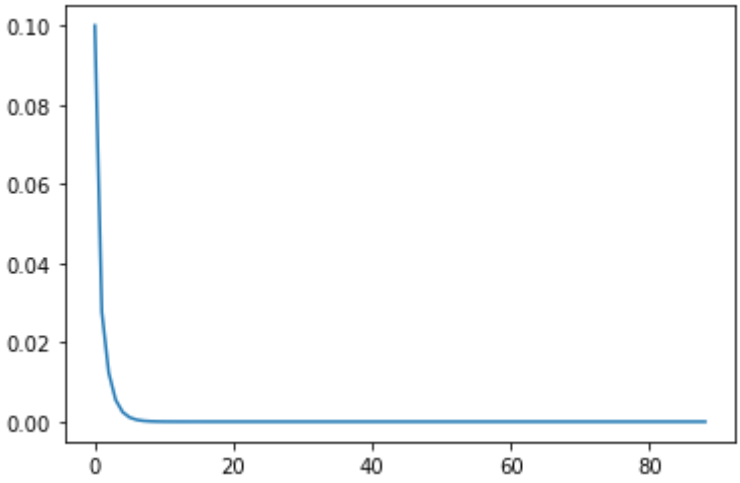
F) [Motivate] Do the real vector match with your first guess? Can you see a pattern between the pagerank score of each node and its edges?

In [ ]:
```
#A) YOUR CODE HERE

# Initialize

G_use = list(G.edges) - np.array([1,1])

x = list(zip(*G_use))[0]
y = list(zip(*G_use))[1]
nodes = list(set(x+y))
NumNodes = len(nodes)

# Create adjacency matrix
A = np.zeros((len(nodes), len(nodes)))

for index in G_use:
```

```
    A[index[0],index[1]] = 1
    A[index[1],index[0]] = 1

# Create degree matrix
D = np.zeros((len(nodes), len(nodes)))

for i in range(len(nodes)):
    D[i,i] = np.sum(A[i,:])

# Create the transition matrix
M = A.T@np.linalg.inv(D)

# Set equal weight for all nodes
r = np.array([1/NumNodes]*NumNodes)

# Keep track of the history
history = [0.1]

# Start the Power Iteration
update = True
while update:
    # Update the rank
    r_new = M@r

    # Get the norm of the difference
    history.append(np.linalg.norm(r_new-r)**2)

    # Check if the rank is converged
    if np.linalg.norm(r_new-r) < 1e-16:
        update = False
    r = r_new

print(r)
```

```
[0.2 0.3 0.3 0.2]
```

In [ ]:
```
#B) YOUR CODE HERE

plt.plot(history)
plt.show()
```



C) Answer:

Yes we do. And this is expected, as we are starting with an equal assignment to each node, even though that we do not have that all the nodes are symmetrically connected. Therefore,

with each iteration, we will find that the nodes with more connections will have a higher PageRank, as they will have a higher probability of being visited (Given that the transition matrix puts equal weight on each outgoing node). So for each iteration, we will get closer to the true PageRank of each vertex, thus the norm square difference will decrease.

D) **Answer:**

As we again have that $\alpha = 1$, we get that $r = Mr$. Therefore we can find the PageRank of the graph by finding its eigenvector corresponding to the eigenvalue of 1. Thus we get that $\lambda r = Mr$.

In [ ]:
```
from sympy import Eq, solve, Symbol

#First, we need to define the variables that we want to solve for, which is the vec
variablenames = []
for i in range(len(nodes)):
    variablenames.append('x' + str(i+1))


#We now need to define the symbols for the variables using the sympy library where
#can use in the equations
for i in range(len(nodes)):
    variablenames[i] = Symbol(variablenames[i])


#We now need to define the equations that we want to solve.
#The eigenvalue equation is given by Mx = λx where M is the transition matrix and x
#In our case, we want to solve for the eigenvector x, where $\lambda = 1$.
# We do this by calculating left hand side of the equation.

# We do this by multiplying the transition matrix with the vector of x. These are t
LHS=M@variablenames

#We want to define n equations +1 to solve for n variables and also take into count
#We do this by defining the equations as a list of equations and then solve them us
equation= [0]*(len(LHS)+1)
for i in range(len(LHS)):
    equation[i] = Eq(LHS[i], variablenames[i])

#We define the last equation such that the sum of the eigenvector is 1.
equation[-1] = Eq(sum(variablenames), 1)



# We can now solve the equations for the variables
solutions = solve(equation, variablenames)

print(solutions)
```

```
{x1: 0.200000000000000, x2: 0.300000000000000, x3: 0.300000000000000, x4: 0.200000
000000000}
```

E) **Answer:**

The Eigenvector method finds the eigenvector of M directly using linear algebra techniques, whereas the Power Iteration method approximates the eigenvector iteratively using the nature of the transition matrix. The Power Iteration method is simpler and requires less computational resources, but may converge slowly for very large graphs. The

Eigenvector method, on the other hand, is more computationally expensive but may be faster for larger graphs and may provide more accurate results.

So, while both methods can be used to solve the PageRank problem, the choice of method depends on the size of the graph and the desired level of accuracy. For small graphs, the Power Iteration method may be sufficient, whereas for larger graphs, the Eigenvector method may be more appropriate.

Therefore, both methods have their advantages and disadvantages, and the choice of method depends on the specific requirements of the application.

F) **Answer:**

Assuming that the question reffers to the real vector as the answer found in question A) and D), and it asks if it matches our guess from task 1.1.1, our guess of $1/n$ (which was not a guess, as we had a d-regular graph), then no. And it has no reason to match, as we have that the graph in task 1.1.2 is not symmetrically connected. And therefore, the PageRank of each node will not be the same.

Yes, we see a clear correlation between the PageRank of each node and the number of edges it has. The nodes with more edges have a higher PageRank, as they have a higher probability of being visited.

Task 1.1.3 (2 points)

[Motivate]

Assume you have embedded the graph in **1.1.2** with a **Linear Embedding** using unormalized Laplacian matrix of the graph as the similarity matrix. How do you expect the embeddings to be if the embedding dimension is $d = 1$? (1) Check the correct box below and (2) motivate your answer.

- ☐ Nodes 1, 2, 3, 4 will be placed in the corners of a hypercube
- ☑ Nodes 2,3 will have the same embedding while 1,4 will be far from each other.
- ☐ Nodes 1,4 and 2,3 will have very close embeddings.
- ☐ Nodes 3,4 will be very far apart.

IMPORTANT: Do NOT just choose one answer. Please clarify WHY this is the correct answer.

Answer:

As per the lecture notes from lecture 9, "If two nodes are “close” under some definition of similarity, they should be also similar in the embedding space." And so, we see that in the Laplacian matrix, the diagonal elements are the degree of each node. And therefore, we see that the nodes with more edges will have a higher degree, and will be closer to each other in the embedding space. And so, we see that nodes 2 and 3 will have the same embedding, as they have an edge to every node, while as seen in the laplacian matrix entry (1,4) = 0. Meaning that nodes 1 and 4 don't have an edge to each other, and will therefore be the

nodes laying furthest away from all others, 'traveling wise', as they have no direct connection to each other. Thus we conclude that 2 and 3 will have the same embedding, while 1 and 4 will be far from each other.

Task 1.2: Spectral Properties of the Graph Laplacian (17 points)

[Prove] the following properties: You will be given points for each of the properties that you prove, rather than points for the exercise as a whole.

Note that all question correspond to the eigenvalues of the LAPLACIAN (NOT THE NORMALIZED)

For a graph with n nodes the eigenvalues of the LAPLACIAN ($L = D - A$) is sorted in ascending order, i.e.,
$\lambda_0 \leq \lambda_1 \leq \ldots \leq \lambda_{n-1}$

Task 1.2.1 (1 points)

For all graphs $\lambda_0 = 0$

Answer:

We know that the Laplacian matrix is defined as $L = D - A$, where D is the diagonal degree matrix and A is the adjacency matrix of the graph. The diagonal elements of D are the degrees of the nodes, i.e., $d_i = \sum_{j=1}^{n} a_{ij}$, where a_{ij} is the element in the i'th row and j'th column of A. The Laplacian matrix L is therefore symmetric and has the property that the sum of each row (or column) is zero. From the lecture notes from lecture 6, we have that for any symmetrical matrix, it must hold that all the eigenvalues are non-negative, i.e. $\lambda_i \geq 0$. And so, if we can find a vector $v \neq 0$ such that $Lv = 0$, for a general laplacian matrix L, then we have that $\lambda_0 = 0$ for all laplacian matricies.

Let L be a laplacian matrix, $v \neq 0$ be a vector. Then let's consider the equation $Lv = \lambda v$. Looking at the left hand side, we can rewrite it as follows:

$$Lv = (D - A)v = Dv - Av.$$

Assume that $v = \bar{1}$, where $\bar{1}$ is a vector of ones. Then we have that $Dv = d$, and $Av = d$, where d if a vector of the degrees of the nodes as desbribed above. This implies that

$$Lv = d - d = 0.$$

And in order for the equation to hold, we must have that $\lambda = 0$. Thus we have that $\lambda_0 = 0$ for all laplacian matricies.

It is worth noting, that this holds for all constant vectors v, and not just $\bar{1}$.

Task 1.2.2 (2 points)

For the complete graph, $\lambda_1, \ldots, \lambda_{n-1} = n$

Answer:

To prove that for the complete graph, $\lambda_1, \ldots, \lambda_{n-1} = n$, we need to compute the Laplacian matrix L and its eigenvalues. The Laplacian matrix of a complete graph with n nodes is given by:

$$L = D - A = \begin{pmatrix} n-1 & 0 & \cdots & 0 \\ 0 & n-1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & n-1 \end{pmatrix} - \begin{pmatrix} 0 & 1 & \cdots & 1 \\ 1 & 0 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 0 \end{pmatrix} = \begin{pmatrix} n-1 & -1 \\ -1 & n-1 \\ \vdots & \vdots \\ -1 & -1 \end{pmatrix}$$

To compute the eigenvalues, we can use the fact that L is a symmetric matrix, so it has n orthogonal eigenvectors that form a basis of $\mathbb{R}^n$. We can find these eigenvectors and eigenvalues using standard linear algebra techniques.

First, note that the sum of the elements in each row of L is 0, so the vector of all ones is an eigenvector of L with eigenvalue 0. This is the smallest eigenvalue, which we also proved before

Next, we need to find an eigenvector, which namely needs to be orthogonal to our first eigenvector corresponding to the smallest eigenvalue 0, i.e. $f = \bar{1}$. For two vectors to be orthogonal, it needs to hold that:

$$f \cdot x = 0 \iff \sum_{i=1}^{n} f_i \cdot x_i = 0 \iff \sum_{i=1}^{n} x_i = 0$$

where f is the first eigenvector and x is any other eigenvector.

For any such vector x, we have the following relationship to the unnormalized laplacian:

$$\begin{aligned} [Lx]_i &= [(D-A)x]_i && (1) \\ &= \sum_{j=1}^{n} d_{ij}x_j - a_{ij}x_j && (2) \\ &= d_{ii}x_i - \sum_{j=1}^{n} a_{ij}x_j && (3) \\ &= d_{ii}x_i - \sum_{j \neq i} a_{ij}x_j && (4) \\ &= (n-1)x_i - \sum_{j \neq i} a_{ij}x_j && (5) \\ &= nx_i - \underbrace{\left(x_i + \sum_{j \neq i} a_{ij}x_j \right)}_{=0} && (6) \\ &= nx_i && (7) \end{aligned}$$

Where in equation (3), we have used that $d_{ij} = 0 \, \forall \, i \neq j$, so the only term surviving is the diagonal.
And in (4), we have used that $a_{ij} = 0 \, \text{for} \, i = j$, and therefore we may take it out of the sum.
In equation (5), we use the fact that, in a complete graph, the degree of every node is $n - 1$.
Finally in (6), we have used the fact from above that the sum of the eigenvector x is zero, and as we have a complete graph, A will be a matrix of ones, except for the diagonal. Therefore, when we add x_i to the sum, we will get a sum of zero.

Which means that we have $Lx = nx$ for any general eigenvector and therefore it must be that $\lambda_1, \ldots, \lambda_{n-1} = n$.

Task 1.2.3 (3 points)

For all the graphs with k connected components $\lambda_0 = \lambda_1 = \ldots = \lambda_k = 0$

Answer:

We have from task 1.2.1, that for all graphs, $\lambda_0 = 0$. And so we only need to prove that $\lambda_1 = \lambda_2 = \cdots = \lambda_k = 0$. As we have k connected components, we can write the Laplacian matrix as:

$$L = \begin{pmatrix} L_1 & 0 & \cdots & 0 \\ 0 & L_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & L_k \end{pmatrix}$$

Where L_i is the Laplacian matrix of the i'th connected component. Thereby, we can easily see that, by defining an eigenvector v that has 1's on all the nodes in the i'th component, and 0's everywhere else, we have that $Lv = 0$ for all i. And so we have that $\lambda_1 = \lambda_2 = \cdots = \lambda_k = 0$. And as these are all different from zero, we get that they are all valid eigenvectors corresponding to the eigenvalue 0, whilst still being orthogonal to each other by construction.

Task 1.2.4 (5 points)

Given a graph G with eigenvalues of the laplacian $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$.
We randomly remove an edge from G and we re-calculate the eigenvalues as $\lambda'_0, \lambda'_1, \ldots, \lambda'_{n-1}$.
Can we have $\lambda'_i > \lambda_i$ for some $0 \leq i \leq n - 1$? Why? Why not?

Answer:

First let us denote the eigenvalues as $\lambda_1, \lambda_2, \ldots, \lambda_n$ for easier notation.

Let X_i be a unit eigenvector of L corresponding to the eigenvalues λ_i, where $X_1, \ldots, X_k$ are the linearly independent and orthogonal to each other. Let S_k be the set of all eigenvectors. Then, if we have a $x \in S_k$, then there exist real numbers $a_1, \ldots, a_k$ such that $x = a_1X_1 + \ldots + a_kX_k$, i.e. it can be written as a linear combination of all the already existing eigenvectors. We know that $X_i^2 = 1$ thus we have that $x^Tx = \sum_{i=1}^{k} a_i^2 = 1$. Now we get the following:

$$\begin{aligned}
x^TL'x &= (a_1X_1 + \ldots + a_kX_k)^TL'(a_1X_1 + \ldots + a_kX_k) \\
&= (a_1X_1 + \ldots + a_kX_k)^TL(a_1X_1 + \ldots + a_kX_k) \\
&- (a_1X_1 + \ldots + a_kX_k)^TH(a_1X_1 + \ldots + a_kX_k) \\
&\leq (a_1X_1 + \ldots + a_kX_k)^TL(a_1X_1 + \ldots + a_kX_k) \\
&= (a_1X_1 + \ldots + a_kX_k)^T(\lambda_1a_1X_1 + \ldots + \lambda_ka_kX_k) \\
&= \lambda_1a_1^2X_1^TX_1 + \ldots + \lambda_ka_k^2X_k^TX_k \\
&\leq \lambda_ka_1^2X_1^TX_1 + \ldots + \lambda_ka_k^2X_k^TX_k \\
&= \lambda_k(a_1^2 + \ldots + a_k^2) \\
&= \lambda_kx^Tx
\end{aligned}$$

Thereby, from the fact that

$$\lambda = \min \frac{x^TLx}{x^Tx},$$

where $x \neq 0$ and all eigenvectors are orthogonal to eachother. We can thus conclude that, as $x^TL'x \leq \lambda_kx^Tx$ for all $x \in S_k$, then we have that $\lambda'_1 \leq \ldots \leq \lambda'_k \leq \lambda_k$. And so we have that $\lambda'_0 \leq \lambda_0 \leq \lambda'_1 \leq \lambda_1 \leq \cdots \leq \lambda'_{n-1} \leq \lambda_{n-1}$.

Task 1.2.5 (6 points)

Suppose that the graph G consists of two connected componentes of equal size named G_1 and G_2. For simplicity assume that n is even.
The Laplacian of G_1 has eigenvalues $\lambda_0^1, \lambda_1^1, \ldots, \lambda_{n/2-1}^1$.
The Laplacian of G_2 has eigenvalues $\lambda_0^2, \lambda_1^2, \ldots, \lambda_{n/2-1}^2$.
Prove that the Laplacian of G is consisted of the eigenvalues of the Laplacians of G_1 and G_2 in ascending order.

Answer:

Let L be the laplacian of the graph G

$$L = \begin{pmatrix} L_1 & 0 \\ 0 & L_2 \end{pmatrix}$$

with the corresponding eigenvalues $\mu_0 \leq \mu_1 \leq \ldots \leq \mu_{n-1}$.

Now, we can use the Cauchy Interlacing Theorem, which states that for any Hermitian matrix A of size $n \times n$ with eigenvalues $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ and any principal submatrix B of A of size $m \times m$ with eigenvalues $\mu_1 \leq \mu_2 \leq \cdots \leq \mu_m$, we have:

$$\lambda_i \leq \mu_i \leq \lambda_{n-m+i} \text{ for } i = 1, 2, \ldots, m.$$

Where, in our case, we have that our laplacian L is a Hermitian matrix of size $n \times n$, and our principal submatrix B is the laplacian of the first connected component L_1 of size $n/2 \times n/2$. Which means that we get the following:

$$\mu_i \leq \lambda_i^1 \leq \mu_{(n-1)-(n/2-1)+i} \text{ for } i = 0, 1, \ldots, n/2 - 1.$$

Similarily, we can use the Cauchy Interlacing Theorem again, but now with the laplacian of the second connected component L_2 of size $n/2 \times n/2$ as our principal submatrix B, and we have

$$\mu_i \leq \lambda_i^2 \leq \mu_{(n-1)-(n/2-1)+i} \text{ for } i = 0, 1, \ldots, n/2 - 1.$$

And thereby, we have that the eigenvalues of the Laplacian of the graph G are sandwiched between the eigenvalues of the Laplacians of its two connected components, G_1 and G_2, in ascending order. This implies that the eigenvalues of G are a merge of the eigenvalues of G_1 and G_2 in ascending order. Thus the eigenvalues of the laplacian of the graph G is consisted of the eigenvalues of the laplacians of G1 and G2 in ascending order.

Part 2: Graphs and Spectral clustering

In this part, you will experiment and reflect on spectral clustering as a technique for partitioning a graph.

Task 2.1: ε-neighbourhood graph (10 points)

In this subsection you will experiment with some biological data https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003268.

!IMPORTANT! First run the following code to load the data.

In [ ]:
```
#Load Data
from utilities.make_graphs import read_edge_list, read_list, load_data
import numpy as np
X, Y = load_data()
```

Task 2.1.1 (4 points)

[Implement] the ε-neighborhood graph, using Eucledian (L_2) distance.

Note: Be sure that your constructed graph does not contain self-loop edges (edges from i to i for each i)

In [ ]:
```
#YOUR CODE HERE
# Be sure that your constructed graphs does not
# contain loop edges (edges from i to i for some node i)
```

```
def nn_graph(data, eps, remove_self=True, directed=False):
    n = len(data)
    G = nx.Graph()
    if directed:
        G = nx.DiGraph()


    ### YOUR CODE STARTS HERE

    nodes = [point for point in range(len(data))]


    for i in nodes:
        G.add_node(i)
        for j in nodes:
            if i != j:
                if np.linalg.norm(data[i] - data[j]) < eps:
                    G.add_edge(i,j)
    ### YOUR ENDS CODE HERE
    return G


G = nn_graph(X,0.05)

print(G)
```

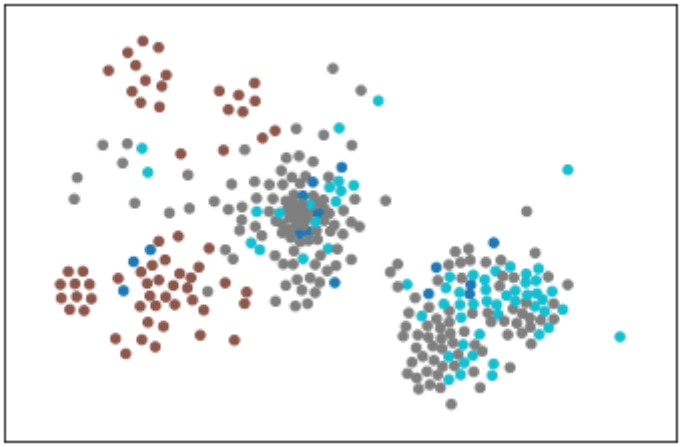
```
Graph with 307 nodes and 1055 edges
```

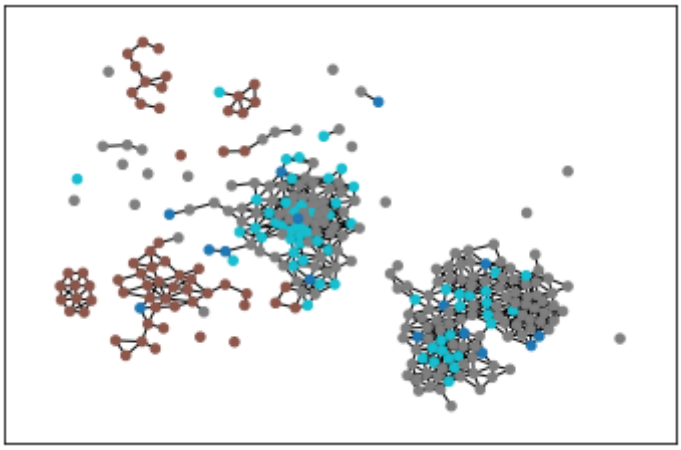
Task 2.1.2 (2 points)

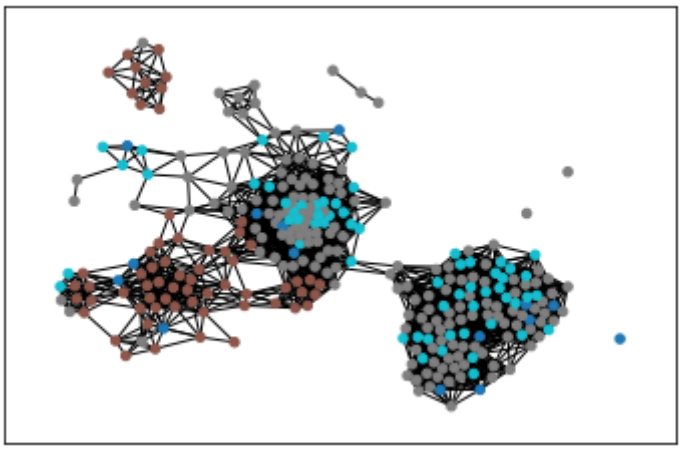
Try with different ε values (select a small set of ε, e.g., 0.01-0.5 values) and plot the graphs.

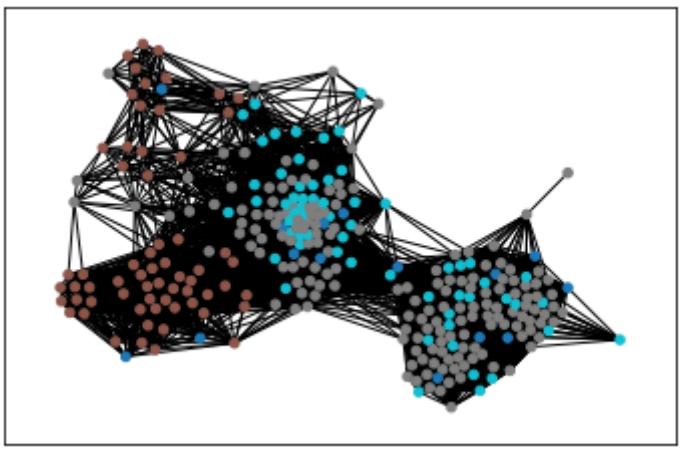
[Motivate] what you observe as epsilon increases.

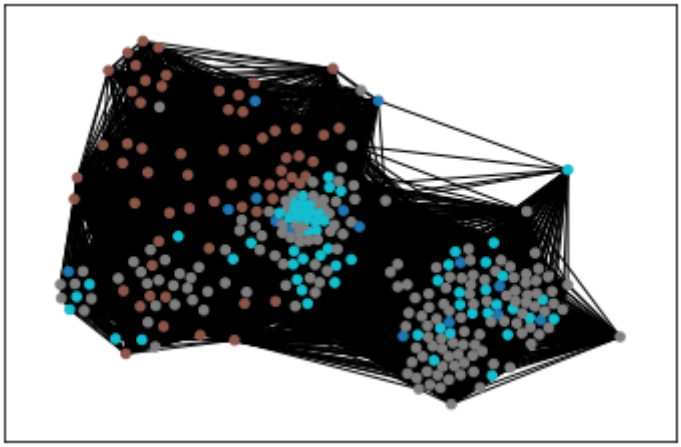
In [ ]:
```
### Run the code below
eps_values = [0.01, 0.05, 0.1, 0.2, 0.4]

for eps in eps_values:
    ax=plt.subplot()
    ax1=plt.subplot()
    G = nn_graph(X, eps)
    pos=nx.spring_layout(G)
    nx.draw_networkx_edges(G,pos=X)
    nx.draw_networkx_nodes(G, pos=X, node_color=Y, node_size=20, cmap=plt.get_cmap(
    ax.set_xlim(-0.1, 1.1)
    ax.set_ylim(-0.1, 1.1)
    plt.show()
```











YOUR ANSWER HERE

When epsilon increases more edges are added to the graph. Epsilon is the threshold for the distance between two nodes. When epsilon is small, the threshold is small and therefore only nodes that are close to each other are connected. When epsilon is large, the threshold is large and therefore more nodes are connected.

Task 2.1.3 (2 points)

Assign to each edge in the ε-neighborhood graph a weight

$$W_{ij} = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{t}}$$

[Implement] the function `weighted_nn_graph` below that returns the weighted graph given the data matrix in input and the values eps and t, where t is the parameter of the equation above.

In [ ]:
```
import math
def weighted_nn_graph(data, eps=20, t=0.1):
    n = len(data)
    G = nx.Graph()
    ### YOUR CODE STARTS HERE

    nodes = [node for node in range(len(data))]

    for i in nodes:
        G.add_node(i)
        for j in nodes:
            if i != j:
                if np.linalg.norm(data[i] - data[j]) < eps:
                    w = math.exp(-(np.linalg.norm(data[i]-data[j]))/t)
                    G.add_edge(i,j, weight = w)
    ### YOUR CODE ENDS HERE
    return G

print(weighted_nn_graph(X))
```

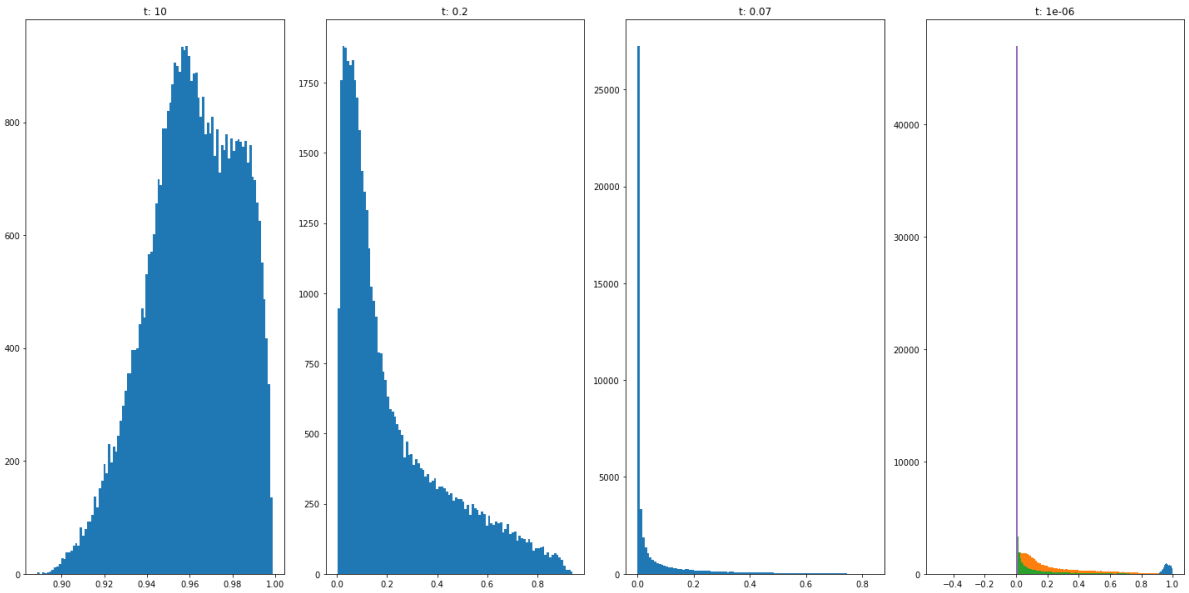
```
Graph with 307 nodes and 46971 edges
```

Task 2.1.4 (2 points)

Vary $t \in \{10, 0.1, 0.000001\}$. Plot the weights as a histogram using the code below in order to analyse the results using the provided code.
What happens when t is very small, close to 0, i.e., $t \to 0$?
What happens when t is very large?
Is the behaviour with $t = 0$ expected?

[Motivate] your answer reasoning on the formula.

In [ ]:
```
ts = [10, 0.2, 0.07, 0.000001]
fig, ax = plt.subplots(1,4, figsize=(20, 10))
row = 0

for i, t in enumerate(ts):
    G = weighted_nn_graph(X, eps=60, t=t)
    ys = []


    col = i
    for i, d in enumerate(G.edges.data()):
        ys.append(d[2]['weight'])
    plt.hist(ys, bins=100)
    ax[col].hist(ys, bins=100)
    ax[col].set_title("t: "+str(t))

plt.tight_layout()
```



YOUR ANSWER HERE The plots show the distance between the nodes and the weights of the edges between them.

When t is very small, close to 0, the weights are very small. When t is very large, the weights are very large. This is expected because the weights are calculated using the formula $W_{ij} = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{t}}$.

If we start by looking at the first graph of t=10, we can see that the weights are small. This is because the denominator is very large. and when the distance between the nodes is small, the weights become small.

When we look at the second graph of t=0.1, the weights are larger for smaller distances. This is because the denominator is greater than 0 and less than 1. When the distance goes towards 0 the exponent will go towards 0 which will result in a larger weight. When the distance between two nodes becomes larger the exponent goes towards -inf because the numerator increases which results in smaller weights when distance is increased. This is also the case for the third graph rest of the graph. Now that t is even smaller the exponent diverge towards -inf faster when distance is increased.

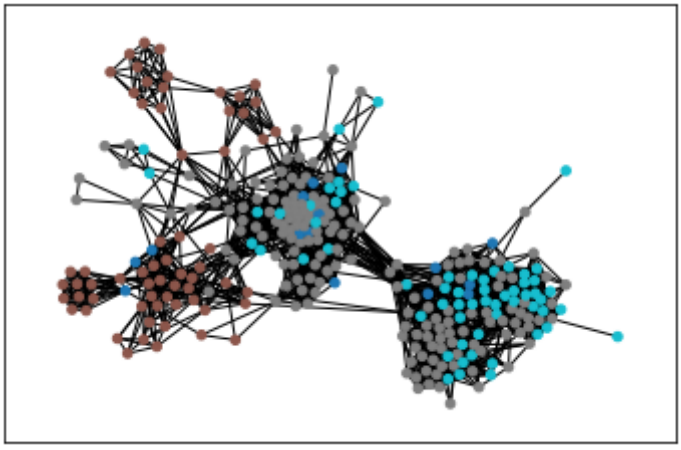
Task 2.2: Spectral clustering (20 points)

We will now look at spectral clustering and its properties.
For this Task we will use a subgraph from malaria_genes.
Note that this dataset is the biological network of the data used in Task 2.1.

Task 2.2.1 (5 point)

Run the code below to load and visualize the network.
By only observing the below plot and the nn-plots (nearest-neighbor plots)of task 2.1.2, which ε values seems to better approximate the real network? (just think of the answer you don't have to write something)

In [ ]:
```
edgelist = read_edge_list('./data/edges.txt')
n = np.max(edgelist)+1
G = nx.Graph()
for i in range(n):
    G.add_node(i)
for edge in edgelist:
    G.add_edge(edge[0], edge[1])
pos=nx.spring_layout(G)
nx.draw_networkx_edges(G,pos=X)
nx.draw_networkx_nodes(G, pos=X, node_color=Y, node_size=20, cmap=plt.get_cmap('tab
plt.show()
```



Now you are having the real network, lets check how good nn-graph (and for which ϵ value) is a good "approximates"the real graph.

A) **[Implement]** a function that calculates the absolute edge difference between the real network G and the one ϵ-neighborhood graph. Note that in order to do that you have to follow two steps:

1. In the first step you have to check if an edge in the real graph is also presented in the nn-graph, if not you increase the counter
2. In the second step you follow the opposite direction, that is you check if for every edge of the nn-graph if is also presented in the original one, if not you increase the counter. (Faster way just use the adjacency matrices)

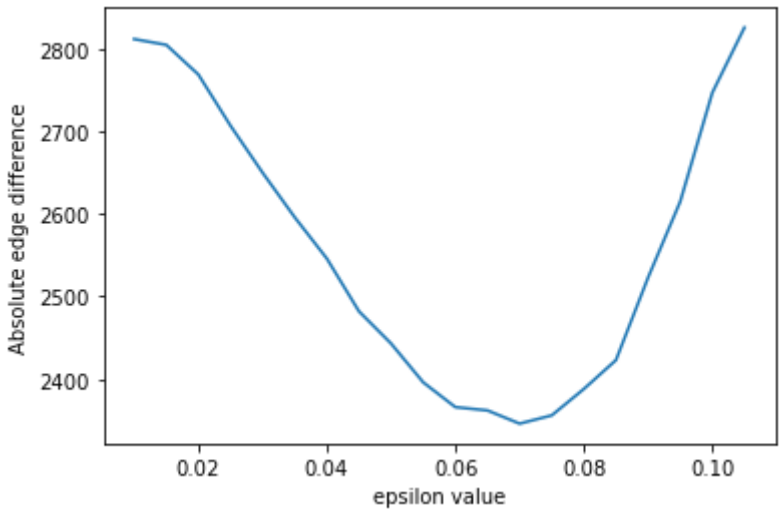
B) **[Implement]** Plot the edge-difference plot for the range of epsilon values in the range[0.01, 0.11] with step = 0.005.
C) **[Motivate]** By observing the plot it seems that there exists only one global minimum and no local minimum. Try to prove/disprove this intuition.

In [ ]:
```
#A) YOUR CODE HERE

def edge_difference(G,eps_graph):
    counter = 0

    for edge in G.edges:
        if edge not in eps_graph.edges:
            counter += 1

    for edge in eps_graph.edges:
        if edge not in G.edges:
            counter += 1

    return counter

e_graph = nn_graph(X,0.1)
print(edge_difference(G,e_graph))
```

```
2747
```

In [ ]:
```
#B) YOUR CODE HERE
epsilon = []
difference = []

for eps in np.arange(0.01, 0.11, 0.005):
    e_graph = nn_graph(X,eps)
    epsilon.append(eps)
    difference.append(edge_difference(G,e_graph))

print(np.where(difference == np.min(difference)))
print(epsilon[12])


plt.plot(epsilon, difference)
plt.xlabel('epsilon value')
plt.ylabel('Absolute edge difference')
plt.show()
```

```
(array([12], dtype=int64),)
0.06999999999999999
```



C) YOUR ANSWER HERE

By observing the plot we realize that there is only one global minimum and no local minimum. This make sense since we have a optimal value of epsilon that is the one that minimizes the edge difference. In our case $\epsilon \approx 0.07$ which we also could approximately dedeuce from the graphs in task 2.1.2 since eps=0.1 looks like the real network G the most. It makes sense that there is one global minimum because when epsilon is too small the graph does not contain enough edges to be similar to the real graph and when epsilon is too large the graph contains too many edges. Therefore, there is only one global minimum.

Task 2.2.2 (2 points)

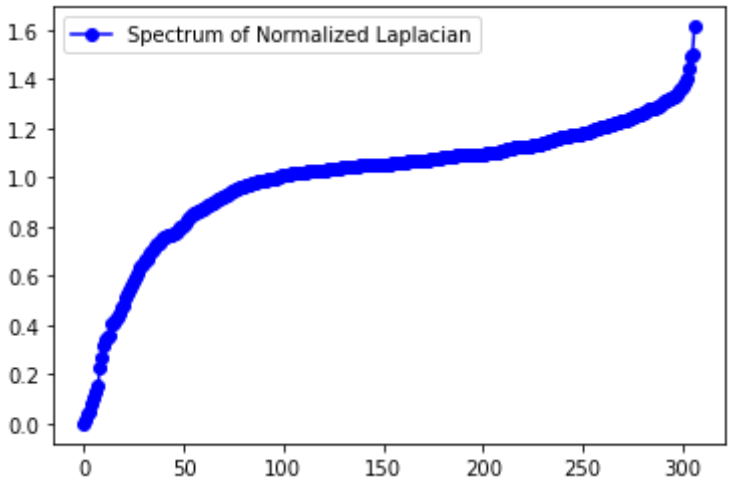
Compute the eigenvectors and eigenvalues (using the provided function) of the Normalized Laplacian and the Random Walk Laplacian of the graph G.
Plot the spectrum (eigenvalues).

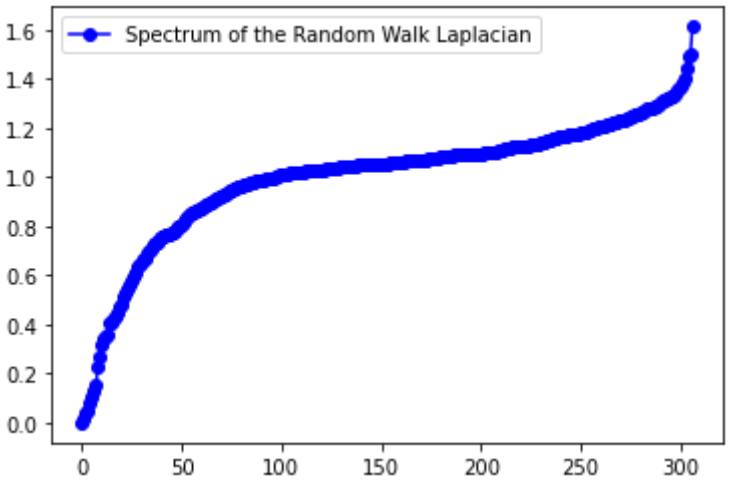
[Implement] the code to compute the different Laplacians.

In [ ]:
```python
def graph_eig(L):
    """
        Takes a graph Laplacian and returns sorted the eigenvalues and vectors.
    """
    lambdas, eigenvectors = np.linalg.eig(L)
    lambdas = np.real(lambdas)
    eigenvectors = np.real(eigenvectors)

    order = np.argsort(lambdas)
    lambdas = lambdas[order]
    eigenvectors = eigenvectors[:, order]

    return lambdas, eigenvectors
```

In [ ]:
```python
L_norm = None
L_rw = None
```

```
### YOUR CODE STARTS HERE

nodes = G.nodes
NumNodes = len(nodes)

# Create adjacency matrix
A = np.zeros((len(nodes), len(nodes)))
for index in G.edges:
    A[index[0],index[1]] = 1
    A[index[1],index[0]] = 1

# Create degree matrix
D = np.zeros((NumNodes, NumNodes))

for i in range(NumNodes):
    D[i,i] = np.sum(A[i,:])


identity_matrix = np.identity(len(nodes))
# Create Normalized Laplacian matrix
L_norm = identity_matrix - np.linalg.inv(D)**(1/2) @ A @ np.linalg.inv(D)**(1/2)



# Create Random Walk Laplacian matrix
L_rw = identity_matrix - np.linalg.inv(D) @ A


### YOUR CODE ENDS HERE


eigval_norm, eigvec_norm = graph_eig(L_norm)
eigval_rw, eigvec_rw = graph_eig(L_rw)

plt.figure(0)
plt.plot(eigval_norm, 'b-o', label='Spectrum of Normalized Laplacian', )
plt.legend()
plt.figure(1)
plt.plot(eigval_rw, 'b-o', label='Spectrum of the Random Walk Laplacian')
plt.legend()
plt.show()
```





Task 2.2.3 (4 points)

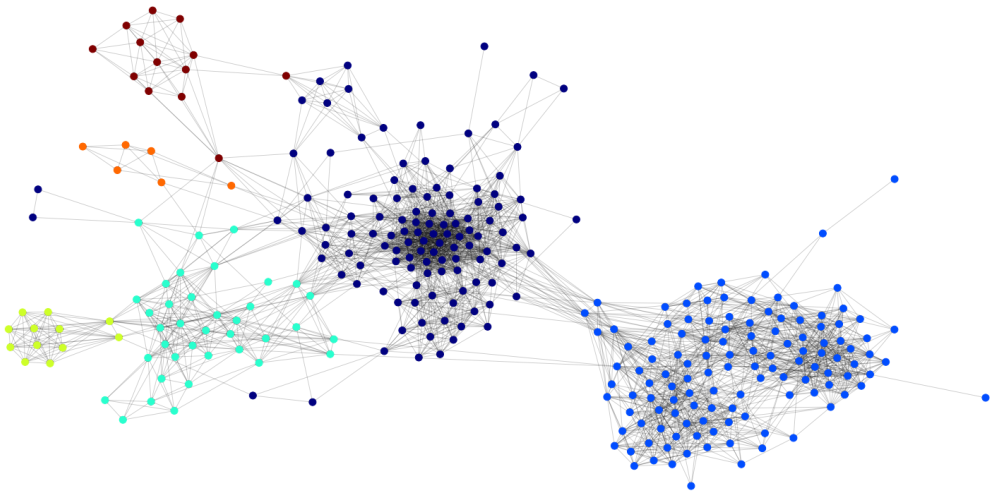
[Implement] the function `spect_cluster` that returns a vector `y_clust` in which each entry y_clust[i] represents the community assigned to node i. The method should be able to handle both the Normalized Laplacian, and the Random Walk Laplacian. You are allowed to use your implementation from the weekly exercises and `sklearn.cluster.k_means` for k-means clustering.

In [ ]:

```
from sklearn.cluster import k_means

def spect_cluster(G, eig_type="normal", k=5, d=5):
    ### YOUR CODE STARTS HERE

    nodes = G.nodes
    NumNodes = len(nodes)


    # Create adjacency matrix
    A = np.zeros((len(nodes), len(nodes)))
    for index in G.edges:
        A[index[0],index[1]] = 1
        A[index[1],index[0]] = 1

    # Create degree matrix
    D = np.zeros((NumNodes, NumNodes))
    for i in range(NumNodes):
        D[i,i] = np.sum(A[i,:])

    identity_matrix = np.identity(len(nodes))

    if eig_type == "normal":
        L = identity_matrix - np.linalg.inv(D)**(1/2) @ A @ np.linalg.inv(D)**(1/2)

    if eig_type == "random":
        L = identity_matrix - np.linalg.inv(D) @ A



    l, e = graph_eig(L)
    order = np.argsort(l)
    l = l[order]
    e = e[:, order]
```

```
    sel = np.arange(len(l))[np.where(l > 1e-15)][:d]

    X_ = e[:, sel]

    best_interia = float('inf')
    y_clust = None
    for i in range(10):
        _, y_, interia, *_ = k_means(X_, k)
        if interia < best_interia:
            y_clust = y_
            best_interia = interia


    ### YOUR CODE ENDS HERE
    return y_clust

#print(spect_cluster(G, eig_type="normal", k=5, d=5))
```

In [ ]:
```
def plot_graph(G, clusters):
    plt.figure(1,figsize=(30,15))
    nodes = G.nodes()
    ec = nx.draw_networkx_edges(G, X, alpha=0.2)
    nc = nx.draw_networkx_nodes(G, X, nodelist=nodes, node_color=clusters, node_siz

    plt.axis('off')
    plt.show()

your_clusters = spect_cluster(G, k=6)
plot_graph(G, your_clusters)
```



Task 2.2.4 (1 points)

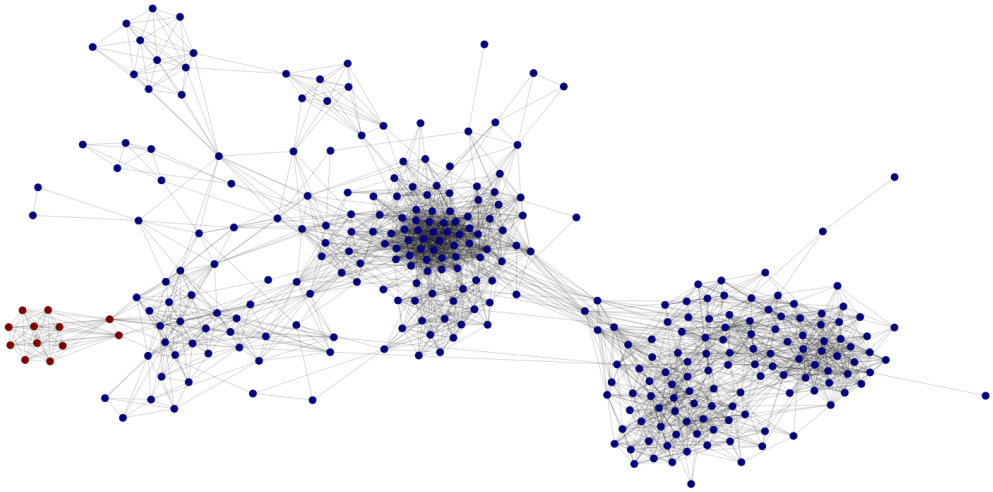
Finally, use your implementation of spectral clustering with different Laplacians and different values of $k \in [2, 7]$ and plot the results using the helper function `plot_graph`.

[Describe] the results you obtain. Especially, what is the difference between the Random Walk and the Normalized Laplacians, if any? How do you explain such differences?

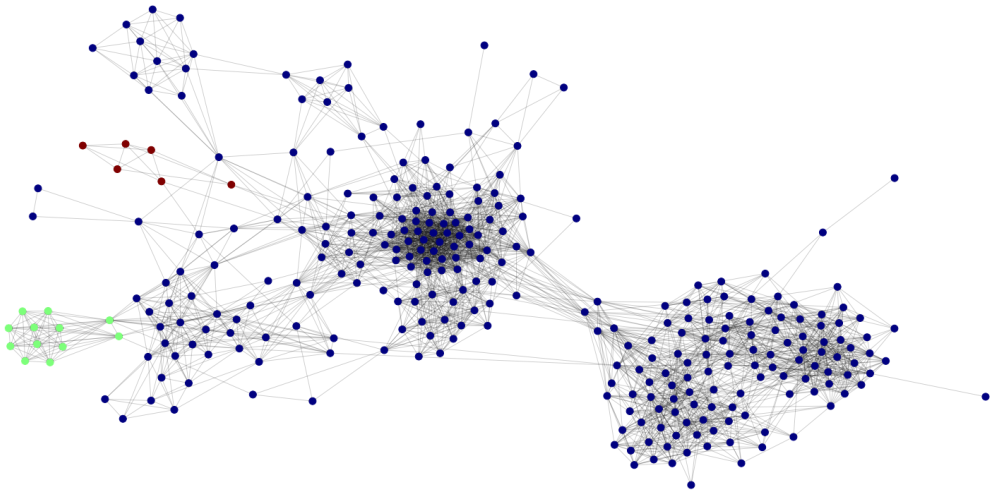
In [ ]:
```
for method in ['normal', 'random']:
    for k in np.arange(2,8):
        print("Method: {}, k: {}".format(method, k))
```

```
        your_clusters = spect_cluster(G,eig_type=method, k=k)
        plot_graph(G, your_clusters)
```

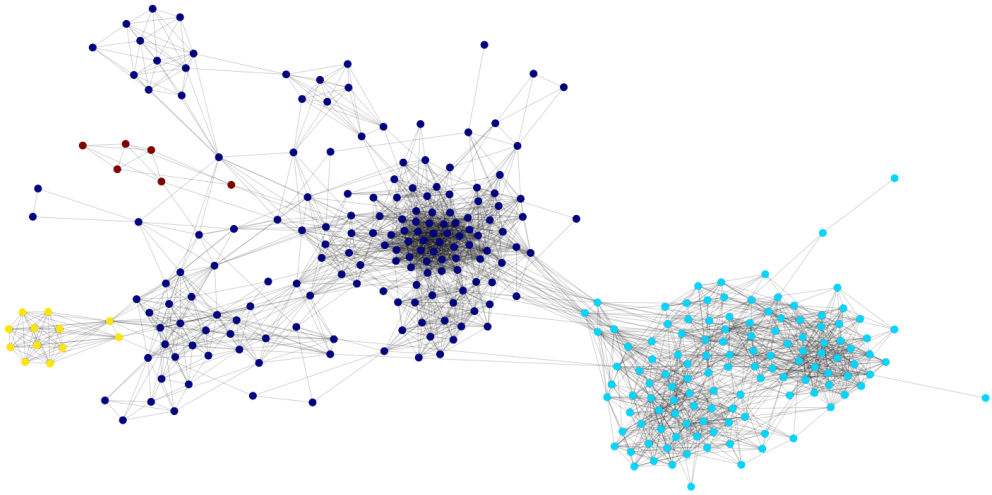
```
Method: normal, k: 2
```



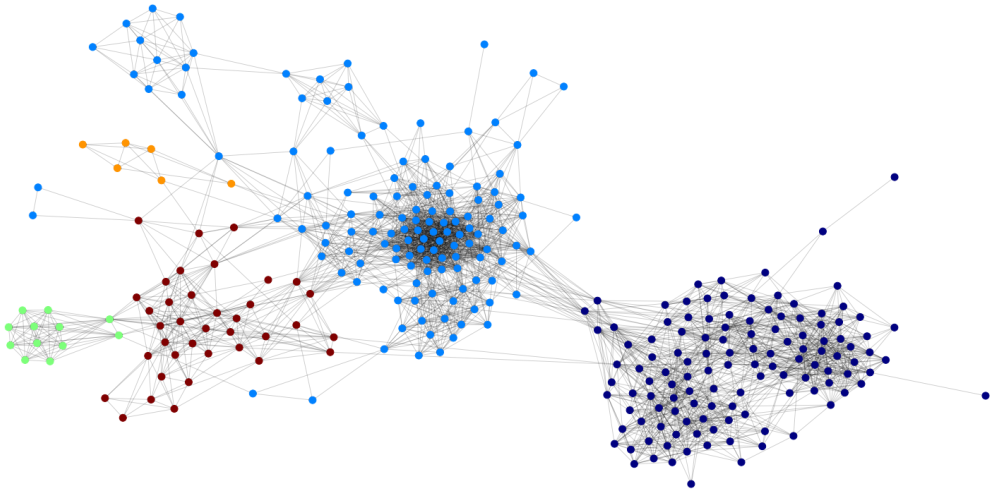
```
Method: normal, k: 3
```



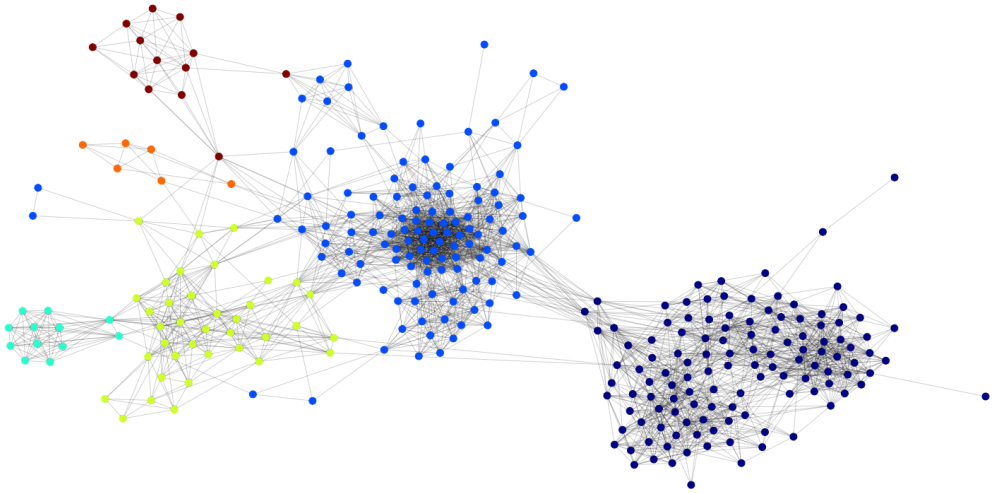
```
Method: normal, k: 4
```



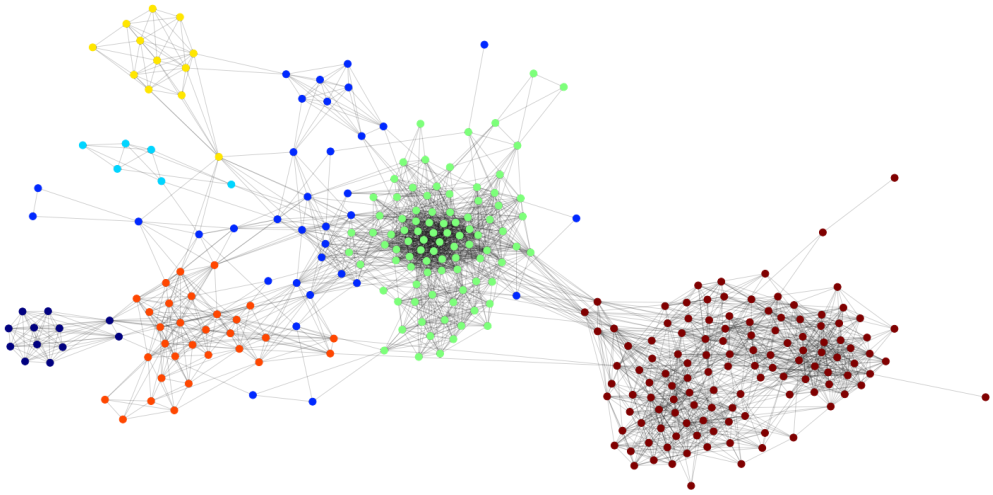
```
Method: normal, k: 5
```



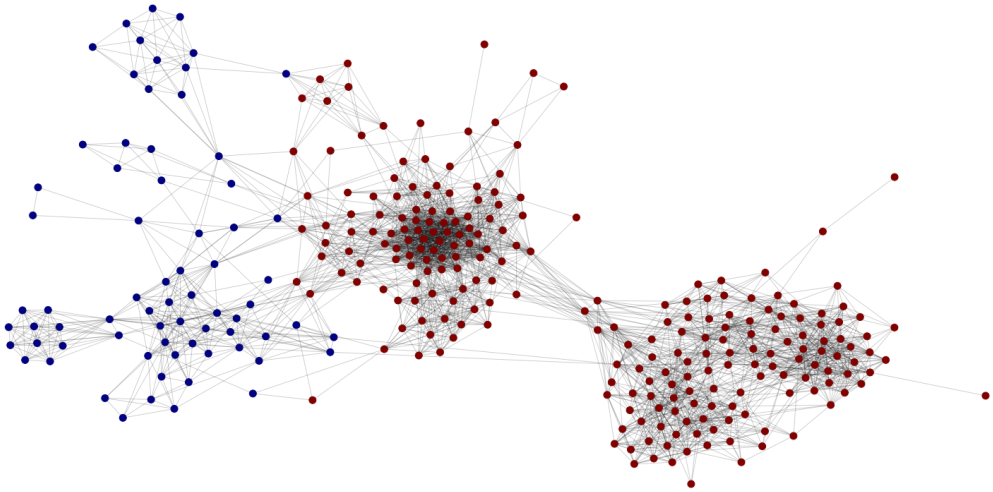
Method: normal, k: 6



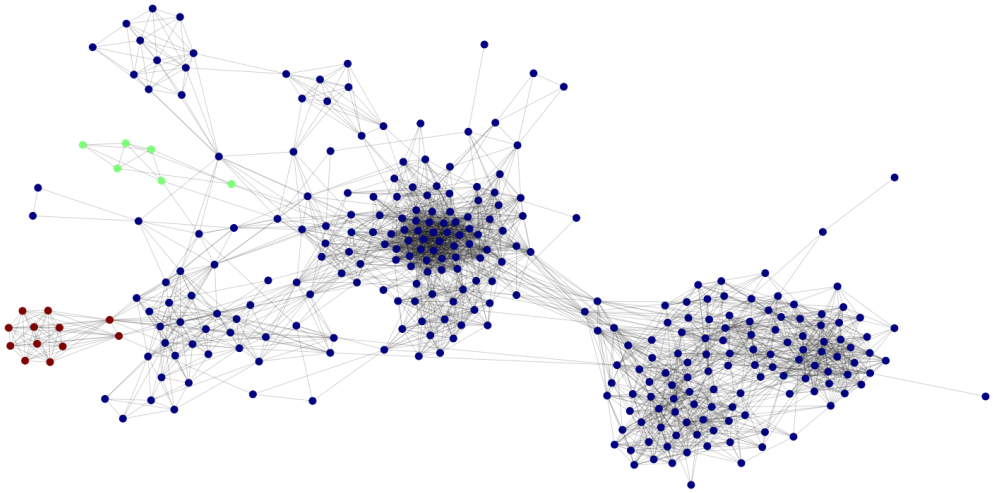
Method: normal, k: 7



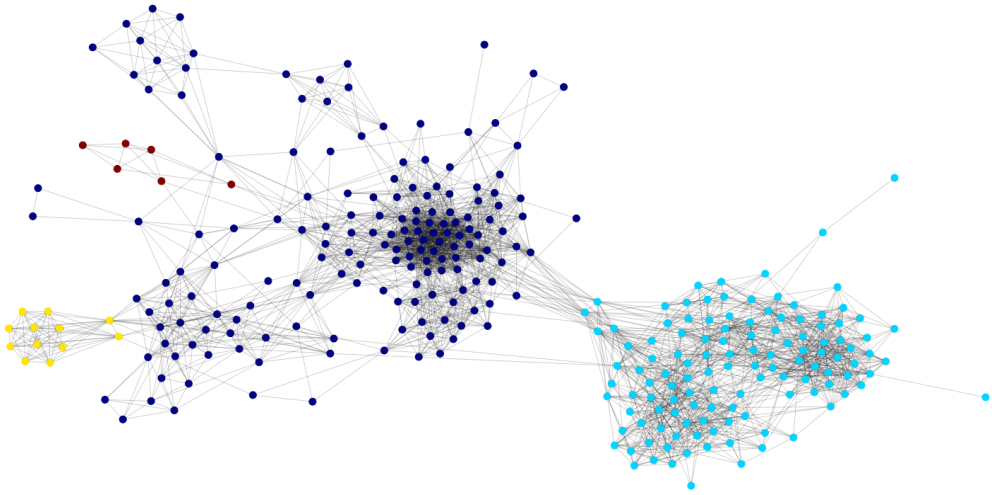
Method: random, k: 2



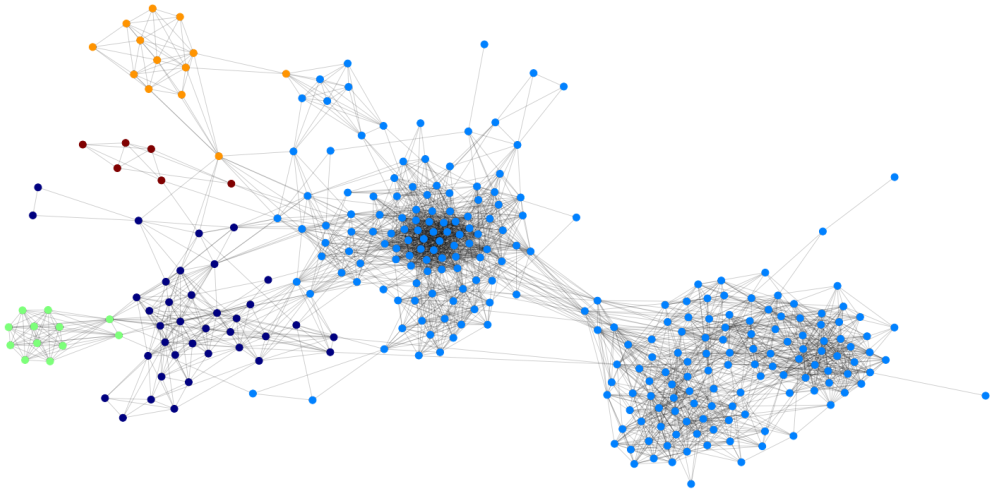
Method: random, k: 3



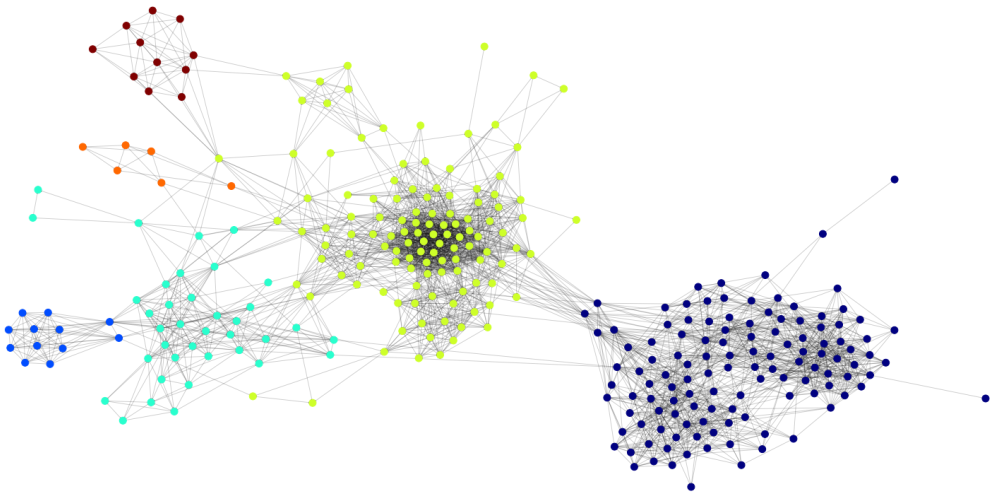
Method: random, k: 4



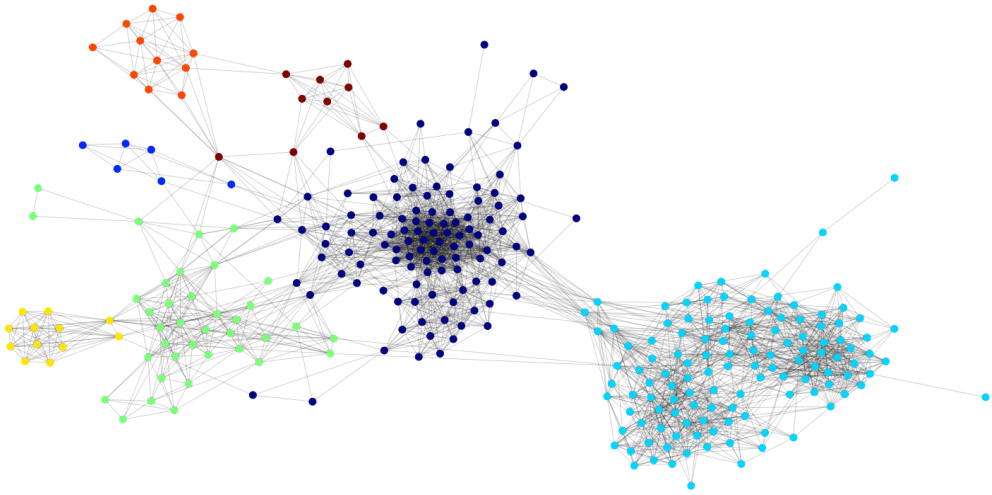
Method: random, k: 5



Method: random, k: 6



Method: random, k: 7



YOUR ANSWER HERE

From the plots we can see that the random walk performs more even clustering e.g. k=5 and k=7 the normalized Laplacians seems to have few nodes lie far from the cluster. This

may show that , the choice of Laplacian can affect the clustering results. The Random Walk Laplacian tends to emphasize clusters that have dense connections within themselves and sparse connections to other clusters. In contrast, the Normalized Laplacian tends to balance the effect of the degree of the nodes in the graph, making it less sensitive to the density of the clusters. Moreover, as we do not classify any nodes as outliers, there will be some cluster that ends up have some nodes that look like outliers. Which could be why we see some clusters overlapping more in the normalized Laplacian.

Task 2.2.5 (4 points)

[Implement] the modularity. Recall that the definition of modularity for a set of communities C is

$$Q = \frac{1}{2m} \sum_{c \in C} \sum_{i \in c} \sum_{j \in c} \left(A_{ij} - \frac{d_i d_j}{2m} \right) \qquad (1)$$

where A is the adjacency matrix, and d_i is the degree of node i

Note: Use `plot_graph` function in order to see for yourself if maximising modularity leads a better clustering. If you did not succeed with the previous Task, you are allowed to use Scikit Learn Spectral Clustering

In [ ]:
```
def modularity(G, clustering):
    modularity = 0
    ### YOUR CODE STARTS HERE
    n = len(G.nodes)
    m = len(G.edges)

    nodes = G.nodes
    NumNodes = len(nodes)

    # Create adjacency matrix
    A = np.zeros((n, n))
    for index in G.edges:
        A[index[0],index[1]] = 1
        A[index[1],index[0]] = 1

    # Create degree matrix
    D = np.zeros((NumNodes, NumNodes))
    for i in range(NumNodes):
        D[i,i] = np.sum(A[i,:])



    clusters = set(clustering)

    Q = []
    for cluster in clusters:
        nodes = np.where(clustering == cluster)[0]
        Q_i = []
        for i in nodes:
            Q_j = []
            for j in nodes:
                if j != i:
```

```
                    Q_j.append(A[i][j] - ((D[i][i]*D[j][j])/(2*m)))
            Q_i.append(sum(Q_j))

        Q.append(sum(Q_i))



    modularity = sum(Q)/(2*m)
    ### YOUR CODE ENDS HERE
    return modularity

clusters = spect_cluster(G, k=10) ### NOTE: If you do not use your implementation s
print(modularity(G, clusters))
```

```
0.5684828559374677
```

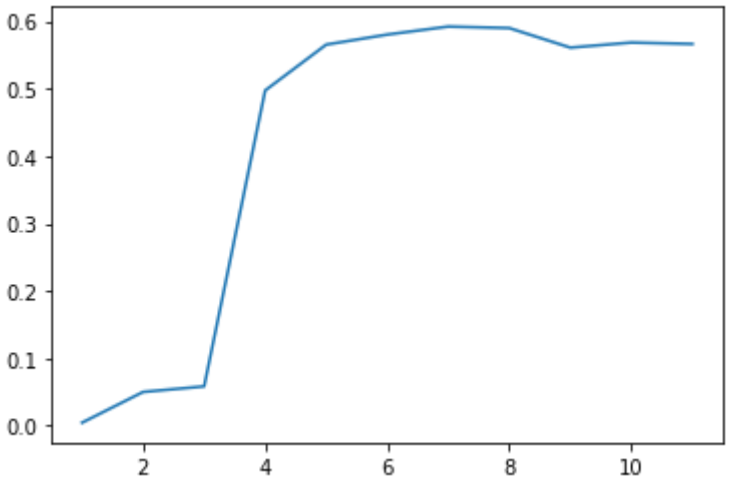
Task 2.2.6 (2 points)

Compute the modularity of your Spectral Clustering Implementation for different values of k .

[Motivate] Which k value maximizes the modularity? From your perspective, does spectral clustering forms "clear" clusters for the best k found by modularity? Using the spectral graph theory, why do you think this is/isn't the case?

In [ ]:
```
mods = []
ks = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
for k in ks:
    clusters = spect_cluster(G, k=k) ### NOTE: If you do not use your implementatio
    mods.append(modularity(G, clusters))

# You may want to use plt.plot to plot the modularity for different values of k
plt.plot(ks, mods)

print(ks[int(np.where(mods == np.max(mods))[0])])
print(mods[int(np.where(mods == np.max(mods))[0])])
```

```
c:\Users\Bjarki Kass Olsen\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:
881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=2.
  warnings.warn(
c:\Users\Bjarki Kass Olsen\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:
881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=2.
  warnings.warn(
c:\Users\Bjarki Kass Olsen\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:
881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=2.
  warnings.warn(
c:\Users\Bjarki Kass Olsen\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:
881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=2.
  warnings.warn(
c:\Users\Bjarki Kass Olsen\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:
881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=2.
  warnings.warn(
c:\Users\Bjarki Kass Olsen\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:
881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=2.
  warnings.warn(
c:\Users\Bjarki Kass Olsen\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:
881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=2.
  warnings.warn(
c:\Users\Bjarki Kass Olsen\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:
881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=2.
  warnings.warn(
c:\Users\Bjarki Kass Olsen\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:
881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=2.
  warnings.warn(
c:\Users\Bjarki Kass Olsen\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:
881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=2.
  warnings.warn(
7
0.5922033365438507
```



YOUR ANSWER HERE In the graph we see that the modularity is highest when k is around 7.
From the spectral clustering we get a modularity of 0.592 when $k = 7$. The modularity tells us how good the clustering is and the higher the modularity the more clear the clustering is, which means that the nodes in the clusters are more connected to eachother than to the nodes in other clusters. Given a score of 0.592 we can say that the clustering is good since the highest modularity score is 1, which is very difficult to attain while lowest score is -1.

Task 2.2.7 (2 points)

[Motivate] There seems to be a relationship between graph embeddings and spectral clustering, can you guess that? *Hint*: Think to the eigenvectors of the graph's Laplacians. (1) Check the correct box below and (2) motivate your answer.

- [x] If the embeddings are linear and the similarity is the Laplacian, the embeddings we obtain minimizing the L_2 norm are equivalent to the eigenvectors of the Laplacian.
- [ ] If the embeddings are random-walk-based embeddings, the eigenvectors of the Random Walk Laplacian are related to the embeddings obtained by such methods.
- [ ] The relationship is just apparent.
- [ ] If the embeddings are linear and the similarity is the Adjacency matrix, the eigenvectors of the Laplacian are equivalent to the embeddings.

IMPORTANT: Do NOT just choose one answer. Please clarify WHY this is the correct answer.

YOUR ANSWER HERE

If the embeddings are linear and the similarity is the Laplacian, the embeddings we obtain minimizing the L_2 norm are equivalent to the eigenvectors of the Laplacian. This is because the Laplacian matrix is a real, symmetric matrix, and its eigenvectors form an orthonormal basis for the space of real-valued functions on the graph. Therefore, any linear combination of the eigenvectors of the Laplacian can be used to embed the graph.

Part 3: Link analysis

In this exercise, we will work with PageRank, Random Walks and their relationships with graph properties. We will use the most generic definition

$$\mathbf{r} = \alpha \mathbf{M} \mathbf{r} + (1 - \alpha)\mathbf{p}$$

with $\mathbf{r}$ the PageRank vector, $\mathbf{M}$ the weighted transition matrix, and $\mathbf{p}$ the personalization vector. Additionally, let $n = |V|$, where V is the nodes in the graph above. Remember that in the case of PageRank the entries of the personalization vector are $p_i = 1/n$ for all i.

Task 3.1 Approximate PageRank (10 points)

Task 3.1.1 (3 points)

[Implement] a different algorithm for computing Personalized PageRank. This algorithm runs a fixed number of iterations and uses the definition of random walks. At each step, the algorithm either selects a random neighbor with probability α or returns to the starting node with probability $1 - \alpha$. Every time a node is visited a counter on the node is incremented by one. Initially, each counter is 0. The final ppr value is the values in the nodes divided by the number of iterations.

In [ ]:
```
import random as rd

def approx_personalized_pagerank(G, node, alpha = 0.85, iterations = 1000):


    # Get number of nodes
    n = G.number_of_nodes()

    # Initialize the pagerank vector
    ppr = np.zeros(n)

    # Set current node to starting node
    curr_node = node

    # Update visited nodes
    ppr[curr_node] += 1

    # Start the Iterations
    for i in range(1,iterations):

        # Get the probability of teleporting
        if rd.random() > alpha:
            # Here we do teleport and so we return to the starting node and increas
            curr_node = node
            ppr[curr_node] += 1
        else:
            # Here we do not teleport and so we move to one of its neighbors at ran
            # We also increase the rank of that node by 1

            # Get the neighbors of the current node
            neighbors = list(G.neighbors(curr_node))
```

```
            # See if there are any neighbors to move to, else stay
            if len(neighbors) == 0:
                neighbors = [curr_node] # Stay

            # Choose a random neighbor
            curr_node = rd.choice(neighbors)

            ppr[curr_node] += 1

    ### YOUR CODE ENDS HERE
    return ppr/iterations
```

Task 3.1.2 (3 points)

Run the `approx_personalized_pagerank` with default α and iterations $\{10, n, 2n, 4n, 100n, 1000n\}$ where n is the number of nodes in the graph and starting node the node with the highest PageRank computed in Task 3.1.2.

[Motivate] what you notice as the number of iterations increase. Why are the values and the top-10 nodes ranked by PPR changing so much?

In [ ]:
```
edgelist = read_edge_list('./data/edges.txt')
n = np.max(edgelist)+1
G = nx.Graph()
for i in range(n):
    G.add_node(i)
for edge in edgelist:
    G.add_edge(edge[0], edge[1])
starting_node = np.argmax(nx.pagerank(G))
for i, iterations in enumerate([10, G.number_of_nodes(), G.number_of_nodes()*2, G.r
    r = approx_personalized_pagerank(G, starting_node, iterations = iterations)
    r[starting_node] = 0
    r_sorted = np.argsort(r)[::-1]
    r_values = np.sort(r)[::-1]
    print(f'Iteration {iterations}: top-10 r={r_sorted[:10]}\n top-10 values={r_val

import operator
rr = nx.pagerank(G, alpha=0.85, personalization = {starting_node: 1})
rr[starting_node] = 0
r=np.zeros(len(rr))
for k in rr: r[k] = rr[k]
r_sorted = np.argsort(r)[::-1]
r_values = np.sort(r)[::-1]
print(f'top-10 r={r_sorted[:10]}\n top-10 values={r_values[:10]}\n')
```

```
Iteration 10: top-10 r=[ 36  19  40  41  42  39  85 110 104 105]
 top-10 values=[0.2 0.1 0.1 0.1 0.1 0.1 0.  0.  0.  0. ]

Iteration 307: top-10 r=[41 15 18 42 40 39 57 17 36 44]
 top-10 values=[0.08469055 0.0781759  0.07166124 0.07166124 0.07166124 0.06514658
 0.06188925 0.05537459 0.04234528 0.04234528]

Iteration 614: top-10 r=[40 41 18 15 39 42 44 17 36 37]
 top-10 values=[0.09609121 0.0732899  0.0732899  0.06677524 0.06677524 0.06351792
 0.04723127 0.04560261 0.04234528 0.03908795]

Iteration 1228: top-10 r=[39 41 40 15 18 42 44 36 57 17]
 top-10 values=[0.08387622 0.0732899  0.06840391 0.06514658 0.06351792 0.05374593
 0.05374593 0.05374593 0.04641694 0.02931596]

Iteration 30700: top-10 r=[39 41 40 18 42 15 36 44 57 37]
 top-10 values=[0.07110749 0.06820847 0.06661238 0.06224756 0.05856678 0.05348534
 0.05188925 0.0457329  0.04188925 0.03351792]

Iteration 307000: top-10 r=[39 41 40 18 15 42 36 44 57 17]
 top-10 values=[0.07335505 0.06821498 0.06814007 0.06495765 0.05771336 0.05742671
 0.05115635 0.04411726 0.04348208 0.03356352]

top-10 r=[39 41 40 18 15 42 36 44 57 37]
 top-10 values=[0.07370303 0.0681049  0.0681049  0.0641951  0.05796678 0.05772109
 0.05078277 0.04421339 0.04303241 0.03313206]
```

YOUR ANSWER HERE

Because, our algorithm is based on what nodes we have visited, then if we only have 10 steps we will only have visited a few nodes, and if we have 1000n steps we will have visited a lot of nodes. This means that the nodes that we have visited will have a higher value in the nodes divided by the number of iterations. Therefor, we see that some of the pagerank values in the 10 iterations are 0. And further, with our algorithm, we are converging to the real value of the pagerank, based on the theory of Law of Large numbers, where the higher iterations we have, the closer we will get to the true distribution. Lastly, as we travel from one note to another (we don't use the whole matrix at once), and we have a probability of teleporting, then given a large graph G, we will have a probability of some of the nodes not being visited, and therefore having a pagerank value of 0.

Task 3.1.3 (2 points)

Compare the 5 nodes with the highest PPR obtained from `nx.pagerank(G, alpha=0.85, personalization={node_highest_pagerank: 1})` and the one obtained by the approximation.

[Describe] the differences. Does the number of iterations affect the results? Is there a relationship between the number of iterations and the results? Is there a relationship between the approximated value of PageRank and the real value? Do you notice anything as the number of iteration increases?

In [ ]:

```
k = 5
ppr_nx = nx.pagerank(G, alpha=0.85, personalization = {starting_node: 1})
r_nx = [0 for _ in range(G.number_of_nodes())]
for k, v in ppr_nx.items():
    r_nx[k] = v
r_est = approx_personalized_pagerank(G, starting_node, alpha=0.85)

topk_nx = np.argsort(r_nx)[-5:]
topk_est = np.argsort(r_est)[-5:]

print(topk_nx, topk_est)

for iterations in [10, G.number_of_nodes(), G.number_of_nodes()*2, G.number_of_node
    print(f'Number of iterations {iterations}')
    ppr_nx = nx.pagerank(G, alpha=0.85, personalization = {starting_node: 1})
    r_nx = [0 for _ in range(G.number_of_nodes())]
    for k, v in ppr_nx.items():
        r_nx[k] = v
    r_est = approx_personalized_pagerank(G, starting_node, iterations = iterations,
    print(f'Approximate PPR: {r_est[:10]}')
    print(f'Real PPR: {r_nx[:10]}')


    topk_nx = np.argsort(r_nx)[-5:]
    topk_est = np.argsort(r_est)[-5:]

    print(f"Topk of nx.pagerank: {topk_nx}, Topk of our estimation {topk_est}, Size
```

```
[18 41 40 39  0] [18 40 39 41  0]
Number of iterations 10
Approximate PPR: [0.4 0.  0.  0.  0.  0.  0.  0.  0.  0. ]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005
17928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629
9, 0.000586579792176299, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [18 57 37 15  0], Si
ze of intersection: 2
Number of iterations 307
Approximate PPR: [0.17263844 0.         0.         0.01628664 0.00325733 0.
 0.00325733 0.         0.         0.         ]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005
17928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629
9, 0.000586579792176299, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [18 39 40 41  0], Si
ze of intersection: 5
Number of iterations 614
Approximate PPR: [0.2019544  0.         0.         0.00488599 0.         0.
 0.         0.00162866 0.         0.         ]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005
17928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629
9, 0.000586579792176299, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [42 41 40 39  0], Si
ze of intersection: 4
Number of iterations 1228
Approximate PPR: [0.19462541 0.         0.00162866 0.002443   0.00325733 0.002443
 0.00081433 0.00081433 0.         0.00081433]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005
17928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629
9, 0.000586579792176299, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [39 18 40 41  0], Si
ze of intersection: 5
Number of iterations 30700
Approximate PPR: [0.2047557  0.00058632 0.00123779 0.00537459 0.00091205 0.0005211
7
 0.00071661 0.00065147 0.00039088 0.00032573]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005
17928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629
9, 0.000586579792176299, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [18 40 41 39  0], Si
ze of intersection: 5
Number of iterations 307000
Approximate PPR: [0.20090228 0.00041368 0.00146906 0.00522476 0.00108143 0.0006384
4
 0.00061889 0.00064169 0.0002899  0.00020521]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005
17928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629
9, 0.000586579792176299, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [18 40 41 39  0], Si
ze of intersection: 5
```

YOUR ANSWER HERE

As mentioned above, based on the law of large number, as the iterations increase towards infinity, the approximated pagerank will converge to the true pagerank of the graph G.

Task 3.1.4 (2 points)

Run again the same experiment but this time use $\alpha = 0.1$.

[Motivate] Motivate whether and why you need more or less iterations to predict the 5 nodes with the highest PPR.

In [ ]:
```
for iterations in [10, G.number_of_nodes(), G.number_of_nodes()*2, G.number_of_node
    ppr_nx = nx.pagerank(G, alpha=0.1, personalization = {starting_node: 1})
    r_nx = [0 for _ in range(G.number_of_nodes())]
    for k, v in ppr_nx.items():
        r_nx[k] = v
    r_est = approx_personalized_pagerank(G, starting_node, iterations = iterations,

    topk_nx = np.argsort(r_nx)[-5:]
    topk_est = np.argsort(r_est)[-5:]

    print(f"Topk of nx.pagerank: {topk_nx}, Topk of our estimation {topk_est}, Size
```

```
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [ 99  98  97 104
0], Size of intersection: 1
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [39 42 41 57  0], Si
ze of intersection: 4
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [18 15 41 42  0], Si
ze of intersection: 3
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [57 15 41 40  0], Si
ze of intersection: 3
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [18 41 39 42  0], Si
ze of intersection: 4
```

YOUR ANSWER HERE

This is because, with $\alpha = 0.1$ we have a VERY high probability of teleporting back to the starting node, and therefore, our visits throughout the graph will be very limited. Therefore, we will need considereably many more iterations to converge to the true pagerank.

Task 3.2 Spam and link farms (12 points)

We will now study the effect of spam in the network and construct a link farm. In this part, if you want to modify the graph, use a copy of the original graph every time you run your code, so that you do not keep adding modifications.

In [ ]:
```
edgelist = read_edge_list('./data/edges.txt')
n = np.max(edgelist)+1
G2 = nx.Graph()
for i in range(n):
    G2.add_node(i)
for edge in edgelist:
    G2.add_edge(edge[0], edge[1])
G = G2.copy()
```

Task 3.2.1 (3 points)

Based on the analysis in the slides, construct a spam farm s on the graph G with T fake nodes. Assume that s manages to get links from node 1. With $\alpha = 0.5$,

[Describe] which is the minimum number of pages T that we you need to add in order to get s being assigned the highest PageRank?

In [ ]:
```
#CODE HERE IF YOU NEED THAT

def spamfarm(G, alpha):
    starting_node = np.argmax(nx.pagerank(G))
    n = np.max(edgelist)+1
    ppr = nx.pagerank(G, alpha=0.5, personalization = {starting_node: (starting_noc

    pX = ppr[1]
    max_ppr = max(ppr.values())

    T=0
    ps = pX/(1-alpha**2)

    while ps < max_ppr:
        T += 1
        ps = pX/(1-alpha**2) + alpha/(1+alpha)*T/n


    return f"Number of fake nodes added {T}, with pagerank of {ps}"


print(spamfarm(G, 0.5))
```

```
Number of fake nodes added 483, with pagerank of 0.524448936087188
```

YOUR ANSWER HERE We use the formula below to calculate the number of pages T needed to get the highest PageRank. Let be p_s the PageRank of the page s and p_X the PageRank of the "honest" pages that link to s. The PageRank p_s is the sum of the contribution of the honest pages p_X and the fake pages

$$p_s = \frac{p_X}{1-\alpha^2} + \frac{\alpha}{1+\alpha}\frac{T}{n}$$

From our calculations we have to set T=483 to get the highest PageRank for the spam farm.

Task 3.2.2 (3 points)

In the above scenario, assume that $T = \frac{1}{5}$ of the nodes in the original graph.

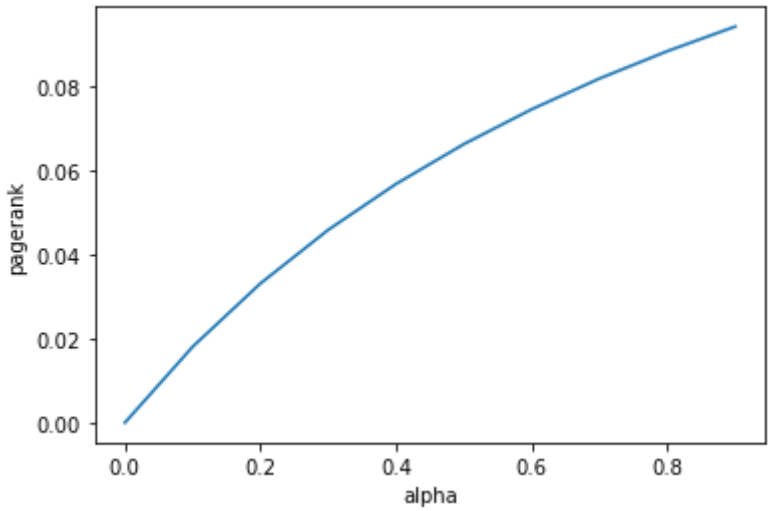
[Motivate] what value of α will maximize the PageRank $\mathbf{r}_s$ of the link farm s. Provide sufficient justification for your choice.

YOUR ANSWER HERE

From the plot below we notice that the pagerank of the linkfarm increases as α goes towards 1. This make sense given the formula of the pagerank for the linkfarm. When alpha

is 0 the pagerank of the linkfarm depends only on thhe pagerank of the honest pages that links to the linkfarm. When alpha increases the pagerank of the linkfarm depends more on the number of fake pages and fraction of the contribution from the honest pages. When alpha is 1 the pagerank of the linkfarm becomes $\frac{1}{2}\frac{T}{n}$.

In [ ]:
```
starting_node = np.argmax(nx.pagerank(G))
n = np.max(edgelist)+1
ppr = nx.pagerank(G, alpha=0.5, personalization = {starting_node: (starting_node +

pX = ppr[1]
T=int(n/5)


pagerank_val = []

for alpha in np.arange(0,1,0.1):
    ps = pX/(1-alpha**2) + alpha/(1+alpha)*T/n
    pagerank_val.append(ps)


plt.plot(np.arange(0,1,0.1), pagerank_val)
plt.xlabel('alpha')
plt.ylabel('pagerank')
plt.show()
```



Task 3.2.3 (3 points)

Now we fix both $\alpha = 0.5$ and $T = \frac{1}{5}n$.

[Implement] `trusted_pagerank` the method for spam mass estimation.

In [ ]:
```
def trusted_pagerank(G, trusted_indices, iterations=500, alpha=0.5):
    r = None
    # Get number of nodes
    n = G.number_of_nodes()

    # Initialize the pagerank vector
    tr = np.zeros(n)

    #Define the starting node as the node with the highest pagerank
    node = np.argmax(nx.pagerank(G, alpha=alpha, personalization={starting_node:1})
```

```
    # Set current node to starting node
    curr_node = node

    # Update visited nodes
    tr[curr_node] += 1

    # Start the Iterations
    for _ in range(1,iterations):

        # Get the probability of teleporting
        if rd.random() > alpha:
            # Here we do teleport and so we return to the starting node and increas
            curr_node = rd.choice(trusted_indices)
            tr[curr_node] += 1
        else:
            # Here we do not teleport and so we move to one of its neighbors at ran
            # We also increase the rank of that node by 1

            # Get the neighbors of the current node
            neighbors = list(G.neighbors(curr_node))

            # See if there are any neighbors to move to, else stay
            if len(neighbors) == 0:
                neighbors = [curr_node] # Stay

            # Choose a random neighbor
            curr_node = rd.choice(neighbors)

            tr[curr_node] += 1

    page_ranks = [i for i in nx.pagerank(G, alpha=alpha, personalization={starting_
    trust_ranks = tr/iterations
    #person_ppr = approx_personalized_pagerank(G, node, iterations=iterations, alph


    r = (page_ranks - trust_ranks)/page_ranks


    return r


print(trusted_pagerank(G, [1,2,3,4,5], iterations=500, alpha=0.5))
```

```
[ 9.96184436e-01 -8.71513120e+03 -1.37763199e+03 -5.90470261e+01
 -1.29047242e+03 -4.99308929e+03 -2.18863848e+02 -1.45575899e+02
  1.00000000e+00  1.00000000e+00 -1.44833255e+02 -2.27244818e+03
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 -2.21231594e+01  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00 -1.53718028e+03 -1.53718028e+03
 -1.93688468e+02 -2.83561719e+02 -2.68728372e+02 -1.18310717e+01
 -1.89193045e+02  1.00000000e+00  1.00000000e+00 -8.81068052e+01
 -7.89493235e+01  1.00000000e+00  1.00000000e+00  1.00000000e+00
  9.06981721e-01  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00 -1.13572409e+03  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 -2.22433969e+01  1.00000000e+00 -2.26712437e+02 -2.83348567e+02
 -9.14926376e+01 -2.57121795e+02 -1.12366209e+03 -5.98145794e+02
 -6.93071937e+02 -5.93758121e+02 -3.85033331e+02 -2.89476708e+02
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00 -1.24066109e+01  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00 -4.19226577e-02  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00 -2.44631120e+02 -8.30131277e-02
 -7.79004531e-02  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00 -1.05818225e-01  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 -1.34398702e+01  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
  1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
```

```
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00 -1.46625764e-01 -6.11674965e+00 -4.85707490e+00
-1.30669850e+01 -1.30669850e+01  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000000e+00
 1.00000000e+00  1.00000000e+00  1.00000000e+00]
```

Task 3.2.4 (3 points)

[Motivate] whether we are able to detect the node s, if the trusted set of nodes is a random sample 10% of the nodes in the original graph. If not, what could be a viable solution? Which nodes would you rather choose as trusted?

You are not obliged to, but you can write some helper code to reach the answer faster.

Hint: Remember the spam mass formula in the Link Analysis lecture

In [ ]:
```
### YOUR CODE HERE
```

YOUR ANSWER HERE

As choosing 10% of the nodes at random, is the same as choosing 10% of the nodes, one of the time with the probability of $1/n$, until they make 10% of the nodes. Therefore, when finding the pagerank with teleporting to these random trusted pages, will be almost the same as the pagerank when teleporting to all the nodes at equal probability, as they were chosen at random with equal probability. So the two processes or almost the same. Therefore, we will have that:

$$r_x \approx r_x^+ \implies r_x^- \approx \bar{0} \implies x \approx 0,$$

Meaning that our spam mass will be almost zero, and therefore, we will probably not be able to detect the node s.

Instead, a better way of choosing the trusted nodes, is to use the intuition of choosing nodes that are far away from the linkfarm. Therefore, one way would be to choose the nodes with the highest pagerank, as they are the most trusted nodes in the network. This way, we will have a higher chance of detecting the node s. Or we could choose nodes with trusted domains, e.g. au.dk etc., as they are not likely to be linking to a spam page in the first place.

Part 4: Graph embeddings (19 points)

In this final part, we will try a different approach for clustering the data from above. The strategy is going to be the following:

1. Use VERSE [1] to produce embeddings of the nodes in the graph.
2. Use K-Means to cluster the embeddings. Measure and report NMI for the clustering.

[1] Tsitsulin, A., Mottin, D., Karras, P. and Müller, E., 2018, April. Verse: Versatile graph embeddings from similarity measures. In Proceedings of the 2018 World Wide Web Conference (pp. 539-548).

In [ ]:
```
G = email.S_dir.copy()
```

Task 4.1.1 (6 points)

[Implement] the methods below to compute sampling version of VERSE. *Hint:* it might be a help to look in the original article [1] above.

In [ ]:
```
def sigmoid(x):
    ''' Return the sigmoid function of x
        x: the input vector
    '''
    ### YOUR CODE STARTS HERE

    x = 1 / ( 1 + np.exp(-x))

    ### YOUR CODE ENDS HERE
    return x

def pagerank_matrix(G, alpha = 0.85) :
    ''' Return the Personalized PageRank matrix of a graph

        Args:
            G: the input graph
            alpha: the damping factor of PageRank

        :return The nxn PageRank matrix P
    '''
    ### YOUR CODE STARTS HERE

    # Get number of nodes
    n = G.number_of_nodes()

    # Initialize the pagerank matrix
    P = np.zeros((n,n))

    # Get the Personalized PageRank vector for each node
    for node in range(n):

        P[:,node] = approx_personalized_pagerank(G, node, iterations=5000, alpha=al

    ### YOUR CODE ENDS HERE
    return P
```

```python
def update(u, v, Z, C, step_size) :
    '''Update the matrix Z using row-wise gradients of the loss function

       Args:
            u : the first node
            v : the second node
            Z : the embedding matrix
            C : the classification variable used in Noise Contrastive estimation ir
            step_size: step size for gradient descent


       :return nothing, just update rows Z[v,:] and and Z[u,:]

       ****(if we don't return anything, then we need to use the keyword global to
       Therefore, it is much simpler to just return the updated Z)****
    '''
    ### YOUR CODE STARTS HERE

    # Get the sigmoid of the dot product of the two nodes
    sigma = sigmoid(Z[u].dot(Z[v]))

    # Compute the gradient
    grad = (C - sigma) * step_size

    # Update the embedding matrix
    Z[u] += grad * Z[v]
    Z[v] += grad * Z[u]

    ### YOUR CODE ENDS HERE

    return Z


def verse(G, S, d, k = 3, step_size = 0.0025, steps = 10000):
    ''' Return the sampled version of VERSE

        Args:
            G: the input Graph
            S: the PageRank similarity matrix
            d: dimension of the embedding space
            k: number of negative samples
            step_size: step size for gradient descent
            steps: number of iterations

        :return the embedding matrix nxd
    '''
    n = G.number_of_nodes()
    Z = 1/d*np.random.rand(n,d)

    ### YOUR CODE STARTS HERE

    # Start the recursive update
    for _ in range(steps):

        # Pick the positive sample
        u = np.random.choice(n)

        # Pick the similar node
        neighbors = list(G.neighbors(u))
```

```
        v_pos = np.random.choice(neighbors, p = (S[neighbors,u]/np.sum(S[neighbors,

        # Pick the k negative samples
        not_neighbors = list(set(G.nodes()) - set(neighbors))
        v_neg = np.random.choice(not_neighbors, size=k, replace=False, p = (S[not_n

        # Update the embedding matrix with the positive sample
        Z = update(u, v_pos, Z, C = 1, step_size = step_size)

        # Update the embedding matrix with all the negative samples
        for v_hat in v_neg:
            Z = update(u, v_hat, Z, C = 0, step_size = step_size)

    ### YOUR CODE ENDS HERE
    return Z
```

In [ ]:
```
# This code runs the `verse` algorithm above on G and stores the embeddings to 'ver
P   = pagerank_matrix(G)
emb = verse(G, P, 128, step_size=0.0025, steps=10_000)
np.save('verse.npy', emb)
```

Task 4.1.2 (3 points)

[Implement] a small piece of code that runs k-means on the embeddings with $k \in [2, 7]$ to evaluate the performance compared to Spectral clustering using the NMI as measure. You can use `sklearn.metrics.normalized_mutual_info_score` for the NMI and `sklearn.cluster.KMeans` for kmeans. In both cases, you can use your own implementation from Handin 1 or the exercises, but it will not give you extra points.

[Describe] which of the method performs the best and whether the results show similarities between the two methods

In [ ]:
```
### YOUR CODE STARTS HERE
from sklearn.cluster import KMeans, SpectralClustering
from sklearn.metrics import normalized_mutual_info_score


true_labels = email.communities


for k in range(2, 8):
    print(f"number of clusters k={k}")

    # Initialize the KMeans model
    kmeans = KMeans(n_clusters=k).fit(emb)
    k_labels = kmeans.labels_
    kmeans_nmi = normalized_mutual_info_score(labels_pred = k_labels, labels_true =

    #Our own spectral clustering
    spectral_clust = spect_cluster(G, k=k)
    spectral_nmi = normalized_mutual_info_score(labels_pred = spectral_clust, label

    print(f"K-means NMI={kmeans_nmi:.3f}, Spectral NMI={spectral_nmi:.3f}\n")

### YOUR CODE ENDS HERE
```

```
number of clusters k=2
K-means NMI=0.059, Spectral NMI=0.356

number of clusters k=3
K-means NMI=0.040, Spectral NMI=0.363

number of clusters k=4
K-means NMI=0.081, Spectral NMI=0.452

number of clusters k=5
K-means NMI=0.069, Spectral NMI=0.666

number of clusters k=6
K-means NMI=0.075, Spectral NMI=0.819

number of clusters k=7
K-means NMI=0.095, Spectral NMI=0.740
```

YOUR ANSWER HERE

The NMI masure is a measure of the similarity between two labels of the same data. The value is between 0 and 1, where 1 means the labels are the same. From the results we see that the NMI measure is higher for the Spectral clustering method. This means that the Spectral clustering method performs better than the K-means method. The K-means method shows very low values for the NMI measure, which means that the labels are very different. The Spectral clustering method shows higher values for the NMI measure, which means that the labels are more similar.

Task 4.1.3 (2 points)

[Motivate] how you would conceptionally expand the way of embedding a graph, if you had a multi-label-graph. E.g. meaning you have multiple labels and each edge needs to have exactly one of those. So you can also have multiple edges between the same nodes, as long as they have different labels.

YOUR ANSWER HERE

In a multi-label-graph, we can have multiple edges between the same nodes, where each edge has a different label. We can then use one-hot encoding to encode the edges going between any two nodes. And then we would get something like this:

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

Where each entry in our matrix is a one-hot encoded vector, where the vector has a 1 in the position of the label of the edge, and 0 otherwise. This way, we can have multiple edges

between the same nodes. And then by multiplying the vector $x = \bar{1}$ with each entry in the matrix as follows $x^T @ A @ x$, we can get the number of edges between any two nodes. And then we can use the same method as before to get the embeddings, as we notice that the matrix A will still be symatric.

Task 4.2 (8 points)

This is a hard exercise. Do it for fun or only if you are done with easier questions.

[Implement] a new GCN that optimizes for modularity. The loss function takes in input a matrix $C \in \mathbb{R}^{n \times k}$ of embeddings for each of the nodes. C represents the community assignment matrix, i.e. each entry C_{ij} contains the probability that node i belong to community j.

The loss function is the following

$$loss = -Tr(C^\top BC) + l\|C\|_2$$

where B is the modularity matrix that you will also implement, and l is a regularization factor controlling the impact of the L_2 regularizer. We will implement a two-layer GCN similar to the one implemented in the exercises, but the last layer's activation fucntion is a Softmax.

In [ ]:
```
import pykeen

# Adjacency matrix
G     = email.S_undir.copy()
A     = np.array(nx.adjacency_matrix(G, weight=None).todense())
I     = np.eye(A.shape[0])
A     = A + I # Add self loop

# Degree matrix
D = np.diag(np.sum(A, axis=1))

### YOUR CODE HERE

# Normalized Laplacian
L = I - np.linalg.inv(D)**(1/2) @ A @ np.linalg.inv(D)**(1/2)

# # Create input features
P = pagerank_matrix(G)
X = verse(G, P, 128, step_size=0.0025, steps=10_000)

### TODO your code here

X = torch.tensor(X, dtype=torch.float, requires_grad=True) # Indicate to pytorch th
As = torch.tensor(A, dtype=torch.float)
L = torch.tensor(L, dtype=torch.float)  # We don't need to learn this so no grad re
```

In [ ]:
```
# Define a GCN
class GCNLayer(nn.Module):
    def __init__(self, L, input_features, output_features, activation=F.relu):
        """
            Inputs:
```

```
                L:              The "Laplacian" of the graph, as defined above
                input_features:  The size of the input embedding
                output_features: The size of the output embedding
                activation:      Activation function sigma
        """
        super().__init__()

        ### TODO Your code here

        self.activation = activation
        self.W = nn.Parameter(torch.randn(input_features, output_features))
        self.L = nn.Parameter(L)

        ### TODO Your code here

    def forward(self, X):
        ### TODO Your code here

        X = torch.mm(torch.mm(self.L, X), self.W)
        if self.activation:
            X = self.activation(X)

        ### TODO Your code here
        return X
```

Define the modularity matrix and the modularity loss

In [ ]:
```
def modularity_matrix(A):
    B = None
    ### YOUR CODE HERE
    d = np.sum(A, axis=0)
    B = A - (1 / np.sum(d)) * np.outer(d, d)
    ### YOUR CODE HERE
    return torch.tensor(B, dtype=torch.float)

def modularity_loss(C, B, l = 0.01):
    ''' Return the modularity loss

        Args:
            C: the node-community affinity matrix
            B: the modularity matrix
            l: the regularization factor

        :return the modularity loss as described at the beginning of the exercise
    '''
    loss = 0
    ### YOUR CODE HERE

    loss = -torch.trace(torch.mm(torch.mm(C.T, B), C)) + l * torch.norm(C)

    ### YOUR CODE HERE
    return loss
```

Compute labels from communities

In [ ]:
```
### Compute labels from communities
labels = None
### YOUR CODE HERE

# Create labels from communities
```

```
labels = email.communities

### YOUR CODE HERE
```

Create the model

In [ ]:
```
from sklearn.preprocessing import LabelEncoder
import torch.nn.functional as F

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

### Encode the labels with one-hot encoding
def to_categorical(y):
    """ 1-hot encodes a tensor """
    num_classes = np.unique(y).size
    return np.eye(num_classes, dtype='uint8')[y]

def encode_label(labels):
    label_encoder = LabelEncoder()
    labels = label_encoder.fit_transform(labels)
    labels = to_categorical(labels)
    return labels, label_encoder.classes_

y, classes = encode_label(labels)
y = torch.tensor(y)

# Define convolutional network
in_features, out_features = X.shape[1], classes.size # output features as many as t
hidden_dim = 16

# Stack two GCN layers as our model
# nn.Sequential is an implicit nn.Module, which uses the layers in given order as t
gcn = nn.Sequential(
    GCNLayer(L, in_features, hidden_dim),
    GCNLayer(L, hidden_dim, out_features, None),
    nn.Softmax(dim=1)
)
gcn.to(device)
```

Out[ ]:
```
Sequential(
  (0): GCNLayer()
  (1): GCNLayer()
  (2): Softmax(dim=1)
)
```

Train the unsupervised model once

In [ ]:
```
l = 100
epochs = 2000

def train_model(model, optimizer, X, B, epochs=100, print_every=10, batch_size = 2,
    for epoch in range(epochs+1):
        y_pred = model(X)
        loss = modularity_loss(y_pred, B, l=l)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if print_bool == True:
```

```
            if epoch % print_every == 0:
                print(f'Epoch {epoch:2d}, loss={loss.item():.5f}')


B = modularity_matrix(A)
optimizer = torch.optim.Adam(gcn.parameters(), lr=0.01)
train_model(gcn, optimizer, X, B, epochs=epochs, print_every=100)
```

```
Epoch  0, loss=510.59735
Epoch 100, loss=-34.06787
Epoch 200, loss=-36.90991
Epoch 300, loss=-37.10010
Epoch 400, loss=-38.58630
Epoch 500, loss=-42.09119
Epoch 600, loss=-42.12708
Epoch 700, loss=-42.13049
Epoch 800, loss=-41.57556
Epoch 900, loss=-41.73877
Epoch 1000, loss=-41.84497
Epoch 1100, loss=-41.84741
Epoch 1200, loss=-42.09131
Epoch 1300, loss=-42.11194
Epoch 1400, loss=-42.11450
Epoch 1500, loss=-42.12646
Epoch 1600, loss=-42.12207
Epoch 1700, loss=-42.07544
Epoch 1800, loss=-41.79138
Epoch 1900, loss=-41.85376
Epoch 2000, loss=-41.81140
```

Evaluate your model using NMI. Since the initialization is random train the model 10 times and take the average NMI. Assign each node to the community with the highest probability. You should obtain an Average $\text{NMI} \approx 0.5$.

Plot the last graph with the nodes colored by commnitieis communities using `plot_graph` below.

Note: You have to create the model 5 times otherwise you are keeping training the same model's parameters!

In [ ]:
```
from sklearn.metrics.cluster import normalized_mutual_info_score

def plot_graph(G, y_pred):
    plt.figure(1,figsize=(15,5))
    pos = nx.spring_layout(G)
    ec = nx.draw_networkx_edges(G, pos, alpha=0.2)
    nc = nx.draw_networkx_nodes(G, pos, nodelist=G.nodes(), node_color=y_pred, node
    plt.axis('off')
    plt.show()


### YOUR CODE STARTS HERE

nmi_scores = []
for i in range(5):
    for j in range(2):
        if j == 0:
            gcn = nn.Sequential(
                GCNLayer(L, in_features, hidden_dim),
                GCNLayer(L, hidden_dim, out_features, None),
```

```
                nn.Softmax(dim=1)
            )
        B = modularity_matrix(A)
        optimizer = torch.optim.Adam(gcn.parameters(), lr=0.01)
        train_model(gcn, optimizer, X, B, print_bool=False)
        y_pred = gcn(X).detach().cpu().numpy()
        y_pred = y_pred.argmax(axis=1)
        nmi = normalized_mutual_info_score(labels_true = labels, labels_pred = y_pr
        nmi_scores.append(nmi)

print("NMI scores: ", np.mean(nmi_scores))
plot_graph(G, y_pred)

### YOUR CODE ENDS HERE
```

```
NMI scores:  0.5208822818575206
```