Assessed Coursework 2

CID 01252821

Problem 1

Part 1

Adjacency matrix

	v1	v2	v3	v4	v5	v6	v7
v1	0	1	0	1	0	0	1
v2	1	0	1	1	0	0	1
v3	0	1	0	0	0	0	1
v4	1	1	0	0	0	0	1
v5	0	0	0	0	0	0	1
v6	0	0	0	0	0	0	0
v7	1	1	1	1	1	0	0

Incidence matrix

	0	1	2	3	4	5	6	7	8
v1	1	1	1	0	0	0	0	0	0
v2	1	0	0	1	1	1	0	0	0
v3	0	0	0	1	0	0	1	0	0
v4	0	1	0	0	1	0	0	1	0
v5	0	0	0	0	0	0	0	0	1
v6	0	0	0	0	0	0	0	0	0
v7	0	0	1	0	0	1	1	1	1

Part 2

If two graphs are isomorphic:

If $G \cong G'$ then there is a bijection $\phi : V \to V'$ s.t. $\phi(v_1)\phi(v_2) \in E'$, and $v_1v_2 \in E$. Let's construct a permutation matrix P where each row and column corresponds to a vertex in V and V' respectively, meaning each row and column has exactly one entry of 1 and the rest are 0s:

$$P := \begin{cases} 1, & \text{if } \phi(v_i) = v'_j \\ 0, & \text{otherwise.} \end{cases}$$

For each pair of vertices $v_i, v_j \in G$ it follows $A_{v_iv_j} = 1$ if and only if $B_{\phi(v_i)\phi(v_j)} = 1$. From the construction of P we have $PA = BP$.

P is a permutation matrix, hence it's orthogonal and invertible (and $P^{-1} = P^T$) $\Rightarrow PAP^{-1} = B$.

If $PAP^{-1} = B$:

Let $PAP^{-1} = B$ for some permutation matrix, where A and B are the adjacency matrices of G and G' respectively. The permutation matrix P corresponds to a bijection ϕ between the vertcies of G and G': $P_{v_iv_j}$ implies $\phi(v_i) = v_j$.

$PAP^{-1} = B$ implies that for any vertices $v_i, v_j \in G$ if $A_{v_iv_j} = 1$ then $B_{\phi(v_i)\phi(v_j)} = 1$. For the same reason if $A_{v_iv_j} = 0$ then $B_{\phi(v_i)\phi(v_j)} = 0$. Therfore, v_i and v_j are adjacent in G if and only if $\phi(v_i)$ and $\phi(v_j)$ are adjacent in G'. Hence, $G \cong G'$.

Problem 2

Part 1

Part a

Let $G = (V, E)$ be an undirected wighted graph with vertices $V = \{v_1, ..., v_n\}$ and weights $w_{ij} \geq 0$. $W = (w_{ij})_{i,j=1,...,n}$ is the wighted adjacency matrix of the graph, and because the graph is undirected it follows $w_{ij} = w_{ji}$, or W is symmetric. The degree matrix D is defined as a diagonal matrix with the degrees $d_1, ..., d_n$ on the diagonal, where $d_i = \sum_{j=1}^{n} w_{ij}$, or D is also symmetric. Therfore, the symmetry of L follows from the symmetry of D and W, and the property that the difference of two symmetric matrices is also a symmetric matrix. Now, let's observe $f'Lf$:

$$f'Lf = f'(D-W)f = f'Df - f'Wf = \sum_{i=1}^{n} d_i f_i^2 - \sum_{i,j=1}^{n} f_i f_j w_{ij} =$$

$$= \frac{1}{2}\left(\sum_{i=1}^{n} d_i f_i^2 - 2\sum_{i,j=1}^{n} f_i f_j w_{ij} + \sum_{j=1}^{n} d_j f_j^2\right) = \frac{1}{2}\sum_{i,j=1}^{n} w_{i,j}\left(f_i - f_j\right)^2 \geq 0, \text{for all } f \in R^n,$$

hence L is positive semi-definite.

Let L has an eigendecomposition $LV = V\Lambda$, where the columns of V are eigenvectors $\in R^n$, and Λ is a diagonal matrix with diagonal elements equal to the corresponding eigenvalues. We have

$$V^T L V = \Lambda,$$

meaning each element of Λ is $\lambda_i = v_i^T L v_i \geq 0$, for $v_i \in R^n$, $i = 1, ..., n$. Therefore, the eigenvalues of L are non-negative.

Part b

We know that D and W are defined as

$$D = \begin{bmatrix} d_1 & 0 & ... & 0 \\ 0 & d_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & ... & 0 & d_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{n} w_{1j} & 0 & ... & 0 \\ 0 & \sum_{j=1}^{n} w_{2j} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & ... & 0 & \sum_{j=1}^{n} w_{nj} \end{bmatrix}, \quad W = \begin{bmatrix} w_{11} & w_{12} & ... & w_{1n} \\ w_{21} & w_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & w_{n-1n} \\ w_{n1} & ... & w_{nn-1} & w_{nn} \end{bmatrix},$$

leading to $L = D - W$ being

$$L = D - W = \begin{bmatrix} \sum_{j=1}^{n} w_{1j} - w_{11} & -w_{12} & ... & -w_{1n} \\ -w_{21} & \sum_{j=1}^{n} w_{2j} & \ddots & \vdots \\ \vdots & \ddots & \ddots & -w_{n-1n} \\ -w_{n1} & ... & -w_{nn-1} & \sum_{j=1}^{n} w_{nj} \end{bmatrix}.$$

It is obvious that the sum of the rows and columns of L is equal to 0. We will show that this means that L is singular. Let 1_n be the $n \times 1$ vector with each element 1, then

$$L1_n = 0,$$

hence, 1_n belongs to the null space, and $rank(L) \leq n-1 \Rightarrow L$ is singular. This implies that $det(L) = 0$, or $\lambda_1 ... \lambda_n = 0$. We already proved that the eigenvalues of L are non-negative. Therefore, there is at least one eigenvalue equal to 0, also meaning that 0 is the smallest eigenvalue of L.

Let's assume f is an eigenvector with eigenvalue 0, then (from above):

$$0 = f'Lf = \sum_{i,j=1}^{n} w_{i,j} \left(f_i - f_j\right)^2 .$$

We know that $w_{ij} \geq 0$, hence, if two vertices v_i and v_j are connected ($w_{ij} > 0$) then $f_i = f_j$. This means that f needs to be constant for all vertices connected by a path in the graph. All vertices of a connected component in an undirected graph can be connected by a path, therefore f needs to be constant on the whole connected component. Thus in a graph with only one connected component we only have the constant vector 1 as eigenvector with eigenvalue 0 - the indicator vector of the connected component.

Part 2

The code is located in the Python notebook. Approach description:

- Assume we are working with high-dimensional data $X \in R^{N \times D}$, and we want to reduce the dimensionality D to d, $d << D$.
- First, for the data in X, considering each data point as a node in a graph, we are constructing a weighted graph W (with not normalised wights) with the help of the `k-NN` algorithm (`kneighbors_graph` from `sklearn.neighbors`). The number of neighbours for `k-NN` can be adjusted according to the dimensionality of the data.
- Then, we are constructing the Laplacian matrix (which in theory is $L = D - W$) with the function `csgraph.laplacian`, where `csgraph` is from `scipy.sparse`. We are passing W to the function, after that the function internally computes D, and returns as e result $L = D - W$.
- Performing eigenvalue decomposition on the Laplacian matrix with the function `eigsh` from `scipy.sparse.linalg` and then selecting the top d smallest non-zero eigenvalues and their corresponding eigenvectors. The smallest eigenvalue as previously prooved should be 0, therefore we should skip it.
- Finally, we are returning the eigenvectors corresponding to the chosen eigenvalues, and they construct the new representation of the data in the reduced-dimensional space. Eevery row of the matrix formed by these eigenvectors represents the corresponding data point.

Problem 3

Part 1

We can notice that:

- A_{ij} counts the edge between i and j if it exists, contributing to E_l for each community l.
- $\frac{k_i k_j}{2m}$, summed over all i, j within the same community, will correspond to $\left(\frac{d_l}{2m}\right)^2$.
- $\delta(C(i), C(j)$ ensures we're only considering vertices pairs i, j within the same community, therefore we can rewrite the summation to be over all of the communites instead.

- Then we sum up the above expressions across all nodes in each community. This involves the handshaking lemma stating that the sum of all degrees is twice the number of edges (since each edge contributes to the degree count of both its endpoints), which helps us to simplify the expected number of edges within communities.

In the given expression $\delta(C(i), C(j)$ yields 1 if the vertices i and j are in the same community, meaning $C(i) = C(j)$, and 0 otherwise. Therefore, in the expression

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - P_{ij})\, \delta(C(i), C(j))$$

we can state that the only contributions to the sum are from pairs of vertices which are in the same cluster, $C(i) = C(j) \Rightarrow \delta(C(i), C(j)) = 1$. This means we can rewrite the sum to be a sum over the clusters instead of the vertix pairs. Grouping the contributions together by cluster we get:

$$Q = \sum_{l=1}^{N} \left(\frac{|E_l|}{m} - \left(\frac{d_l}{2m} \right)^2 \right),$$

where N represent the number of clusters, $|E_l|$ is the number of edges in G_l (number of edges joining vertices of module l), and d_l is the sum of the degrees of the vertices of l.

We can also rewrite th initial expression as

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - P_{ij})\, \delta(C(i), C(j)) = \sum_{l=1}^{N} (e_{ii} - a_i^2),$$

where e_{ij} represents the edges with one end vertices in community i and the other in j:

$$e_{ij} = \sum_{kl} \frac{A_{kl}}{2m} 1_{k \in C(i)} 1_{l \in C(j)},$$

and a_i represents the ends of edges attached to vertices in community i:

$$a_i = \frac{k_i}{m} = \sum_j e_{ij}.$$

In our expression we are interested in components in the same community, therefore we are only contributing to the sum with e_{ii}.

Part 2

First, we are loading the graph with `nx.read_edgelist(file_path, create_using=nx.Graph())` and check it is loaded as undirected by confirming that `G.is_directed()` returns `False`.

Then, I am processing the true lables, and computing the true modularity which is equal to 0.3137611028706121.

I first decided to apply the `Girvan-Newman` algorithm by using the function `girvan_newman()` from `networkx.algorithms.community`. The algorithm has complexity $O(m^2 n)$, where m is the number of edges (in our case $\sim 25k$ which is quite a lot), and n is the number of vertices.If we want to perform all iterations over all the communites generated via `communities_generator = nx_community.girvan_newman(G)` this would cots us hours because it is very computationally heavy. Still, I ran the code for the first 30 iterations, and it can be noticed that the modularity of the detected communities on iteration 1 is 0.0031089277455570302, whereas on iteration 30 it is 0.003701898620417084, meaning there is a slight improvement, indicating a stronger community structure, as the number of iterations along with the number of communities increases (iteration 1 consists of 21 communities, iteration 30 consists of 50 communities). Results can be seen in `Figure 1`.

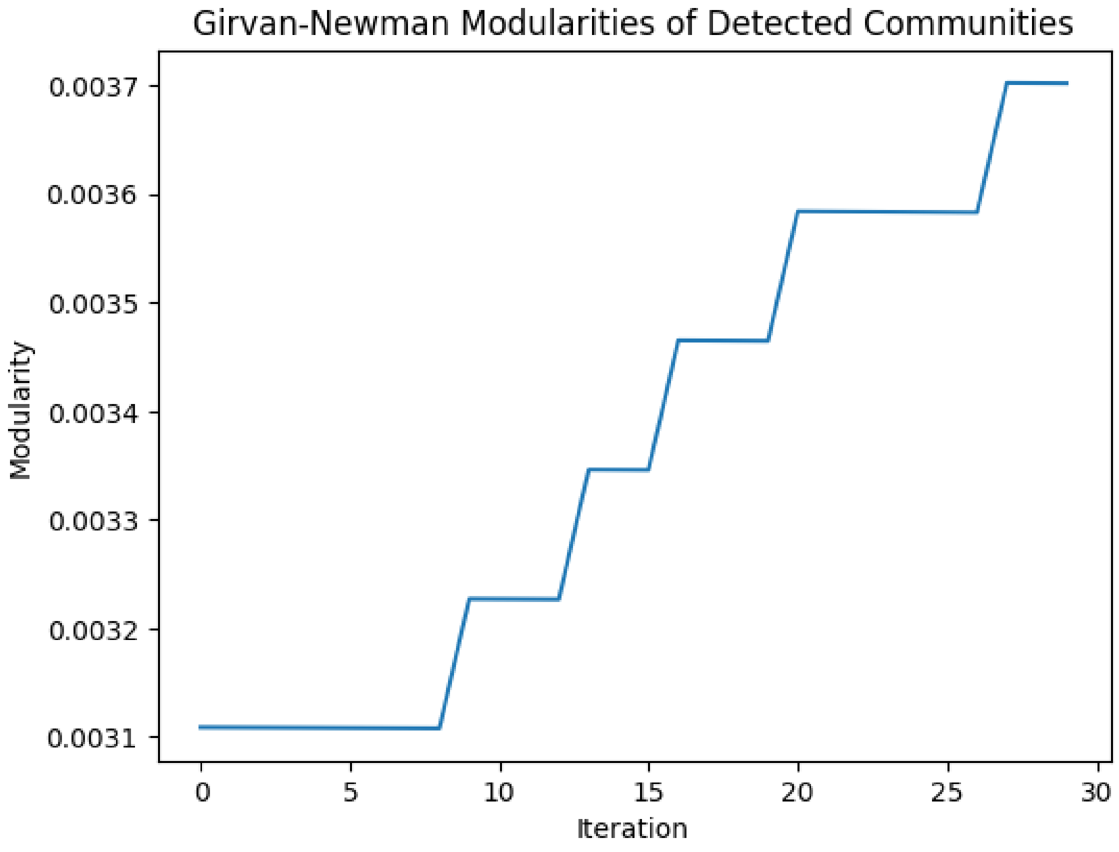

Figure 1: Modularities of 30 iterations of Girvan-Newman.

This made me search for other faster algorithms in the library `networkx`, and decided to attempt using `label_propagation_communities()`. The method is based on the concept of label propagation, where nodes update their community labels based on the labels of their neighbors. The idea is that each node is initially assigned a unique label, and at every step, each node adopts the label that most of its neighbors currently have. This makes the LPA faster and less computationally intensive than Girvan-Newman, which iteratively removes edges and calculates communities at each step. However, the label propagation may not explore the community space as thoroughly, potentially leading to less optimal community detection.

With applying `label_propagation_communities()` the modularity of the detected communities is 0.08900284744977784. The true modularity is significantly higher than that for the communities detected by the label propagation method. This suggests that the label propagation method may not have captured the community structure as effectively as the true underlying community labels.