- This homework is due at 11:59pm on April 22, 2022. Please submit by email to `natalies@cs.unc.edu+comp790`.
- There are a few files provided:
  - Protein-Protein Interaction Network 1, with edges determined according to co-expression in `Coexpress_Edges.csv`
  - Protein-Protein Interaction Network 2, with edges determined according to experimental information given in `Experimental_Edges.csv`
- You are welcome to consult with other colleagues, but please write up your own independent solution.
- You are welcome to use Python, Julia, or R here.
- You are welcome to write up your assignment using the `HW2_790-166.tex` template, or write up the solutions in the method of your choice.
- This homework is worth 62 points total.
- Please submit your final writeup as a PDF. Please try not to end me pages of output from Jupyter notebooks :) I simply want to see the few lines of code you used to answer each sub-question.
- Make sure to comment and elaborate on your answers in places where you are asked to comment!

Problem 1

Understanding the Rayleigh Ritz Theorem (12 points)
Here we will empirically explore the Rayleigh Ritz Theorem which says the following.

- Let $\mathbf{A} \in \mathbb{R}^{N \times N}$ be a square symmetric matrix with eigenvalues $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ and corresponding eigenvectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots \mathbf{v}_n\}$. Defining $R_{\mathbf{A}}(\mathbf{x}) = \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}}$, then the minimum value of $R_{\mathbf{A}}(\mathbf{x})$ is λ_1 and occurs when $\mathbf{x} = \mathbf{v}_1$.
- Obviously, this is a nice property to understand, as $\mathbf{x}^T\mathbf{A}\mathbf{x}$ represents the quadratic form that we are often trying to minimize.
- This property can be extended to find the matrix $\mathbf{X}$ that minimizes trace$(\mathbf{X}^T\mathbf{L}\mathbf{X})$. Specifically, the k-dimensional matrix, $\mathbf{X}$ that minimizes trace$(\mathbf{X}^T\mathbf{L}\mathbf{X})$ is the first k eigenvectors of $\mathbf{L}$ (e.g. those corresponding to the k smallest eigenvalues) and the minimum value obtained for trace$(\mathbf{X}^T\mathbf{L}\mathbf{X})$ will be $\lambda_1 + \lambda_2 + \cdots + \lambda_k$.

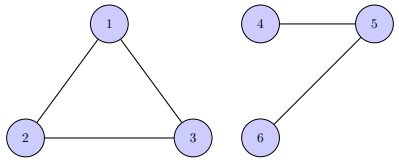

Figure 1: The graph, $\mathcal{G}$ that we already met in homework 1.

1. **Problem Setup 1** (1 point) Create the adjacency matrix, $\mathbf{A}$ for $\mathcal{G}$ and compute the graph Laplacian matrix, $\mathbf{L}$ for this graph. Note that you already did this in homework 1. Just copy it here!

   ### Solution
   I coded this problem in Julia.

```
# adjacency matrix
A = [
     [0 1 1 0 0 0];
     [1 0 1 0 0 0];
     [1 1 0 0 0 0];
     [0 0 0 0 1 0];
     [0 0 0 1 0 1];
     [0 0 0 0 1 0]
]
# degree and laplacian matrices
D = diagm(sum(A, dims = 1)[1, :])
L = D - A
```

```
6 6 Matrix{Int64}:
  2  -1  -1   0   0   0
 -1   2  -1   0   0   0
 -1  -1   2   0   0   0
  0   0   0   1  -1   0
  0   0   0  -1   2  -1
  0   0   0   0  -1   1
```

2. **Problem Setup 2** (1 point) Find the eigenvalues and eigenvectors of **L**. Note that you also have done this in homework 1.

Solution

```
using LinearAlgebra
Lambda = eigvals(L)
```

```
6-element Vector{Float64}:
 -1.1102230246251565e-16
  3.9250536344271737e-17
  0.9999999999999998
  3.0
  3.0
  3.0
```

```
V = eigvecs(L)
```

```
6 6 Matrix{Float64}:
 -0.57735   0.0        0.0             0.707107  -0.408248   0.0
 -0.57735   0.0        0.0            -0.707107  -0.408248   0.0
 -0.57735   0.0        0.0             0.0        0.816497   0.0
  0.0       0.57735   -0.707107        0.0        0.0       -0.408248
  0.0       0.57735    9.42055e-16     0.0        0.0        0.816497
  0.0       0.57735    0.707107        0.0        0.0       -0.408248
```

3. **Eigenvalue Sorting** (2 points). We first need to order the eigenvalues of **L** from largest to smallest. That is, you need to return the indices of eigenvalues that would sort them from smallest to largest.

Solution

```
# get indices of sorted eigenvalues
sort_idx = sortperm(Lambda)
```

```
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
```

4. **What Won't Be the Minimum** (2 points). Find the fourth smallest eigenvalue, λ_4 and its corresponding eigenvector $\mathbf{v}_4$. Evaluate $\mathbf{v}_4^T\mathbf{L}\mathbf{v}_4$ and write down the number that you get.

Solution

The value of $\mathbf{v}_4^T\mathbf{L}\mathbf{v}_4$ is equal to the fourth smallest eigenvalue.

```
lambda4 = Lambda[sort_idx[4]]
```

```
3.0
```

```
v4 = V[:, sort_idx[4]]
```

```
6-element Vector{Float64}:
  0.7071067811865476
 -0.7071067811865475
  0.0
  0.0
  0.0
  0.0
```

```
# total variation of fourth eigenvector
transpose(v4) * L * v4
```

```
3.0
```

5. **Compare to the Following, which will be smaller** (2 points). Find the first smallest eigenvalue, λ_1 and its corresponding eigenvector $\mathbf{v}_1$. Evaluate $\mathbf{v}_1^T\mathbf{L}\mathbf{v}_1$ and write down the number that you get. Comment on this wrt what you got using the fourth eigenvalue/eigenvector.

Solution

The value of $\mathbf{v}_1^T\mathbf{L}\mathbf{v}_1$ is equal to the smallest eigenvector (both are 0). This is indeed smaller than using λ_1.

```
lambda1 = Lambda[sort_idx[1]]
```

```
-1.1102230246251565e-16
```

```
v1 = V[:, sort_idx[1]]
```

```
6-element Vector{Float64}:
 -0.5773502691896257
 -0.5773502691896255
 -0.5773502691896256
  0.0
  0.0
  0.0
```

```
transpose(v1) * L * v1
```

```
7.395570986446986e-32
```

6. **Forming the Non-Optimal 2d Embedding** (2 points). Now we will use the eigenvectors that minimize trace($\mathbf{X}^T\mathbf{L}\mathbf{X}$). Form a matrix with two columns where the first column is the third eigenvector $\mathbf{v}_3$ and the second column is the fourth eigenvector $\mathbf{v}_4$. Define the embedding matrix, $\mathbf{E}$ as the matrix that horizontally concatenates $\mathbf{v}_3$ and $\mathbf{v}_4$ as $\mathbf{E} = [\mathbf{v}_3|\mathbf{v}_4]$.

   - Compute trace($\mathbf{E}^T\mathbf{L}\mathbf{E}$). Record what you get.
   - Compute $\lambda_3 + \lambda_4$ and comment about what you get with respect to the trace you just computed.

   ## Solution

   The trace of the total variation on the embedding matrix is equal to the sum of the corresponding eigenvalues.

```
E34 = V[:, sort_idx[3:4]]
```

```
6×2 Matrix{Float64}:
  0.0          0.707107
  0.0         -0.707107
  0.0          0.0
 -0.707107     0.0
  9.42055e-16  0.0
  0.707107     0.0
```

```
tr(transpose(E34) * L * E34)
```

```
4.0
```

```
sum(Lambda[sort_idx[3:4]])
```

```
4.0
```

7. **Forming the Optimal 2 Embedding** (2 points) Now we will use the eigenvectors that minimize trace($\mathbf{X}^T\mathbf{L}\mathbf{X}$). Form a matrix with two columns where the first column is the first eigenvector $\mathbf{v}_1$ and the second column is the second eigenvector $\mathbf{v}_2$. Define the embedding matrix, $\mathbf{E}$ as the matrix that horizontally concatenates $\mathbf{v}_1$ and $\mathbf{v}_2$ as $\mathbf{E} = [\mathbf{v}_1|\mathbf{v}_2]$.

   - Compute trace($\mathbf{E}^T\mathbf{L}\mathbf{E}$). Record what you get.
   - Compute $\lambda_1 + \lambda_2$ and comment about what you get with respect to the trace you just computed.

Solution

The trace of the total variation on the embedding matrix is approximately zero, and is close to the sum of the first two eigenvalues.

```
E12 = V[:, sort_idx[1:2]]

        6 2 Matrix{Float64}:
         -0.57735   0.0
         -0.57735   0.0
         -0.57735   0.0
          0.0       0.57735
          0.0       0.57735
          0.0       0.57735

tr(transpose(E12) * L * E12)

    5.2015515938010466e-30

sum(Lambda[sort_idx[1:2]])

    -7.177176611824392e-17
```

Problem 2

(50 Points Total) **Protein-Protein Interaction (PPI) Graph Alignment**

Two protein-protein interaction networks for Humans were downloaded from the string database `https://string-db.org/` and pre-processed to produce sufficiently large but not too large subgraphs. In the first graph, edges were determined according to co-expression information `Coexpress_Edges.csv`. In the second graph, edges were determined according to validated, experimental information. Our challenge is to apply a graph alignment technique to see if the same proteins map to each other between these two graphs.
Recall REGAL alignment `https://arxiv.org/pdf/1802.06257.pdf`. The following homework sub-problems will **walk us through implementing the REGAL graph alignment approach**.

1) **Constructing Node Features (5 points):** The first part of REGAL is to create a feature vector for each node that helps to summarize something about its context. We will use a simple k-hop method to construct a feature vector for each node. Recall that for a node, i, its 'k-hop subgraph' can be obtained by considering nodes that are within k hops from i. (Hint: you may find the following useful `https://networkx.org/documentation/stable//reference/generated/networkx.generators.ego.ego_graph.html`).

We will consider k-hop networks for $k = 1, 2, 3, 4$. **Write a function, where for a particular k, you collect the set of neighboring nodes within k hops of each node and summarize the degree distribution of these collective 'k-hop neighbors' with 4 statistics : {min degree, median degree, mean degree, max degree}.** After doing this for each value of k, you should ultimately be able to represent each node with 16 features (4 considered hops $\times$ 4 summary statistics per hop). As an example, assuming Graph 1 has N_1 nodes, define its node feature matrix, $\mathbf{X}_1 \in \mathbb{R}^{N_1 \times 16}$ matrix.

Solution

My naive solution to this problem is to find the subgraphs centered at every node, including nodes of distance k or less. Then, I can find the degrees in the original graph for each node in a subgraph. This approach is simple to implement, however it is computationally expensive (finding all nodes for Graph 1 took over an hour). I will improve upon this method in question 9.

```
import numpy as np
import networkx as nx

def get_khop_degree_distribution(graph, k):
    degree_stats = []
    for node in graph.nodes:
        subgraph = nx.generators.ego.ego_graph(graph, node, radius = k)
        degrees = np.array([graph.degree[nbr] for nbr in subgraph.nodes])
        node_degree_stats = np.array(
            [degrees.min(), np.median(degrees), np.mean(degrees), degrees.max()]
        )
        degree_stats.append(node_degree_stats)
    return(np.array(degree_stats))
```

2) **Intuition Building (5 points):** Use your new function to build the described feature vectors for Supragingival Plaque Network (Network 1). Assuming this network has N_1 nodes, **project these N_1 nodes into two dimensions using your dimensionality reduction method of choice**, based on the 16 computed features ($\mathbf{X}_1 \in \mathbb{R}^{N_1 \times 16}$).

Solution

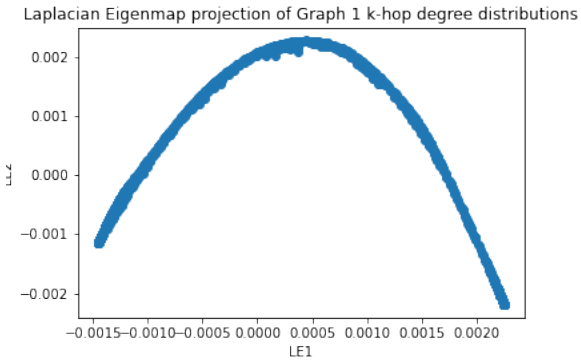

Figure 2: The projection computed based on the k-hop degree distribution features.

3) **Choosing Landmarks (5 points):** Recall that REGAL constructs an embedding for each node by specifying landmark nodes that have been collected across both of the graphs being aligned. Choose a set of d landmark nodes **collectively** across Graphs 1 and 2. You can play with d later, but perhaps $d = 30$ is a good place to start. You can choose the set of d landmarks at random, or use a more sophisticated approach. **Explain your choice of landmarks and write a function to return these landmark nodes**.

Solution

I wanted to choose landmarks that are central to all other nodes. This would be desirable, because nodes on the periphery can be described by their relative distance to the hub nodes. I used the eigenvector centrality

as measure of node centrality. For each graph, I selected a number of landmark nodes proportional to the number of nodes in the graph. I selected the nodes with the highest eigenvector centrality from each graph.

```
def choose_central_landmarks(graph, d):
    cent = nx.eigenvector_centrality(graph)
    top_cent = sorted(cent.items(), key = lambda item: item[1])[-d:]
    landmarks = list(dict(top_cent).keys())
    return landmarks


def allocate_landmarks(graph1, graph2, d = 30):
    # get representative proportion of landmarks from each graph
    n1 = graph1.number_of_nodes()
    n2 = graph2.number_of_nodes()
    d1 = np.round(n1*d/(n1 + n2)).astype(int)
    d2 = d - d1
    landmarks1 = choose_central_landmarks(graph1, d1)
    landmarks2 = choose_central_landmarks(graph2, d2)

    return landmarks1, landmarks2
```

4) **Computing Similarities to Landmarks (5 points):** In part 1), you wrote a function to compute feature vectors for each node. Assuming Graph 1 has N_1 nodes and Graph 2 has N_2 nodes, **write a function that computes a similarity measure in this 16-dimensional space between each of the nodes in Graph 1 and Graph 2 to each of the d landmarks.** So, you should en up with a matrix, $\mathbf{C} \in \mathbb{R}^{(N_1+N_2)\times d}$.

Solution

For this problem, I computed the similarity using cosine similarity. In order to subset the landmark nodes, I selected nodes from the feature matrices (`strucs`) using the indexing functionality in `pandas.DataFrames`. I could then pass the landmarks directly from the `allocate_landmarks` function.

```
import numpy as np
import pandas as pd
from scipy.spatial import distance


def compute_node_similarity_to_landmarks(strucs, landmark_set):
    all_nodes = pd.concat(strucs)
    landmark_nodes = []

    for struc, landmarks in zip(strucs, landmark_set):
        # this requires that the indices are set to the node names!
        landmark_strucs = struc.iloc[struc.index.isin(landmarks), :]
        landmark_nodes.append(landmark_strucs)

    landmark_nodes = pd.concat(tuple(landmark_nodes))
    n = all_nodes.shape[0]
    d = landmark_nodes.shape[0]
    sim = np.zeros((n, d))
```

```
    for i, node in enumerate(all_nodes.iterrows()):
        for j, landmark in enumerate(landmark_nodes.iterrows()):
            sim[i, j] = 1 - distance.cosine(node[1], landmark[1])

    return sim
```

5) **Extract Landmark × Landmark Matrix (5 points):** As you know, the $\mathbf{C}$ that you constructed contains the d landmark nodes! Write a function to construct $\mathbf{W} \in \mathbb{R}^{d \times d}$ submatrix of $\mathbf{C}$ where the similarities between the landmarks were stored.

Solution

If I know the landmark indices, I can just subset the similarity matrix. I perform this in `get_landmark_similarity`. However, I need to first get the landmark indices. In the case that the landmark nodes are represented as integers, this is trivial. In many graphs, though, the nodes are represented as strings. I come up with a workaround, `get_landmark_idx`, which takes advantage of the `networkx` API to extract the landmark indices for graphs with any kind of node labels.

```
def get_landmark_idx(graphs, landmark_set):
    n = 0
    landmark_idx = []
    for graph, landmarks in zip(graphs, landmark_set):
        for i, node in enumerate(graph.nodes):
            if node in landmarks:
                landmark_idx.append(i + n)
        n += graph.number_of_nodes()
    return landmark_idx

def get_landmark_similarity(sim, landmark_idx):
    landmark_sim = sim[landmark_idx, :]
    return landmark_sim
```

6) **Embedding via Landmarks (5 points)**: Given Theorem 3.1 in `https://arxiv.org/pdf/1802.06257.pdf`, we can compute the collective node embedding matrix (across Network 1 and Network 2), $\tilde{\mathbf{Y}} \in \mathbb{R}^{(N_1+N_2) \times d}$, as

$$\tilde{\mathbf{Y}} = \mathbf{C}\mathbf{U}\mathbf{\Sigma}^{1/2}$$

.

Recall that here, $\mathbf{U}$ and $\mathbf{\Sigma}$ are obtained through an SVD on the pseudo inverse ($\mathbf{W}^{\text{pinv}}$) of the (landmark × landmark) similarity matrix, $\mathbf{W} \in \mathbb{R}^{d \times d}$ extracted from $\mathbf{C}$.

$$\mathbf{W}^{\text{pinv}} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

Hints: These are useful for pseudoinverse (`https://numpy.org/doc/stable/reference/generated/numpy.linalg.pinv.html`) and SVD (`https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html`).

Given this information, write a function to compute $\tilde{\mathbf{Y}}$.

Solution

```
import numpy as np
def calculate_node_embeddings(C, W):
    W_pinv = np.linalg.pinv(W)
    U, Sigma, _ = np.linalg.svd(W_pinv)
    emb = C @ U @ np.diag(Sigma**(0.5))
    return emb
```

7) **Putting it All Together Visualization 1 (5 points):** You have now defined an embedding for all nodes in Networks 1 and 2 in some d-dimensional space through $\tilde{\mathbf{Y}}$. **Use your favorite dimensionality reduction method of choice to project the collective set of nodes in Networks 1 and 2 into two dimensions. Color the nodes by which network they are from. Comment on any observations.**

Solution

REGAL produces similar embeddings between the two graphs–they are on top of each other in the 2-d Laplacian Eigenmap projection.

Solution

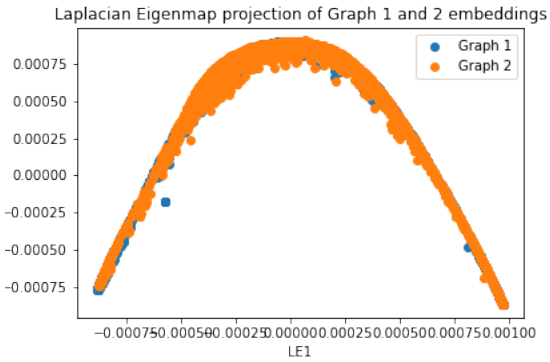

Figure 3: Node embeddings from Graphs 1 and 2 projected on top of each other.

8) **Alignment Between Graphs (5 points):** Given $\tilde{\mathbf{Y}}$, **calculate a similarity score (your choice) between each node in Network 1 and every node in Network 2.**

Solution

I chose to calculate the alignment score between the nodes using a Gaussian kernel on the Euclidean distance. This worked substantially faster than cosine similarity for the dimension of the embeddings.

```
def calculate_alignment_score(emb1, emb2):
    n = emb1.shape[0]
    m = emb2.shape[0]
    alignment = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            dist_ij = np.linalg.norm(emb1[i, :] - emb2[j, :])
            alignment[i, j] = np.exp(-dist_ij)
    return alignment
```

```
alignment = calculate_alignment_score(graph1_emb, graph2_emb)

        array([[0.81249237, 0.79552233, 0.89632289, ..., 0.72853754, 0.74441511,
                0.73037866],
               [0.9328034 , 0.89880404, 0.89825136, ..., 0.63357656, 0.64018991,
                0.63295388],
               [0.89156008, 0.86247563, 0.91617763, ..., 0.66233625, 0.67103654,
                0.66276606],
               ...,
               [0.87786695, 0.8682243 , 0.81402309, ..., 0.58205669, 0.58304653,
                0.58304967],
               [0.88871344, 0.88531273, 0.82133514, ..., 0.58395997, 0.58489599,
                0.58481202],
               [0.84022633, 0.84884531, 0.84713515, ..., 0.62975178, 0.63372547,
                0.63337026]])
```

9) **Creativity (5 points):** Now that you have the entire pipeline in place, play around with it a bit. For example, considering changing how you define the features for nodes in part 1), changing the value of d, changing how you choose landmarks, or anything else that is interesting to you! **Re-run steps 1-7 with your modification and comment on how it changes the interpretation of alignment between Network 1 and Network 2 given in $\tilde{\mathbf{Y}}$.**

Solution

I was pleased with the results of the alignment, however I found that the k-hop neighbor step took too long. I also realized that by the fourth k-hop, most of the nodes that were included in the degree distribution had already been encountered before. So, I decided to change the k-hop degree distribution method.

I came up with two main changes. First, rather than creating a subgraph for every node, I would perform a random walk by powering the graph adjacency matrix. This would tell me which nodes could be reached after a walk of length k. Even better, this works for every node simultaneously. Next, I would keep track of which nodes had been visited before during random walks from each possible starting node. This could be done in a Boolean matrix, which I called H, for history. Then, for the kth step of the random walk, I could extract degree distributions from only those nodes which had not been visited before. H would then be updated.

Since most nodes would be visited after a small number of k steps from any starting node, I found it more efficient to store the logical complement of H in sparse matrix format.

The resulting embeddings were similar to the original algorithm, however finding k-hop neighbors became much faster. The time to produce the k-hop neighbor distributions for Graph 1 decreased from over an hour for the original implementation, to just 10 seconds!

Output:

```
get_khop_degree_distribution(G1, k = 4)

        finding 1-hop degree distributions...
        found 1-hop degree distributions in 3.30s.
        finding 2-hop degree distributions...
```

```
found 2-hop degree distributions in 3.74s.
finding 3-hop degree distributions...
found 3-hop degree distributions in 2.50s.
finding 4-hop degree distributions...
found 4-hop degree distributions in 1.54s.
```

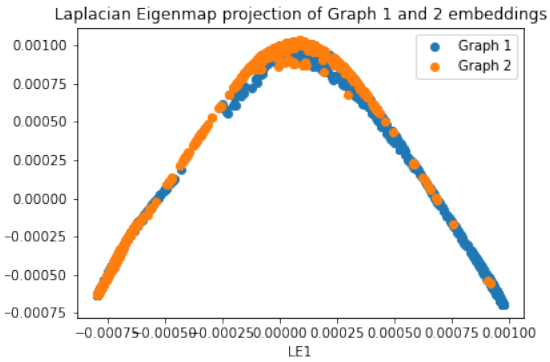

Figure 4: Node embeddings from Graphs 1 and 2 projected on top of each other, using improved k-hop degree distribution method.

Code:

```
def get_khop_degree_distribution(G, k):

    deg_distr = []

    n = G.number_of_nodes()
    A = nx.adjacency_matrix(G)
    K = A.copy()
    # define history: each node starts at itself
    H = np.identity(n, dtype = bool)
    # define nodes that have not been visited yet.
    # starts dense, but will become sparser with further iterations.
    Hnot = sp.csr_matrix(np.logical_not(H))
    for i in range(k):
        print(f"finding {i+1}-hop degree distributions...")
        start = perf_counter()
        # define candidate steps in random walk
        J = K @ A
        J = J.astype(bool)
        # update K to include only nodes that have not been visited yet.
        # using Hadamard product (elementwise multiplication), translates to
        # logical AND.
        K = J.multiply(Hnot)
        # update Hnot by subtracting nodes that got visited in step i
        Hnot = (Hnot.astype(int) - K.astype(int)).astype(bool)

        deg_distr_i = np.zeros((n, 4))
```

```
        for node in range(n):
            k_hop_neighbor_idx = K[node, :].nonzero()[1]
            k_hop_neighbors = np.array(G.nodes())[k_hop_neighbor_idx]
            degrees = G.degree(k_hop_neighbors)
            degrees = list(dict(degrees).values())

            if not degrees: # check if empty: i.e. if no more unvisited nbrs
                deg_distr_i[node, :] = [0, 0, 0, 0]
            else:
                deg_distr_i[node, :] = [np.min(degrees), np.median(degrees),
                                        np.mean(degrees), np.max(degrees)]

        deg_distr.append(deg_distr_i)

        stop = perf_counter()
        print(f"found {i+1}-hop degree distributions in {stop-start:.2f}s.")

    return np.hstack(tuple(deg_distr))
```

10) **Creativity Part 2 (5 points):** Imagine a collaborator dropped these two networks on your desk. They are paying you from their grant, so you need to produce something to give them. **Create a visualization of your choice that reflects something about the similarity between Network 1 and Network 2** (in terms of node alignment, clustering structure, etc).

Solution

I plotted the alignment scores on a heatmap, and clustered the axes to group proteins that align to similar proteins. This should give an overall sense of how similar the two PPI networks are. Of note, is that there are some proteins with strong alignment between the two networks, especially in the bottom left. However, in the middle right, there is a large group of proteins from the coexpression network which do not align well to the experimental network. Investigating these nodes may be a promising next step.

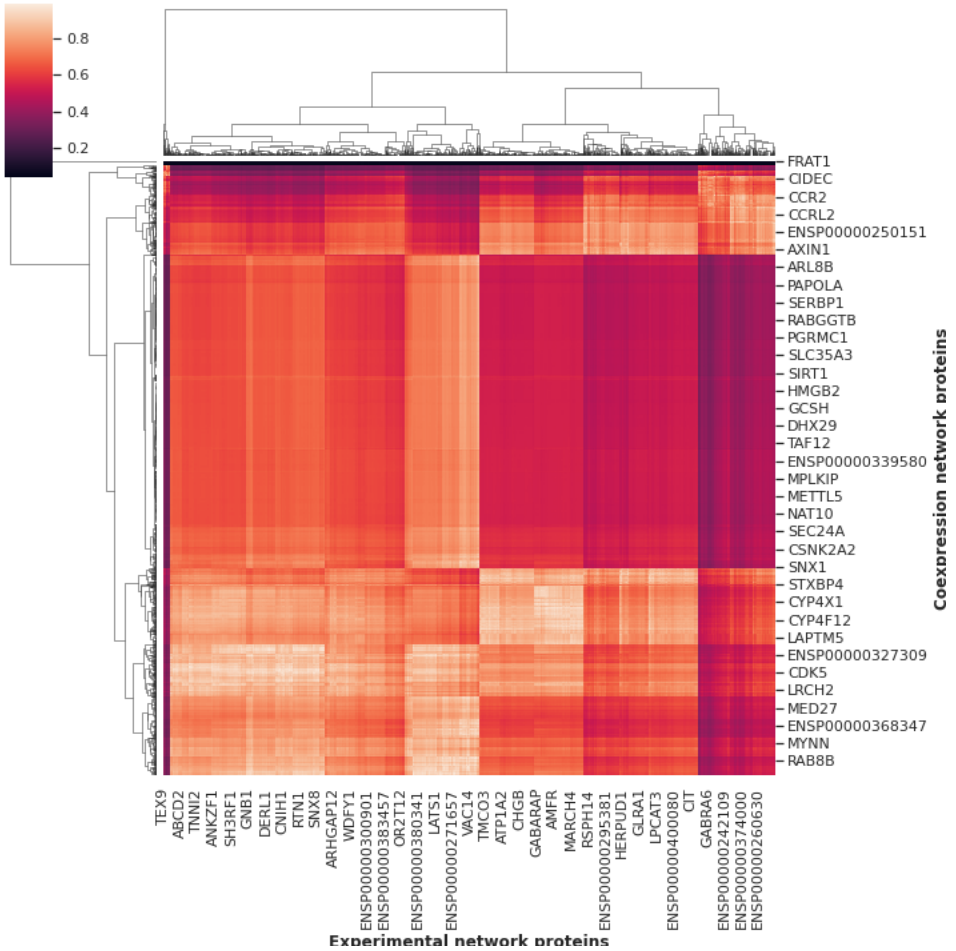

Figure 5: Alignment score between proteins from coexpression and experimental PPI networks.

Congratulations! You implemented REGAL from scratch!