Lab 4 Power Iteration and Link Preidction

Due: Midnight, October 9th

In this lab, we will introduce

1. **Power iteration for Eigenvector Centrality**
2. **Eigendecompstion for Eigenvector Centrality**
3. **Node Similarity**
4. **Link Preidciton`**

Save Your Notebook!

- Click on File (upper left corner), Select “Save” or press Ctrl+S.
- Important: You may loose your modification to a notebook if you do not Save it explicitly.
- Advice: Save often.

Submission

- Please follow the instructions and finish the exercises.
- After you finish the lab, please Click on File, Select “Download .ipynb”
- After download is complete, Click on File, Select “Print”, and and Choose ``Save as PDF"
- Submit both the Notebook file and the PDF File as your submission for Lab 4.

1. Preparation

Before we start to visualize the networks, we have to install the packages and prepare the network dataset.

1.1 Connect this Colab notebook with your Google Drive

```
# The following code will mount the drive
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call d
```

1.2. Install Packages

The following packages should be avaialbe in Colab. In case not, run the following codes

```
!pip install matplotlib
!pip install networkx
!pip install numpy
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-whe
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dis
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dis
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-package
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-whe
Requirement already satisfied: networkx in /usr/local/lib/python3.7/dist-package
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-whe
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (
```

1.2. Import and Visualize the Graph

We will use the same undirected weighted graph as we used in Lab 3. If you didn't upload to Google drive, please download it from Canvas and upload it to the folder DS420 of Google Drive. Next, we will load the graph and visualize it. In particular, the width of the edge is based on the weight of edge.

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

```
%matplotlib inline
plt.figure(figsize=(6,6))
G = nx.read_edgelist(path="/content/gdrive/My Drive/DS420/undirected_weighted.edgelist
pos = nx.fruchterman_reingold_layout(G)
edges = []
weights = []
for (source, target, weight) in G.edges.data('weight'):
    edges.append((source, target))
    weights.append(weight)
nx.draw_networkx_nodes(G, pos, node_size=400, node_color='orange')
nx.draw_networkx_edges(G, pos, edgelist=edges, width=weights*32)
nx.draw_networkx_labels(G, pos, font_size=12)
plt.show()
```

You should see the visualization of the graph. The thicker the edge, the larger the weight. The weight plays an important role in eigenvector centrality.

2. Power Iteration for Eigenvector Centrality

We will implement power iteration for eigenvector centrality. In eigenvector centrality, a node aggregate its neighbors' centralities as

$$C_e(v_i) = \frac{1}{\lambda} \sum_{v_j \in \mathcal{N}^{in}(v_i)} A_{ji} C_e(v_j)$$

where $\mathcal{N}^{in}(v_i)$ denotes the set of nodes that have incoming links to node v_i. It is a recursive definition as the calcualtion of $C_e(v_i)$ depends on $C_e(v_j)$, and the calcualtion of $C_e(v_j)$ depends on $C_e(v_i)$. We will use power iteration to learn the eigenvector centralities. The basic idea of power iteration is that every node starts with the same score, then each node gives away its score to its successors. We iteratively update the score until convergence. The pseduo code is given below

Power Iteration

1. Initialization: set $C_e(v_1), C_e(v_2), \dots, C_e(v_N)$ to 1
2. Update $C_e(v_1), C_e(v_2), \dots, C_e(v_n)$ based on the equation
$$C_e(v_i) \leftarrow \sum_{v_j \in \mathcal{N}^{in}(v_i)} A_{ji} C_e(v_j)$$
3. Normalize each term as
$$C_e(v_i) \leftarrow \frac{C_e(v_i)}{\sqrt{\sum_{j=1}^{N} C_e(v_j)^2}}$$
4. Repeat 2 and 3 until convergence

2.1 Basic Operations

Before we implement the power iteration. Let's first introduce some basic operations

```
# Get nodes of a Graph: G.nodes
G.nodes
```

```
# Get neighbor of node 1:  G.neighbors(1)
current_node = 1
list(G.neighbors(current_node))
```

```
# Get weights of edges connecting to node 1
for neighbor in G.neighbors(current_node):
    print(G.get_edge_data(neighbor, current_node)['weight'])

# With the above operations, we can update the eigenvector centrality of a node as fol
Ce = {node:1.0 for node in G.nodes} # create a dictionary with key as the nodes and va
new_score = 0
for neighbor in G.neighbors(current_node):
    new_score += Ce[neighbor] * G.get_edge_data(neighbor, current_node)['weight'] # ac
print(new_score)


# If you are familar with Python, the above code can also be written as
Ce = {node:1.0 for node in G.nodes} # create a dictionary with key as the nodes and va
new_score = sum(Ce[neighbor] * G.get_edge_data(neighbor, current_node)['weight'] for n
print(new_score)
```

2.2 Implementing Power Iteration

With the basic operations above, we can implement power iterations. Note we use **maxiter** to control the iterations as we want to show how the eigenvector centralarity changes.

```
# create a dictionary with key as the nodes and values are 1
Ce = {node:1.0 for node in G.nodes}
maxiter = 10
Ce_record = np.ones((len(G), maxiter+1)) # we use this to record how Ce changes
Ce_tmp = {node:0.0 for node in G.nodes}  # this is used to store the intermediate valu


# main loop
for i in range(0, maxiter):

    # for each node, calculate their new eigenvector score and put in Ce_tmp
    for current_node in Ce.keys():
        # aggregate the centrality score from connected neighbors
        Ce_tmp[current_node] = sum(Ce[neighbor] * G.get_edge_data(neighbor, current_no

        """
        # If you are not familar with Python, the above line is equivalent to the foll
        tmp = 0
        for neighbor in G.neighbors(current_node):
          tmp += Ce[neighbor] * G.get_edge_data(neighbor, current_node)['weight']
        Ce_tmp[current_node] = tmp
        """

    # normalization
    normalization_term = sum(Ce_tmp[node]**2 for node in Ce_tmp) ** 0.5
    for node in Ce:
        Ce[node] = Ce_tmp[node] / normalization_term
```

```
    # record the values
    for node,j in zip(Ce, range(0, len(G))):
        Ce_record[j,i+1] = Ce[node]



# results
print('eigenvector centralities: {}'.format(Ce))

# visualize how the centrality changes
plt.figure()
plt.plot(np.transpose(Ce_record))
plt.title('Eigenvector Centrality in Each Iteration')
plt.xlabel('Iteration')
plt.ylabel('Eigenvector Centrality')
plt.show()

# visualiz the graph with node size reflecting the centrality
plt.figure(figsize=(6,6))
nodesize = [Ce[node]*1200 for node in Ce]
nx.draw_networkx(G, pos, with_labels=True, node_size=nodesize, font_size=12, node_colo
plt.show()
```

From the above visualization, we can observe that after 3 iterations, the algorithm converges.

Exercise 1: Power Iteration for Katz Centraltiy

Following the above example, please implement power iteration for Katz Centrality. The equation for Katz centrality is

$$C_k(v_i) = \alpha \sum_{v_j \in \mathcal{N}^{in}(v_i)} A_{ji} C_k(v_j) + \beta$$

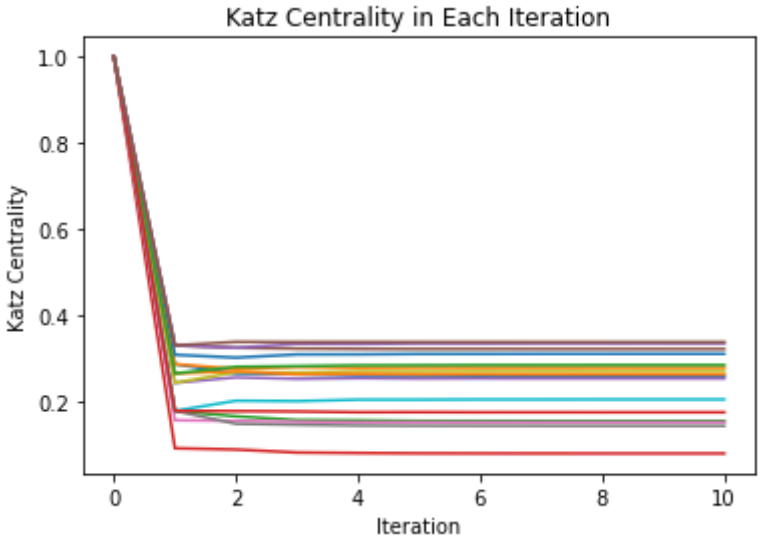
Please set $\alpha = 0.85, \beta = 0.15$

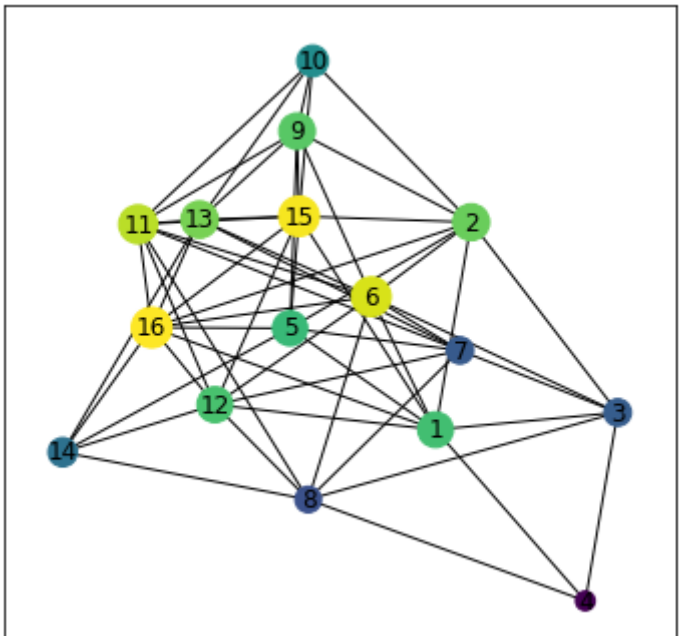
Power Iteration

1. Initialization: set $C_k(v_1), C_k(v_2), \dots, C_k(v_N)$ to 1
2. Update $C_k(v_1), C_k(v_2), \dots, C_k(v_n)$ based on the equation
$$C_k(v_i) \leftarrow \sum_{v_j \in \mathcal{N}^{in}(v_i)} A_{ji} C_k(v_j) + \frac{\beta}{\alpha}$$
3. Normalize each term as
$$C_k(v_i) \leftarrow \frac{C_k(v_i)}{\sqrt{\sum_{j=1}^{N} C_k(v_j)^2}}$$
4. Repeat 2 and 3 until convergence

```
# TODO
# Hint: the only difference with eigenvector centrality is in step 2, we need to add k
alpha = 0.85
beta = 0.15

# Your code here:
# create a dictionary with key as the nodes and values are 1
Ce = {node:1.0 for node in G.nodes}
maxiter = 10
Ce_record = np.ones((len(G), maxiter+1)) # we use this to record how Ce changes
Ce_tmp = {node:0.0 for node in G.nodes}  # this is used to store the intermediate valu

# main loop
for i in range(0, maxiter):

    # for each node, calculate their new eigenvector score and put in Ce_tmp
    for current_node in Ce.keys():
        # aggregate the centrality score from connected neighbors
        Ce_tmp[current_node] = (beta/alpha)+ sum( Ce[neighbor] * G.get_edge_data(neigh


    # normalization
    normalization_term = sum(Ce_tmp[node]**2 for node in Ce_tmp) ** 0.5
    for node in Ce:
        Ce[node] = Ce_tmp[node] / normalization_term

    # record the values
    for node,j in zip(Ce, range(0, len(G))):
        Ce_record[j,i+1] = Ce[node]


print('katz centralities: {}'.format(Ce))

# visualize how the centrality changes
plt.figure()
plt.plot(np.transpose(Ce_record))
plt.title('Katz Centrality in Each Iteration')
plt.xlabel('Iteration')
plt.ylabel('Katz Centrality')
plt.show()

# visualiz the graph with node size reflecting the centrality
plt.figure(figsize=(6,6))
nodesize = [Ce[node]*1200 for node in Ce]
nx.draw_networkx(G, pos, with_labels=True, node_size=nodesize, font_size=12, node_colo
plt.show()
```

```
katz centralities: {1: 0.2586565545266479, 2: 0.27819120071068093, 3: 0.15403960
```





2.3 Vectorization

The equation for eigenvector centrality can be vectrozied as

$$\mathbf{c} = \frac{1}{\lambda}\mathbf{A}^T\mathbf{c}$$

Where $\mathbf{c}$ is a vector of size N with the i-th element as the eigenvector centrality of the i-th node in the graph. With the vectorized version, the implementation is much easier.

Power Iteration

1. Initialization: set **c** to all one vector
2. Update **c** based on the equation

$$\mathbf{c} \leftarrow \mathbf{A}^T\mathbf{c}$$

3. Normalize each term as

$$\mathbf{c} \leftarrow \frac{\mathbf{c}}{\|\mathbf{c}\|_2}$$

4. Repeat 2 and 3 until convergence

basic operations

We will first introduce some basice operations

```
# we can use nx.adjacency_matrix(G) to get the adjacency matrix of G
A = nx.adjacency_matrix(G).todense()
print(A)
plt.matshow(A)
```

The visualization of the adjency matrix has 3 colors, which represent there weights, yellow: 2, green: 1 and blue: 0

```
# create a vector of size (A.shape[0], 1)
c = np.ones((A.shape[0], 1))
print(c)
```

```
# update the the centrality score with the equation
c = np.dot(np.transpose(A), c)  # A^T c
print(c)
```

```
# normalization
c = c / np.linalg.norm(c)
print(c)
```

With these operations, we can now implemnt the power iteration for eigenvector centrality

```
# initialization
c = np.ones((A.shape[0], 1))

# main loop
maxiter = 10
record_c = np.ones((A.shape[0], maxiter+1))
record_c[:, i] = np.squeeze(c, axis=1)
for i in range(1, maxiter+1):
    c = np.dot(np.transpose(A), c) # c = A^T c
    c = c / np.linalg.norm(c)
    record_c[:,i] = np.squeeze(c, axis=1)
```

```
# results
print('eigenvector centralities: {}'.format(c))

# visualize how the centrality changes
plt.figure()
plt.plot(np.transpose(record_c))
plt.title('Eigenvector Centrality in Each Iteration')
plt.xlabel('Iteration')
plt.ylabel('Eigenvector Centrality')
plt.show()

# visualiz the graph with node size reflecting the centrality
plt.figure(figsize=(6,6))
nx.draw_networkx(G, pos, with_labels=True, node_size=list(c*800), font_size=8, node_c
plt.show()
```

```
eigenvector centralities: [[0.2559327 ]
 [0.27865895]
 [0.14527372]
 [0.06614117]
 [0.25241708]
 [0.32370061]
 [0.14362957]
 [0.1331308 ]
 [0.27217739]
 [0.20269006]
 [0.31362083]
 [0.26157012]
 [0.28656625]
 [0.16840959]
```

Exercise 2: Vectorized Version for Katz Centrality

Following the above example, please implement the vectorized version for Katz Centrality. The vectorized equation for Katz centrality is

$$\mathbf{c} = \alpha \mathbf{A}^T \mathbf{c} + \beta \mathbf{1}$$

where $\mathbf{c}$ is a vector of size N with the i-th element as the Katz centrality of the i-th node in the graph. $\mathbf{1}$ is an all one vector. We have provided the code to get the all one vector below.

```
# TODO
alpha = 0.85
beta = 0.15
all_one_vector = np.ones((A.shape[0], 1))
bias = np.dot(all_one_vector,beta) # get bias

# main loop
maxiter = 10
record_c = np.ones((A.shape[0], maxiter+1))
record_c[:, i] = np.squeeze(c, axis=1)
for i in range(1, maxiter+1):
    c = alpha * np.dot(np.transpose(A), c) + bias # c = A^T c + bias
    c = c / np.linalg.norm(c)
    record_c[:,i] = np.squeeze(c, axis=1)

# results
print('katz centralities: {}'.format(c))

# visualize how the centrality changes
plt.figure()
plt.plot(np.transpose(record_c))
plt.title('Katz Centrality in Each Iteration')
plt.xlabel('Iteration')
plt.ylabel('Katz Centrality')
plt.show()
```

```
# visualiz the graph with node size reflecting the centrality
plt.figure(figsize=(6,6))
nx.draw_networkx(G, pos, with_labels=True, node_size=list(c*800), font_size=8, node_c
plt.show()
```

```
katz centralities: [[0.2586561 ]
```

2.4 Eigendecomposition for eigenvector centrality

Eigenvector centrality is called "eigenvector" becasue the scores are actually the eigenvector of the largest eigenvalues of the adjacency matrix. Next we will show how to use eigendecomposition to get the centrality score

```
# eigen decomposition of A
eigenValues, eigenVectors = np.linalg.eigh(A)

# sort the eigenvalues from largest to smallest
idx = eigenValues.argsort()[::-1]
eigenValues = eigenValues[idx]
eigenVectors = eigenVectors[:,idx]
print(eigenValues)
```

```
[11.94903763  4.71745713  4.00945036  2.34246659  1.29210882  0.84385161
  0.26388917 -0.3047154  -0.56103915 -1.69079368 -2.00184459 -2.53573353
 -3.11296371 -4.60215813 -5.07836522 -5.53064788]
```

```
# The eigenvector centrality is obtained as the eigenvector corresponding to the large
c_eig = np.transpose(np.squeeze(eigenVectors[:,0], axis=1))
print(c_eig)
```

```
[[0.25593012]
 [0.27866195]
 [0.14526482]
 [0.06613518]
 [0.25240925]
 [0.32369758]
 [0.14363115]
 [0.13312674]
 [0.27218106]
 [0.20269293]
 [0.31362702]
 [0.26156113]
 [0.28656748]
 [0.16840979]
 [0.33932951]
 [0.34199123]]
```

Exercise 3: Verify that c_eig and c we get in 2.2 are the same

To verify this, please calculate the l2 distance of c_eig and c, i.e.

$$\|c\_eig - c\|_2 = \sqrt{\sum_i (c(i) - c\_eig(i))^2}$$

Hint: You can do this by calling np.linalg.norm(c - c_eig). Please make sure that c is the eigenvector centrality. You might have reused c for katz centrality.

```
# TODO: Please calcula

# initialization
c = np.ones((A.shape[0], 1))

# main loop
maxiter = 10
record_c = np.ones((A.shape[0], maxiter+1))
record_c[:, i] = np.squeeze(c, axis=1)
for i in range(1, maxiter+1):
    c = np.dot(np.transpose(A), c) # c = A^T c
    c = c / np.linalg.norm(c)
    record_c[:,i] = np.squeeze(c, axis=1)


l2_distance = np.linalg.norm(c - c_eig)

l2_distance
```

```
2.1061563759636258e-05
```

Question: Are c_eig and c the same? Why?

Answer: They are no the same because the l2 distance between them is not 0

3. Link Prediction

In this part, we will implement the Jaccard Similairy and Cossine for Link Prediction

3.1 Basic Operations

```
# get neighbors of node u and v
u = 1 # assume u is node 1
v = 2 # assume v is node 2
u_neighbors = set(G.neighbors(u))
v_neighbors = set(G.neighbors(v))
print(u_neighbors)
print(v_neighbors)
```

```
{2, 3, 4, 5, 6, 12, 15, 16}
{1, 3, 5, 6, 9, 10, 15, 16}
```

```
# union of two sets
unique_friends = u_neighbors.union(v_neighbors)
print(unique_friends)

# intersect of
common_friends = u_neighbors.intersection(v_neighbors)
print(common_friends)
```

```
{1, 2, 3, 4, 5, 6, 9, 10, 12, 15, 16}
{3, 5, 6, 15, 16}
```

```
# number of common_friends
len(common_friends)
print(len(common_friends))

# number of unique_friends
len(unique_friends)
print(len(unique_friends))
```

```
5
11
```

Exercise 4: Jaccard Similarity of Two nodes

The jaccard simialrity of two nodes u and v is defined as

$$jaccard\_simialrity(u, v) = \frac{\left|u\_neighbors \cap v\_neighbors\right|}{\left|u\_neighbors \cup v\_neighbors\right|} = \frac{\text{\# of common friends of u and}}{\text{\# of unique friends of u and v}}$$

With the definiton and above code please finish the following code

```
# TODO: Please calculate the jaccard similarity of node u and v, and return the score
def jaccard_similarity(G, u, v):
    """
    This function calculate the jaccard similarity of two nodes u and v based on the g
    :param G: the networkx graph
    :param u: node
    :param v: node
    :return: a scalar, the jaccard simialrity of node u and v
    """
    if u not in G.nodes or v not in G.nodes:
        raise ValueError
    u_neighbors = set(G.neighbors(u))
    v_neighbors = set(G.neighbors(v))
```

```
    ## plese calculate the jaccard simialrity
    intercetion = u_neighbors & v_neighbors
    union = u_neighbors | v_neighbors


    # return the similarity
    return  len(intercetion)/len(union)
```

```
# call your function to calculate the jaccard similarity of node 1 and 2
# if you see 0.45454545454545453, then you answer is correct
similarity = jaccard_similarity(G, 1, 2)
print(similarity)
if similarity == 0.45454545454545453:
    print('Correct!')
else:
    print('Incorrect')
```

```
0.45454545454545453
Correct!
```

Exercise 5: Cosine

The cosine simialrity of two nodes u and v is defined as

$$jaccard\_simialrity(u, v) = \frac{\left|u\_neighbors \cap v\_neighbors\right|}{\sqrt{|u\_neighbors| \times |v\_neighbors|}} = \frac{\text{\# of common friends of}}{\sqrt{|u\_neighbors| \times |v\_nei}}$$

With the definiton and above code please finish the following code

```
# TODO: Please calculate the cosine similarity of node u and v, and return the score
import math
def cosine_similarity(G, u, v):
    """
    This function calculate the cosine similarity of two nodes u and v based on the gr
    :param G: the networkx graph
    :param u: node
    :param v: node
    :return: a scalar, the cosine simialrity of node u and v
    """
    if u not in G.nodes or v not in G.nodes:
        raise ValueError
    u_neighbors = set(G.neighbors(u))
    v_neighbors = set(G.neighbors(v))
    ## plese calculate the cosine simialrity
    intercetion = u_neighbors & v_neighbors
```

```
    numerator = len(u_neighbors)
    denominator = math.sqrt(len(u_neighbors) * len(v_neighbors))


    # return the cosine similarity
    return numerator / denominator



# call your function to calculate the cosine similarity of node 1 and 2
similarity = cosine_similarity(G, 1, 2)
print(similarity)
if similarity == 0.625:
    print('Correct!')
else:
    print('Incorrect')
```

```
1.0
Incorrect
```

Exercise 6: Link Prediction Example

Use jaccard simialrity to recommend top 2 users to node 1

```
# TODO:
user = 1 # the user we want to suggest friends
# the set of nodes that are not linked with the user
preds = []
for node in G.nodes:
    if node not in G.neighbors(user) and node != user:
        # call your function to calculate the jaccard similarity of user and node
        similarity = jaccard_similarity(G, user, node)
        # store the simialrity to preds
        preds.append((user, node, similarity))


# rank based on the jaccard similarity
ranked = sorted(preds, key=lambda x: x[2], reverse=True)
print(ranked)
```

```
[(1, 7, 0.36363636363636365), (1, 8, 0.36363636363636365), (1, 9, 0.363636363636
```

Question: Which two nodes should we recommende to node 1? Why?

Answer: 7,8,9 because they have the same top similarity

4. Regular Equivalence (Opitional)

We will implement the regular equivalence. The basic idea of the (simplified) regular equivalence is v_i and v_j are simialr if v_j is simialr to v_i's neighor v_k. This can be mathmatically written as

$$\sigma_r(v_i, v_j) = \alpha \sum_{v_k \in \mathcal{N}(v_i)} A_{ik} \cdot \sigma_r(v_k, v_j)$$

where $\sigma_r(v_i, v_j)$ denotes the regular equivalence between v_i and v_j and α is a scalar to make the equality hold. To solve the above equation, we will first vectorize it. Let σ_r be an $n \times n$ matrix with its (i, j)-th element as the regular equivlance between v_i and v_j. Then the above equation can be written as

$$\sigma_r = \alpha \cdot \mathbf{A} \cdot \sigma_r$$

To guarantee that a node is highly similar to itself, we add the identity matrix to the above equation as

$$\sigma_r = \alpha \cdot \mathbf{A} \cdot \sigma_r + \mathbf{I}$$

where $\mathbf{I}$ is an $n \times n$ identity matrix. With the above equation,we can get

$$\sigma_r = (\mathbf{I} - \alpha \cdot \mathbf{A})^{-1}$$

To make sure that the inverse of $\mathbf{I} - \alpha \cdot \mathbf{A}$ exist, we will set $\alpha < \frac{1}{\lambda_{max}}$, where λ_{max} is the largest eigenvalue of $\mathbf{A}$.

```
# obtain adjacency matrix
A = nx.adjacency_matrix(G).todense()
```

```
# eigen decomposition of A
eigenValues, eigenVectors = np.linalg.eigh(A)

# sort the eigenvalues from largest to smallest
idx = eigenValues.argsort()[::-1]
eigenValues = eigenValues[idx]

print(eigenValues)
```

```
1/eigenValues[0]
```

```
# set alpha smaller than 1/lambda_max
alpha= 0.05
P = np.linalg.pinv(np.identity(A.shape[0]) - alpha*A) # (I - alpha* A)^-1
print(P)
plt.matshow(P)
```

Exercise 7 (Optional)

Run experiments $\alpha > \frac{1}{\lambda_{max}}$. Does the results make sense? Why?

```
alpha= 0.1
P = np.linalg.pinv(np.identity(A.shape[0]) - alpha*A) # (I - alpha* A)^-1
print(P)
```

Your Answer Here: ??????????????

Exercise 8 Approximating $(\mathbf{I} - \alpha \cdot \mathbf{A})^{-1}$ (Optional)

In the class, we showed that $(\mathbf{I} - \alpha \cdot \mathbf{A})^{-1} = \sum_{k=0}^{\infty} \alpha^k \mathbf{A}^k$. Thus, we can approximate $(\mathbf{I} - \alpha \cdot \mathbf{A})^{-1}$ as $(\mathbf{I} - \alpha \cdot \mathbf{A})^{-1} = \sum_{k=0}^{K} \alpha^k \mathbf{A}^k$, where K is a large number. Please understand the code below, explore the results of different choices of K and compare with $\mathbf{P}$ calculated directly using $(\mathbf{I} - \alpha \cdot \mathbf{A})^{-1}$

```
Pa = 0
K = 10
alpha = 0.05
tmp = np.eye(A.shape[0])
for i in range(0,K):
  Pa = Pa + tmp
  tmp = tmp * alpha * A
print(Pa)
```

✓ 0s completed at 6:27 PM