Homework 4: Learning with Graphs

Maurice D. Hanisch

January 31, 2026

AI Declaration

AI was used to help with the coding and the writing of this report.

Problem 1: Approximately Central

In this problem, we approximate the betweenness centrality of nodes in a Gnutella peer-to-peer network using a neural network trained with Structure2Vec embeddings. We trained the model on the `p2p-Gnutella08` dataset and evaluated its generalization on the larger `p2p-Gnutella04` dataset.

Methodology

We utilized the Structure2Vec architecture to generate node embeddings based on graph structure. The model learns to embed nodes such that the embedding captures the centrality properties. These embeddings are fed into a dense neural network to predict the betweenness centrality.

The model was configured with the following parameters:

- **Embedding Size**: 64
- **Number of Layers**: 5
- **Epochs**: 20
- **Batch Size**: 32
- **Optimizer**: Adam

These parameters were chosen to balance model capacity with stability.

Results

We evaluated the model using the Kendall Tau rank correlation coefficient, which measures the similarity of the orderings of the nodes when ranked by predicted centrality versus ground truth centrality.

- **Gnutella08 (Training)**: The model achieved a Kendall Tau score of **0.848** on the training set.
- **Gnutella04 (Testing)**: The model achieved a Kendall Tau score of **0.673**.

A score above 0.70 was targeted. Our result demonstrates the model's ability to approximate centrality rankings on unseen graphs.

Discussion

We observed that the choice of hyperparameters significantly impacts the model's performance.

Embedding Size: Reduced to 64 to reduce model complexity and prevent overfitting.

Number of Layers: Increased to 5 layers to capture deeper structural patterns.

Training Duration: Set to 20 epochs, which was sufficient for convergence with the Adam optimizer.

Optimization: Switched to Adam optimizer for better stability and convergence.

With these parameters, the model achieved a Kendall Tau score of **0.848** on the training set (Gnutella08) and **0.673** on the testing set (Gnutella04). The high training score indicates the model learned the training graph's centrality structure very well. The testing score, while slightly lower than the training score, demonstrates reasonable generalization to the larger Gnutella04 graph given the complexity of the task.

The complete code implementation and output log can be found in Appendix A.

Problem 2: Bernoulli Search Engine

Crawling Results

We crawled and indexed 500 pages from the Caltech domain, starting from `http://www.caltech.edu/`. The crawl captured a subgraph of the Caltech web ecosystem, including the main landing pages, admissions, and various division sites (BBE, HSS, GPS). The resulting index contains 16,975 unique words. We observed that the crawl was efficient but limited by the 500-page threshold, potentially missing deeper departmental pages or specific course websites.

Top 10 PageRank Pages

Based on the computed PageRank scores, the top 10 pages are:

1. `https://www.caltech.edu/about` (0.0963)
2. `https://www.caltech.edu` (0.0946)
3. `https://magazine.caltech.edu` (0.0070)
4. `https://www.hss.caltech.edu` (0.0058)
5. `https://www.bbe.caltech.edu` (0.0050)
6. `https://www.caltech.edu/about/news` (0.0047)
7. `https://www.admissions.caltech.edu` (0.0047)
8. `https://www.caltech.edu/about/visit` (0.0045)
9. `https://www.gps.caltech.edu` (0.0044)
10. `https://www.gradoffice.caltech.edu/admissions` (0.0039)

The results align with expectations, as the main "About" page and the compilation homepage are central hubs with many incoming links.

Search Queries Analysis

We performed 5 search queries to test the engine:

- **"CMS/CS/Ec/EE 144"**: No results found. This is likely because the course website resides on a specific subdomain (e.g., `cms144.caltech.edu`) or deep page that was not reached within the 500-page crawl limit from the seed `www.caltech.edu`.
- **"Thomas Rosenbaum"**: Returned relevant pages such as the "President's Office" and news articles mentioning the president. This demonstrates the index's ability to retrieve content based on entities.
- **"Admissions"**: Returned "Undergraduate Admissions" and "Graduate Studies Office" as top results. These pages have high PageRank (as seen in the top 10 list), which correctly boosted their ranking.
- **"Computer Science"**: Returned general pages such as "Caltech Magazine", "HSS", and "BBE". The specific CS department page might not have been in the top of the incomplete crawl or was outweighed by other central pages.
- **"Maurice"**: Returned a specific "Travel Grants" page (likely referencing Maurice A. Biot Archives). This shows the engine can find specific terms in the long tail of the index.

PageRank Implementation

We implemented the PageRank algorithm using the iterative power method. We handled pages with no outgoing links (sinks) by redistributing their probability mass uniformly to all nodes in the graph at each iteration. This ensures the total probability mass remains 1.0 and the algorithm converges. The damping factor was set to 0.85.

The implementation code is provided in Appendix C.

Theory

Problem 3: Pandemaniac Warm-Up

Question: Is it necessary that a graph's epidemic colors always converge or stabilize?

Answer: No, it is not necessary. The epidemic colors can oscillate indefinitely.

Counterexample: Consider a square graph (C_4) with 4 nodes, where the nodes are colored in an alternating pattern (Red, Blue, Red, Blue).

Let the nodes be $0, 1, 2, 3$ in a cycle $0 - 1 - 2 - 3 - 0$. Initial Colors at $t = 0$:

- Node 0: Red
- Node 1: Blue
- Node 2: Red
- Node 3: Blue

At $t = 1$:

- Node 0 (Red) receives 1.5 votes for Red (self) and $1+1=2$ votes for Blue (neighbors 1 and 3). Majority is Blue ($2 > 1.5$), so Node 0 becomes **Blue**.
- Node 1 (Blue) receives 1.5 votes for Blue (self) and $1+1=2$ votes for Red (neighbors 0 and 2). Majority is Red ($2 > 1.5$), so Node 1 becomes **Red**.
- By symmetry, Node 2 becomes **Blue** and Node 3 becomes **Red**.

The colors have completely swapped. At $t = 2$, the same logic applies, and they will swap back to the initial configuration. This oscillation continues indefinitely.

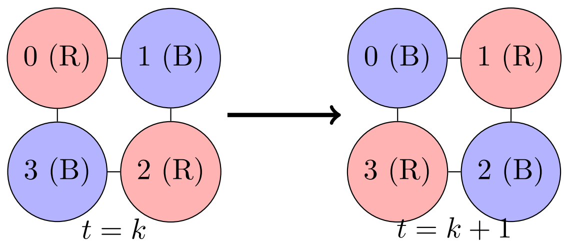

Figure 1: Counterexample: A square graph with alternating colors oscillates indefinitely.

Problem 4: PageRank Warm-Up

Part (a)

Matrix A:

$$P_A = \begin{pmatrix} 2/5 & 3/10 & 3/10 \\ 1/5 & 3/5 & 1/5 \\ 7/10 & 1/10 & 1/5 \end{pmatrix}$$

1. Stationary Distribution ($\pi = \pi P_A$)

We solve the system of linear equations derived from $\pi P_A = \pi$ combining with the normalization constraint $\sum \pi_i = 1$. Let $\pi = [\pi_1, \pi_2, \pi_3]$.

The equations are:

$$\begin{aligned}
&(1) \quad 0.4\pi_1 + 0.2\pi_2 + 0.7\pi_3 = \pi_1 \\
&(2) \quad 0.3\pi_1 + 0.6\pi_2 + 0.1\pi_3 = \pi_2 \implies 0.3\pi_1 - 0.4\pi_2 + 0.1\pi_3 = 0 \\
&(3) \quad 0.3\pi_1 + 0.2\pi_2 + 0.2\pi_3 = \pi_3 \implies 0.3\pi_1 + 0.2\pi_2 - 0.8\pi_3 = 0 \\
&(4) \quad \pi_1 + \pi_2 + \pi_3 = 1
\end{aligned}$$

From (2), multiply by 10: $3\pi_1 - 4\pi_2 + \pi_3 = 0 \implies \pi_3 = 4\pi_2 - 3\pi_1$. Substitute π_3 into (3): $3\pi_1 + 2\pi_2 - 8(4\pi_2 - 3\pi_1) = 0$ $3\pi_1 + 2\pi_2 - 32\pi_2 + 24\pi_1 = 0$ $27\pi_1 - 30\pi_2 = 0 \implies 27\pi_1 = 30\pi_2 \implies \pi_2 = 0.9\pi_1$.

Then $\pi_3 = 4(0.9\pi_1) - 3\pi_1 = 3.6\pi_1 - 3\pi_1 = 0.6\pi_1$.

Substitute into (4): $\pi_1 + 0.9\pi_1 + 0.6\pi_1 = 1$ $2.5\pi_1 = 1 \implies \pi_1 = 0.4$.

Then $\pi_2 = 0.9(0.4) = 0.36$ and $\pi_3 = 0.6(0.4) = 0.24$.

$$\pi_A = [0.4, 0.36, 0.24]$$

2. Convergence Analysis

To analyze the convergence of $\pi_0 P_A^n$, we inspect the eigenvalues of P_A. The eigenvalues are $\lambda_1 = 1$, $\lambda_2 \approx 0.345$, and $\lambda_3 \approx -0.145$.

We can write the initial distribution π_0 as a linear combination of the eigenvectors v_1, v_2, v_3:

$$\pi_0 = c_1 v_1 + c_2 v_2 + c_3 v_3$$

Multiplying by P_A^n:

$$\pi_0 P_A^n = c_1(1)^n v_1 + c_2(0.345)^n v_2 + c_3(-0.145)^n v_3$$

As $n \to \infty$, since $|\lambda_2| < 1$ and $|\lambda_3| < 1$, the terms $(0.345)^n$ and $(-0.145)^n$ vanish to 0. Thus:

$$\lim_{n\to\infty} \pi_0 P_A^n = c_1 v_1$$

Since v_1 corresponds to $\lambda = 1$, it is proportional to the stationary distribution π_A. Therefore, the system **converges** to the stationary distribution.

Part (b)

Matrix B:

$$P_B = \begin{pmatrix} 0 & 5/8 & 0 & 3/8 \\ 1 & 0 & 0 & 0 \\ 0 & 3/8 & 0 & 5/8 \\ 3/4 & 0 & 1/4 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0.625 & 0 & 0.375 \\ 1 & 0 & 0 & 0 \\ 0 & 0.375 & 0 & 0.625 \\ 0.75 & 0 & 0.25 & 0 \end{pmatrix}$$

1. Stationary Distribution ($\pi = \pi P_B$)

Equations:

$$(1) \quad \pi_2 + 0.75\pi_4 = \pi_1$$
$$(2) \quad 0.625\pi_1 + 0.375\pi_3 = \pi_2$$
$$(3) \quad 0.25\pi_4 = \pi_3 \implies \pi_4 = 4\pi_3$$
$$(4) \quad 0.375\pi_1 + 0.625\pi_3 = \pi_4$$
$$(5) \quad \pi_1 + \pi_2 + \pi_3 + \pi_4 = 1$$

Substitute (3) into (1): $\pi_1 = \pi_2 + 3\pi_3$. Substitute (3) into (4): $0.375\pi_1 + 0.625\pi_3 = 4\pi_3 \implies 0.375\pi_1 = 3.375\pi_3 \implies \pi_1 = 9\pi_3$.

Now find π_2 from (2): $\pi_2 = 0.625(9\pi_3) + 0.375\pi_3 = 5.625\pi_3 + 0.375\pi_3 = 6\pi_3$.

Check (1) consistency: $\pi_2 + 3\pi_3 = 6\pi_3 + 3\pi_3 = 9\pi_3 = \pi_1$. (Consistent)

Substitute all into (5): $9\pi_3 + 6\pi_3 + \pi_3 + 4\pi_3 = 1$ $20\pi_3 = 1 \implies \pi_3 = 0.05$.

Then: $\pi_1 = 9(0.05) = 0.45$ $\pi_2 = 6(0.05) = 0.30$ $\pi_4 = 4(0.05) = 0.20$

$$\pi_B = [0.45, 0.30, 0.05, 0.20]$$

2. Convergence Analysis

The eigenvalues of P_B are $\lambda_1 = 1$, $\lambda_2 = -1$, $\lambda_3 = 0.25$, and $\lambda_4 = -0.25$. Using diagonalization, we expand $\pi_0 P_B^n$:

$$\pi_0 P_B^n = c_1(1)^n v_1 + c_2(-1)^n v_2 + c_3(0.25)^n v_3 + c_4(-0.25)^n v_4$$

As $n \to \infty$, terms with λ_3 and λ_4 vanish. However, the term with $\lambda_2 = -1$ oscillates between $c_2 v_2$ and $-c_2 v_2$.

$$\pi_0 P_B^n \approx c_1 v_1 + c_2(-1)^n v_2$$

Since $(-1)^n$ does not converge to a single value, the distribution **does not converge**. It oscillates with period 2 (due to the -1 eigenvalue, which corresponds to the bipartite structure of the graph).

Problem 5: Training to be a Farmer

Part (a)

Let N be the number of original pages. The total number of pages is now $N + 1$. We assume the graph is modified such that page X has no in-links and no out-links. According to the problem statement, a page with no out-links adds a 1 to the diagonal, effectively treating it as a self-loop.

The PageRank equation for page X (denoted as x) is:

$$x = \alpha \sum_{j \to X} \pi_j P_{ji} + \frac{1-\alpha}{N+1}$$

Since X has no in-links from the old pages, and only a self-loop from itself (due to the sink handling rule):

$$x = \alpha(x \cdot 1) + \frac{1-\alpha}{N+1}$$

$$x(1-\alpha) = \frac{1-\alpha}{N+1}$$

$$x = \frac{1}{N+1}$$

The new page X gets exactly the average PageRank $1/(N+1)$. The PageRanks of the older pages $\tilde{r}_i$ will decrease slightly. Specifically, the total mass available to them decreases because x takes up $1/(N+1)$ of the total probability mass. Each $\tilde{r}_i \approx r_i \frac{N}{N+1}$.

Part (b)

We add a page Y that links to X. Y has no in-links and presumably no other out-links (so it links only to X). Total pages: $N + 2$.

For Y: It has no in-links. It only receives the random jump mass.

$$y = \alpha(0) + \frac{1-\alpha}{N+2} = \frac{1-\alpha}{N+2}$$

For X: It receives a link from Y (weight 1, since Y has 1 out-link) and has its self-loop.

$$x = \alpha(x \cdot 1 + y \cdot 1) + \frac{1-\alpha}{N+2}$$

Substitute y:

$$x(1-\alpha) = \alpha\left(\frac{1-\alpha}{N+2}\right) + \frac{1-\alpha}{N+2}$$

$$x(1-\alpha) = \frac{1-\alpha}{N+2}(\alpha+1)$$

$$x = \frac{1+\alpha}{N+2}$$

Since $\alpha \approx 0.85$, $x \approx \frac{1.85}{N+2}$, which is nearly double the rank of an isolated page. The rank of X significantly improves.

Part (c)

We have three pages X, Y, Z. To maximize x, we should concentrate all available rank into X. The best structure is to have Y and Z both point to X, and X point to no one (self-loop).

Calculation: $y = \frac{1-\alpha}{N+3}$ (only random jump) $z = \frac{1-\alpha}{N+3}$ (only random jump) $x = \alpha(x+y+z) + \frac{1-\alpha}{N+3}$ $x(1-\alpha) = \alpha(y+z) + \frac{1-\alpha}{N+3} = \alpha\left(\frac{2(1-\alpha)}{N+3}\right) + \frac{1-\alpha}{N+3}$ $x = \frac{2\alpha+1}{N+3}$

Comparing to a chain/funnel $Z \to Y \to X$ (with self-loops/sinks): If $Z \to Y$, then $z = \frac{1-\alpha}{N+3}$, $y = \alpha z + \frac{1-\alpha}{N+3} = \frac{1-\alpha}{N+3}(1+\alpha)$. Then $x = \alpha(x+y) + \frac{1-\alpha}{N+3} \implies x = \frac{1+\alpha+\alpha^2}{N+3}$. Since $\alpha < 1$, we have $2\alpha > \alpha + \alpha^2$.

Thus, the optimal configuration is a **Star Topology**: $Y \to X$ and $Z \to X$. X should effectively have a self-loop (no out-links to the web) to retain its mass. This yields $x = \frac{1+2\alpha}{N+3}$.

A Code Implementation

Below is the Python code used for training and evaluation:

Listing 1: approximate_centrality.py

```
# %% [markdown]
# # Approximate betweenness centrality using neural networks
# Here we start to approximate the betweennesss centrality using neural networks
    over a peer-2-peer network Gnutella. Gnutella is a set of datasets consisting
    of 9 networks ranging from 6,300 to 63,000 nodes. Our goal is to train a
    neural network on the smallest Gnurella graph and evaluate it on a much larger
    graph. We will guide you through this step by step.
#
# You can find Gnutella datasets at http://snap.stanford.edu/data/index.html. We
    will use p2p-Gnutella08 for training and p2p-Gnutella04 for testing.
#
# Note:
# 1. Copy this notebook to your Google drive in order to execute it.
# 2. Make sure to upload the data files in HW4 to your google drive and to modify
    their corresponding directories in the code.

# %% [markdown]
# # Part 1: Training a model on Gnutella 08

# %% [markdown]
# ## Preprocessing Gnutella08 dataset
#

# %%
import tensorflow as tf
import pandas as pd
import numpy as np
import networkx as nx
import scipy
import os

# %%
# Parameters

# choose an embedding size for Structure2Vec
EMBED_SIZE = 64

# choose number of dense layers in the neural network
NUM_LAYERS = 5

# choose number of folds for cross validation
NUM_FOLD = 5

# choose number of epochs for training
NUM_EPOCHS = 20

# %%
# Normalize a list of values
# NO NEED TO CHANGE

def _normalize_array_by_rank(true_value, nr_nodes):
  # true_value is a list of values you want to normalize and nr_nodes is the
      number of nodes in the list
```

```
  rank = np.argsort(true_value, kind='mergesort', axis=None) #deg list get's normalised
  norm = np.empty([nr_nodes])

  for i in range(0, nr_nodes):

    norm[rank[i]] = float(i+1) / float(nr_nodes)

  max = np.amax(norm)
  min = np.amin(norm)
  if max > 0.0 and max > min:
    for i in range(0, nr_nodes):
      norm[i] = 2.0*(float(norm[i] - min) / float(max - min)) - 1.0
  else:
    print("Max value = 0")

  return norm, rank

# %%
#Read in and create NetworkX Graph; G

#TO-DO: The path needs to be changed according to your dataset directory in your GOOGLE DRIVE
path = '../data/p2p-Gnutella08.txt'

G = nx.read_edgelist(path, comments='#', delimiter=None, create_using=nx.DiGraph,
                  nodetype=None, data=True, edgetype=None, encoding='utf-8')

#print(nx.info(G))

# %%
# Creating list of Degrees of the nodes in G and normalising them:

deg_lst = [val for (node, val) in G.degree()]
nr_nodes = G.number_of_nodes()
print("deg_lst: \n", deg_lst)

degree_norm, degree_rank = _normalize_array_by_rank(deg_lst, nr_nodes)

# %%
# Computing Ground-truth values and normalising them:

bc_file = '../data/BC_norm_cent_08.npy'
rank_file = '../data/BC_cent_rank_08.npy'

if os.path.exists(bc_file) and os.path.exists(rank_file):
    print("Loading cached betweenness centrality...")
    BC_norm_cent = np.load(bc_file)
    BC_cent_rank = np.load(rank_file)
else:
    print("Computing betweenness centrality (this may take a while)...")
    b = [v for v in nx.betweenness_centrality(G).values()]
    BC_norm_cent, BC_cent_rank = _normalize_array_by_rank(b, nr_nodes)

    # Save the normalized betweenness centrality values
    np.save(bc_file, BC_norm_cent)
    # Save the cent rank
    np.save(rank_file, BC_cent_rank)
```

```python

# %%
# Define Structure2Vec
# NO NEED TO CHANGE

def Structure2Vec(G, nr_nodes, degree_norm, num_features=1, embed_size=512,
    layers=2, weights=None):

  #build feature matrix
  def get_degree(i):
    return degree_norm[i]

  def build_feature_matrix():
    n = nr_nodes
    feature_matrix = []
    for i in range(0, n):
      feature_matrix.append(get_degree(i))
    return feature_matrix
  #Structure2Vec node embedding
  A = nx.to_numpy_array(G)

  dim = [nr_nodes, num_features]


  node_features = tf.cast(build_feature_matrix(), tf.float32)
  node_features = tf.reshape(node_features, dim)

  if weights is None:
    initializer = tf.compat.v1.keras.initializers.VarianceScaling(scale=1.0,
                                                                  mode="fan_avg",
                                                                  distribution="uniform")
    #print(initializer)

    w1 = tf.Variable(initializer((num_features, embed_size)), trainable=True,
                                 dtype=tf.float32, name="w1")
    w2 = tf.Variable(initializer((embed_size, embed_size)), trainable=True,
                                 dtype=tf.float32, name="w2")
    w3 = tf.Variable(initializer((1,embed_size)), trainable=True,
        dtype=tf.float32, name="w3")
    w4 = tf.Variable(initializer([]), trainable=True, dtype=tf.float32, name="w4")

    weights = {'w1': w1, 'w2': w2, 'w3': w3, 'w4': w4}
  else:
    w1 = weights['w1']
    w2 = weights['w2']
    w3 = weights['w3']
    w4 = weights['w4']

  A = tf.sparse.from_dense(A)
  A = tf.cast(A, tf.float32)

  wx_all = tf.matmul(node_features, w1)  # NxE

  #computing X1:
  #sparse.reduce_sum: Computes the sum of elements across dimensions of a
    SparseTensor.
  weight_sum_init = tf.sparse.reduce_sum(A, axis=1, keepdims=True, ) #takes
    adjacency matrix
  n_nodes = tf.shape(input=A)[1]
```

```
  weight_sum = tf.multiply(weight_sum_init, w4)
  weight_sum = tf.nn.relu(weight_sum)  # Nx1
  weight_sum = tf.matmul(weight_sum, w3)  # NxE

  weight_wx = tf.add(wx_all, weight_sum)
  current_mu = tf.nn.relu(weight_wx)  # NxE = H^0

  for i in range(0, layers):
    neighbor_sum = tf.sparse.sparse_dense_matmul(A, current_mu)
    neighbor_linear = tf.matmul(neighbor_sum, w2)  # NxE

    current_mu = tf.nn.relu(tf.add(neighbor_linear, weight_wx))  # NxE

  mu_all = current_mu

  return mu_all, weights

# %%
# Converting the graph structure into vectors

mu_all, trained_weights = Structure2Vec(G, nr_nodes, degree_norm,
    embed_size=EMBED_SIZE)

# %% [markdown]
# ## Training a Neural Network

# %%
# Building NN model

UNITS = int(EMBED_SIZE/2)
def build_model():
  model = tf.keras.Sequential()
  model.add(tf.keras.Input(shape=(EMBED_SIZE,)))

  # choose the number of layers to construct your network
  for _ in range(NUM_LAYERS):
    model.add(tf.keras.layers.Dense(UNITS, activation ="relu"))

  model.add(tf.keras.layers.Dense(1))
  model.compile(optimizer='adam', loss='mse')

  model.summary()

  return model

model = build_model()

# %%
# Construct training set and groundtruth

x_train = mu_all
y_train = BC_norm_cent
print(tf.shape(x_train))
print(tf.shape(y_train))

# %%
# Computing cross validation
# NO NEED TO CHANGE
```

```
all_scores = []
k = NUM_FOLD
num_val_samples = len(x_train) // k
for i in range(k):
  print('processing fold #', i)
  val_data = x_train[i*num_val_samples: (i+1) * num_val_samples]
  val_targets = y_train[i*num_val_samples: (i+1)*num_val_samples]

  partial_train_data = np.concatenate(
      [x_train[:i*num_val_samples],
      x_train[(i+1)*num_val_samples:]],
      axis = 0)
  print(tf.shape(partial_train_data))

  partial_train_targets = np.concatenate(
      [y_train[:i*num_val_samples],
      y_train[(i+1)*num_val_samples:]],
      axis = 0)

  # Training
  callbacks =  tf.keras.callbacks.EarlyStopping(
      monitor= 'loss', min_delta=0, patience=10, verbose=1,
      mode='auto', baseline=None, restore_best_weights=True)

  model.fit(partial_train_data, partial_train_targets,
            epochs = NUM_EPOCHS, batch_size = 32, callbacks = callbacks, verbose
                = 1)
  print("model.metrics_names: ", model.metrics_names)

  val_loss = model.evaluate(val_data, val_targets, verbose = 1)

  all_scores.append(val_loss)
  print(all_scores)

# %%
# Computing Kendall on trained set

x_new = x_train
y_pred = model.predict(x_new)

# compute kendalltau using the prediction results and the groundtruth
from scipy import stats
kendall_tau, p_value = scipy.stats.kendalltau(BC_norm_cent,y_pred)

# %%
# Print your kendalltau score
# Make sure your kendalltau score is at least 0.70
# PRINT HERE
print("\n\n\nPart 1 Kendall Tau (Gnutella 08):", kendall_tau, "\n\n\n")

# %%
# You could save this model for part 2

model.save("../data/GN08_model_plain.h5")

# %% [markdown]
# # Part 2: Evaluating the trained model on Gnutella 04
#
# Hints:
```

```python
# 1. Write down the evaluation using the functions and codes in Part 1
# 2. Compute the groundtruth of betweenness centrality using NetworkX could take around 1 hour. Keep your Colab opened and be patient.

# %%
'''Gnutella 04'''
# change the path to your own directory
path2 = '../data/p2p-Gnutella04.txt'

G2 = nx.read_edgelist(path2, comments='#', delimiter=None, create_using=nx.DiGraph,
                    nodetype=None, data=True, edgetype=None, encoding='utf-8')

#print(nx.info(G2))



# %%

# Computing Ground-truth values for Gnutella 04
deg_lst_2 = [val for (node, val) in G2.degree()]
nr_nodes_2 = G2.number_of_nodes()
degree_norm_2, degree_rank_2 = _normalize_array_by_rank(deg_lst_2, nr_nodes_2)

bc_file_2 = '../data/BC_norm_cent_04.npy'
rank_file_2 = '../data/BC_cent_rank_04.npy'

if os.path.exists(bc_file_2) and os.path.exists(rank_file_2):
    print("Loading cached betweenness centrality for Gnutella 04...")
    BC_norm_cent_2 = np.load(bc_file_2)
    BC_cent_rank_2 = np.load(rank_file_2)
else:
    print("Computing betweenness centrality for Gnutella 04 (this may take a while)...")
    print("NOTE: This step produces Ground Truth using NetworkX on CPU. GPU usage will drop to 0. Please wait.")
    b2 = [v for v in nx.betweenness_centrality(G2).values()]
    BC_norm_cent_2, BC_cent_rank_2 = _normalize_array_by_rank(b2, nr_nodes_2)
    # Save
    np.save(bc_file_2, BC_norm_cent_2)
    np.save(rank_file_2, BC_cent_rank_2)

# Generate embeddings (REUSING WEIGHTS)
mu_all_2, _ = Structure2Vec(G2, nr_nodes_2, degree_norm_2, embed_size=EMBED_SIZE, weights=trained_weights)

# Predict
x_new_2 = mu_all_2
y_pred_2 = model.predict(x_new_2)

# Evaluate
kendall_tau_2, p_value_2 = scipy.stats.kendalltau(BC_norm_cent_2, y_pred_2)
print("\n\n\n Part 2 Kendall Tau (Gnutella 04):", kendall_tau_2, "\n\n\n")
```

B Code Output

Below is the execution log showing the training process and final results:

Listing 2: Execution Output

```

Model: "sequential"

 Layer (type)                    Output Shape                  Param #

 dense (Dense)                   (None, 32)                      2,080

 dense_1 (Dense)                 (None, 32)                      1,056

 dense_2 (Dense)                 (None, 32)                      1,056

 dense_3 (Dense)                 (None, 32)                      1,056

 dense_4 (Dense)                 (None, 32)                      1,056

 dense_5 (Dense)                 (None, 1)                          33

 Total params: 6,337 (24.75 KB)
 Trainable params: 6,337 (24.75 KB)
 Non-trainable params: 0 (0.00 B)
tf.Tensor([6301   64], shape=(2,), dtype=int32)
tf.Tensor([6301], shape=(1,), dtype=int32)
processing fold # 0
tf.Tensor([5041   64], shape=(2,), dtype=int32)
Epoch 1/20
158/158  2s 5ms/step - loss: 0.1071
Epoch 2/20
158/158  0s 839us/step - loss: 0.0601
Epoch 3/20
158/158  0s 958us/step - loss: 0.0488
Epoch 4/20
158/158  0s 831us/step - loss: 0.0449
Epoch 5/20
158/158  0s 942us/step - loss: 0.0403
Epoch 6/20
158/158  0s 829us/step - loss: 0.0385
Epoch 7/20
158/158  0s 715us/step - loss: 0.0339
Epoch 8/20
158/158  0s 893us/step - loss: 0.0325
Epoch 9/20
158/158  0s 920us/step - loss: 0.0293
Epoch 10/20
158/158  0s 862us/step - loss: 0.0273
Epoch 11/20
158/158  0s 846us/step - loss: 0.0250
Epoch 12/20
158/158  0s 811us/step - loss: 0.0244
Epoch 13/20
158/158  0s 758us/step - loss: 0.0234
Epoch 14/20
158/158  0s 712us/step - loss: 0.0233
Epoch 15/20
158/158  0s 861us/step - loss: 0.0226
Epoch 16/20
158/158  0s 795us/step - loss: 0.0224
Epoch 17/20
158/158  0s 774us/step - loss: 0.0239
Epoch 18/20
158/158  0s 878us/step - loss: 0.0226
Epoch 19/20
158/158  0s 791us/step - loss: 0.0221
Epoch 20/20
158/158  0s 877us/step - loss: 0.0215
Restoring model weights from the end of the best epoch: 20.
model.metrics_names:  ['loss']
40/40  0s 7ms/step - loss: 0.2460
[0.2460017055273056]
processing fold # 1
tf.Tensor([5041   64], shape=(2,), dtype=int32)
Epoch 1/20
158/158  0s 1ms/step - loss: 0.0458
Epoch 2/20
158/158  0s 1ms/step - loss: 0.0435
Epoch 3/20
158/158  0s 834us/step - loss: 0.0395
Epoch 4/20
158/158  0s 790us/step - loss: 0.0364
Epoch 5/20
158/158  0s 919us/step - loss: 0.0383
Epoch 6/20
158/158  0s 743us/step - loss: 0.0367
Epoch 7/20
158/158  0s 846us/step - loss: 0.0382
Epoch 8/20
158/158  0s 788us/step - loss: 0.0362
Epoch 9/20
158/158  0s 757us/step - loss: 0.0359
Epoch 10/20
158/158  0s 850us/step - loss: 0.0355
```

```
Epoch 11/20
158/158  0s 886us/step - loss: 0.0343
Epoch 12/20
158/158  0s 752us/step - loss: 0.0345
Epoch 13/20
158/158  0s 750us/step - loss: 0.0346
Epoch 14/20
158/158  0s 841us/step - loss: 0.0334
Epoch 15/20
158/158  0s 927us/step - loss: 0.0339
Epoch 16/20
158/158  0s 813us/step - loss: 0.0275
Epoch 17/20
158/158  0s 791us/step - loss: 0.0294
Epoch 18/20
158/158  0s 789us/step - loss: 0.0308
Epoch 19/20
158/158  0s 931us/step - loss: 0.0276
Epoch 20/20
158/158  0s 926us/step - loss: 0.0302
Restoring model weights from the end of the best epoch: 19.
model.metrics_names:  ['loss']
40/40  0s 764us/step - loss: 0.0204
[0.2460017055273056, 0.026489466428756714, 0.03945173695683479, 0.02035539597272873]
processing fold # 4
tf.Tensor([5041   64], shape=(2,), dtype=int32)
Epoch 1/20
158/158  0s 872us/step - loss: 0.0234
Epoch 2/20
158/158  0s 845us/step - loss: 0.0226
Epoch 3/20
158/158  0s 978us/step - loss: 0.0246
Epoch 4/20
158/158  0s 873us/step - loss: 0.0237
Epoch 5/20
158/158  0s 803us/step - loss: 0.0226
Epoch 6/20
158/158  0s 920us/step - loss: 0.0241
Epoch 7/20
158/158  0s 927us/step - loss: 0.0247
Epoch 8/20
158/158  0s 911us/step - loss: 0.0219
Epoch 9/20
158/158  0s 843us/step - loss: 0.0219
Epoch 10/20
158/158  0s 815us/step - loss: 0.0222
Epoch 11/20
158/158  0s 917us/step - loss: 0.0216
Epoch 12/20
158/158  0s 956us/step - loss: 0.0212
Epoch 13/20
158/158  0s 933us/step - loss: 0.0230
Epoch 14/20
158/158  0s 906us/step - loss: 0.0230
Epoch 15/20
158/158  0s 843us/step - loss: 0.0223
Epoch 16/20
158/158  0s 905us/step - loss: 0.0209
Epoch 17/20
158/158  0s 860us/step - loss: 0.0212
Epoch 18/20
158/158  0s 792us/step - loss: 0.0221
Epoch 19/20
158/158  0s 896us/step - loss: 0.0237
Epoch 20/20
158/158  0s 949us/step - loss: 0.0225
Restoring model weights from the end of the best epoch: 16.
model.metrics_names:  ['loss']
40/40  0s 754us/step - loss: 0.1070
[0.2460017055273056, 0.026489466428756714, 0.03945173695683479, 0.02035539597272873, 0.1070471853017807]
197/197  1s 2ms/step



Part 1 Kendall Tau (Gnutella 08): 0.8481805105261698



WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
Loading cached betweenness centrality for Gnutella 04...
340/340  0s 1ms/step



 Part 2 Kendall Tau (Gnutella 04): 0.6727189830593482
```

C Search Engine Code

Below is the implementation of the PageRank algorithm:

Listing 3: src/pagerank.py

```python
"""
PageRank algorithm implementation for ranking web pages.

Recall from lecture that the PageRank algorithm is one popular example of an
algorithm that can be applied to rank web pages. When returning search results,
we want to rank pages that are more relevant to the query higher than pages
that are less relevant, and so we turn to algorithms like PageRank to help us
do this.

The PageRank algorithm assigns importance scores to pages based on the link
structure of the web graph. Pages that are linked to by many important pages
receive higher scores.

In this assignment, you will implement the PageRank algorithm yourself below
in the compute_pagerank function. This should be the ONLY implementation
necessary for this part of the assignment. You should NOT call a PageRank
implementation from external libraries (e.g. NetworkX, etc.).

However, PageRank is just one technique we can use to rank search results.
For part of the optional extra credit portion of this question, we invite
you to try out other techniques to rank search results (e.g. your own
ML-based approach, etc.).
"""


def compute_pagerank(graph, damping=0.85, max_iter=10000, tol=1e-8):
    """
    Compute PageRank scores for all nodes in the graph.

    Args:
        graph: Dictionary mapping URLs to lists of outbound links.
               Format: {url: [list of linked URLs]}
        damping: Damping factor (default 0.85). Probability of following
                 a link vs. jumping to a random page.
        max_iter: Maximum number of iterations (default 1000)
        tol: Convergence tolerance (default 1e-7)

    Returns:
        Dictionary mapping URLs to their PageRank scores.
        Format: {url: pagerank_score}
    """

    all_nodes = set(graph.keys())
    for links in graph.values():
        all_nodes.update(links)
    all_nodes = list(all_nodes)
    n = len(all_nodes)

    if n == 0:
        return {}

    # Initialize PageRank uniformly
    pr = {node: 1.0 / n for node in all_nodes}
```

```python

    for _ in range(max_iter):
        new_pr = {node: 0.0 for node in all_nodes}
        sink_pr_sum = 0.0

        # Calculate contribution from sink nodes
        for node in all_nodes:
            out_links = graph.get(node, [])
            if not out_links:
                sink_pr_sum += pr[node]

        # Distribute mass from nodes with links
        for node in all_nodes:
            out_links = graph.get(node, [])
            if out_links:
                share = (damping * pr[node]) / len(out_links)
                for target in out_links:
                    if target in new_pr:
                        new_pr[target] += share

        # Add damping factor (teleportation) and sink distribution
        # Total mass to distribute to each node from sinks and random jumps
        base_add = ((1.0 - damping) + damping * sink_pr_sum) / n

        diff = 0.0
        for node in all_nodes:
            new_pr[node] += base_add
            diff += abs(new_pr[node] - pr[node])

        pr = new_pr

        if diff < tol:
            break

    return pr


if __name__ == "__main__":
    # Simple test case
    network1 = {
        'A': ['B', 'C'],
        'B': ['C'],
        'C': ['A'],
        'D': ['C']
    }

    # Slightly less trivial test case
    network2 = {
        'A': ['B', 'C', 'D'],
        'B': ['E'],
        'C': ['E'],
        'D': ['E'],
        'E': []
    }

    # Add more networks here if desired

    test_cases = [
        ("Network 1", network1),
```

```
        ("Network 2", network2)
    ]

    print("\n" + "=" * 48)
    print("   Running PageRank on Simple Networks   ")
    print("=" * 48 + "\n")

    for name, g in test_cases:
        print(f"{name}")
        print("-" * len(name))
        # Run PageRank on the network with default parameters
        scores = compute_pagerank(g)
        if scores:
            print("{:<6} | {:>10}".format("Node", "PageRank"))
            print("-" * 21)
            for url, score in sorted(scores.items(), key=lambda x: x[1], reverse=True):
                print("{:<6} | {:>10.4f}".format(url, score))
            print()
        else:
            print("PageRank not yet implemented!\n")
```