Data Science Task

Name: Pallav Gupta
Registration Number: 21BAI1169
College: Vellore Institute of Technology, Chennai

What is Knowledge Graph?

Knowledge graphs have become influential tools in arranging and portraying information interconnected and systematically. They offer a more profound comprehension of intricate areas by documenting the connections and context among various entities. Knowledge graphs offer a comprehensive perspective on data by depicting knowledge in a graph form, where entities act as nodes and their interconnections as edges. This aids in seamless data amalgamation, the unveiling of new knowledge, and sophisticated analysis. The knowledge graph comprises three main components:

1. Nodes (the points of connection via dots)
2. Edges (lines connecting the dots, representing relationships)
3. Labels (These are the labels that are provided for the relationship)

Example:

Human ---Drinks---> Coffee
Coffee ---Prevents---> Sleep
Human ---Needs---> Sleep

Here we have nodes for humans, Coffee, and sleep. We have three edges We have 3 labels for 3 edges referenced as Drinks, Prevents, and needs.

References: https://www.ibm.com/topics/knowledge-graph -> What is the Knowledge Graph by IBM?

What are GNN

Graph Neural Networks (GNNs) have risen as groundbreaking computational models for processing information directly on graphs. By operating on a structure where data points (nodes) and their interactions (edges) form the foundation, GNNs can learn and capture intricate patterns within relational data. GNNs have found applications in various fields, enabling superior data interpretation, pattern recognition, and predictive analytics in domains where traditional neural networks might fall short.

A graph consists of nodes or vertices, and connections between these nodes are called edges. The information about these connections in a graph can be represented in an adjacency matrix. Nodes and edges can have further properties called node features and edge features. The structure of graphs is non-Euclidean, making distance metrics such as Euclidean distance not clearly defined.

Machine learning can perform node-level predictions, link predictions, or use the whole graph as input for classification or attribute prediction. Graph Neural Networks (GNNs) extend classical feed-forward models to handle the difficulties of graph data, such as changing size and shape within a dataset. GNNs use all information about the graph, including node features and connections stored in an adjacency matrix, to output new representations or embeddings for each node.

The major difference between CNN and GNN is that CNN uses a sliding window algorithm with a stride of k whereas GNN convolves on the features of neighboring nodes and creates node embeddings.

Message passing is the sharing mechanism used to update node features or states. Node embeddings contain feature-based and structural information about the nodes. Message passing is done by all nodes to update their embeddings. The size of the new embeddings is a hyperparameter that depends on the graph data used. The number of message-passing steps corresponds to the number of layers in the GNN. The number of layers in a GNN defines how many neighborhood hops are performed and is a hyperparameter that depends on the graph data and learning task. Over-smoothing can occur if too many message-passing layers are used, but methods such as Paranorm can handle this.

$$h_u^{(k+1)} = UPDATE^{(k)}\left(h_u^{(k)}, AGGREGATE^{(k)}(\{h_v^{(k)}, \forall\, v \in \mathcal{N}(u)\})\right)$$

The update operation combines the current state with neighbor states and different variants of message-passing graph neural networks use different methods for update and aggregate functions.

References: https://www.youtube.com/watch?v=fOctJB4kVlM -> Understanding GNN by DeepFindr

Task: Providing Authors with Co-author Suggestions

Link to dataset: https://bit.ly/BalkanID_DSTask_dataset. The provided file is a dump file from a Neo4j Database (A Graph Database). This dump file was created with Neo4j version 5.3.0. The Nodes in the graph represent authors, and the edge (undirected) shows that the two authors it connects have already co-authored a publication together. Each node or author has 224 features associated with it, which have been anonymized.

Neo4j is nothing but a graph Database Management System. The query to retrieve data is called cypher.

Here, I have discussed how to extract data.

Make sure to install the required libraries in requirement.txt, which include neo4j and other libraries that will support the database.

pip install -r requirements.txt

import the graph database from Neo4j, and then we initiate a connection with the created file using the driver function, and then we can begin our session.

MATCH (n) RETURN n.author_id as author_id,n.data as data

This query returned us a dictionary with the column author_id, data, and we can create a data frame out of it. There are Features 1 to 224 we are going to extract and push into the same data frame using

df[<column_name>]=[<list of all data>]

we define a feature extraction function that extracts each feature using string formatting.

```
def data_feature_creation(num):
  global session, df
```

```
  result = session.run("""MATCH (n) RETURN n.Feature%s"""%str(num) +""" as
Feature%s"""%str(num))
  data = result.data()
  lis=[]
  for i in range(df.shape[0]):
    lis.append(int(data[i]['Feature'+str(num)]))
  df['Feature'+str(num)] = lis
```

The cypher used in the above function is MATCH (n) RETURN n.Feature%s"""%str(num) +""" as Feature%s"""%str(num). This Extracts Feature is based on numbers 1 to 224.

It is to be noted as data was given as a Null column we can use it at our discretion as no specific functionality is assigned for it.

```
# Now the data is going to be the count of authors the author has worked with
def data_attribute_creation(check):
    global session,df
    result = session.run('''
    MATCH (author1)-[:CO_AUTHORED]->(author2)
    where author1.author_id="%s"'''%str(check)+''' and author2.author_id<>"%s"
    RETURN author2.author_id'''%str(check))
    data = result.data()
    return (len(data))
```

We sorted the dataset based on the data column and that is the point where we extracted all the data. The cypher used here was MATCH (author1)-[:CO_AUTHORED]->(author2) where author1.author_id="%s"'''%str(check)+''' and author2.author_id<>"%s" RETURN author2.author_id'''%str(check)) here check gives us the value of author_id we are counting the number of ccnnections for.

the data can be found here

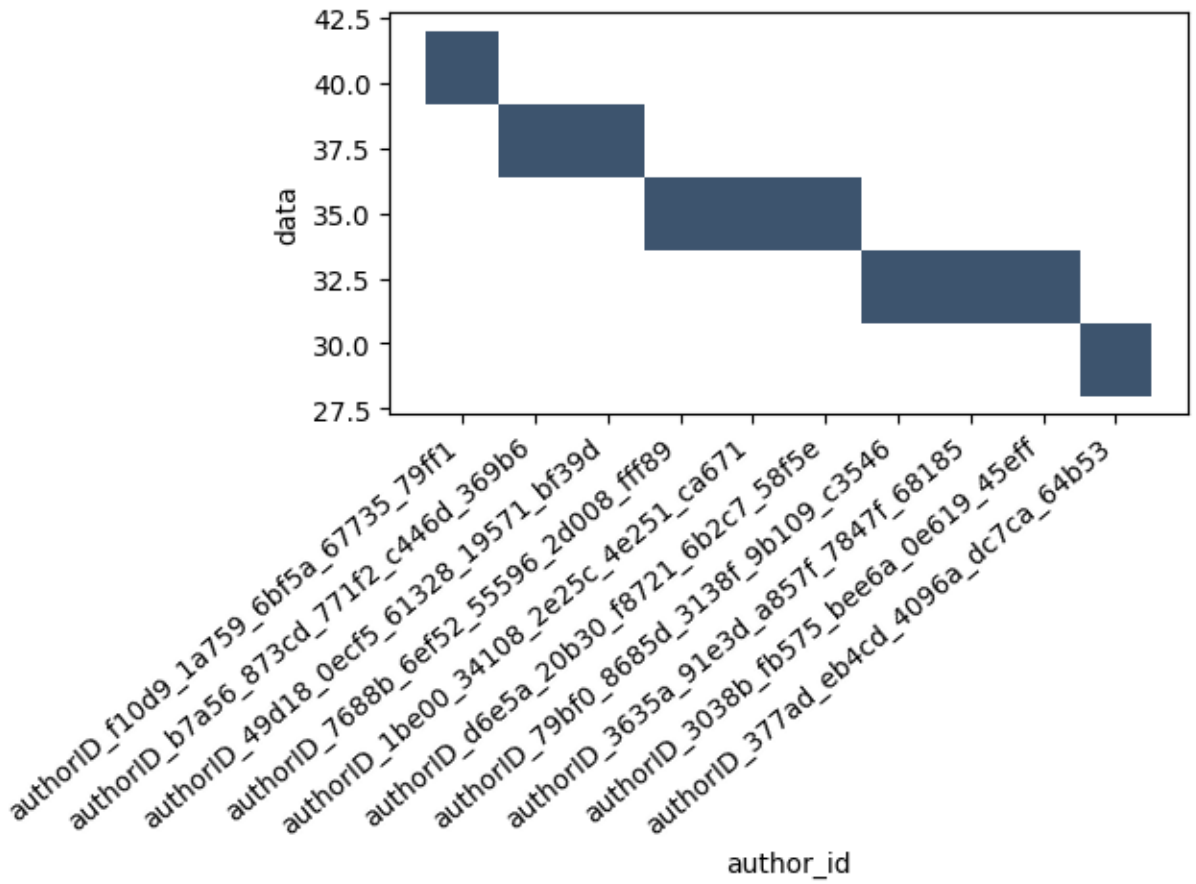
Apart from this I also created another file where I recorded all the relationships the file is here

The cypher used here was similar to that of the data MATCH (author1)-[:CO_AUTHORED]->(author2) where author1.author_id="%s"'''%str(i)+''' and author2.author_id<>"%s" RETURN author1.author_id,author2.author_id'''%str(i)) here, "I" represents the author_id.

Part 1: Unveiling Patterns

This was my first time putting my hands on a Graph database. I looked up for various techniques with which I can derive various results.

My First approach was that of a Histogram showing top 10 authors that are most likely to contribute based on their collaboration. This was derieved from data of 'data' column. The plot is provided below.



My next approach was that of a community detection using Louvian Algorithm on a networkx graph. The code is designed to detect communities among authors based on their co-authored relationships.

```
def community_check():
  lis=[]
  for i in df['author_id']:
    result = session.run('''
    MATCH (author1)-[:CO_AUTHORED]->(author2)
    where author1.author_id="%s"'''%str(i)+''' and author2.author_id<>"%s"
    RETURN author1.author_id,author2.author_id'''%str(i))
    data = result.data()
    for i in data:
      lis.append((i['author1.author_id'],i['author2.author_id']))
      lis.append((i['author2.author_id'],i['author1.author_id']))
  return lis
```

The function above uses cypher query of MATCH (author1)-[:CO_AUTHORED]->(author2) where author1.author_id="%s"'''%str(i)+''' and author2.author_id<>"%s" RETURN author1.author_id,author2.author_id'''%str(i)) where str(i) represents the author_id for which we are looking for collaborated authors.

Now We create a networkx Graph and run our community detection

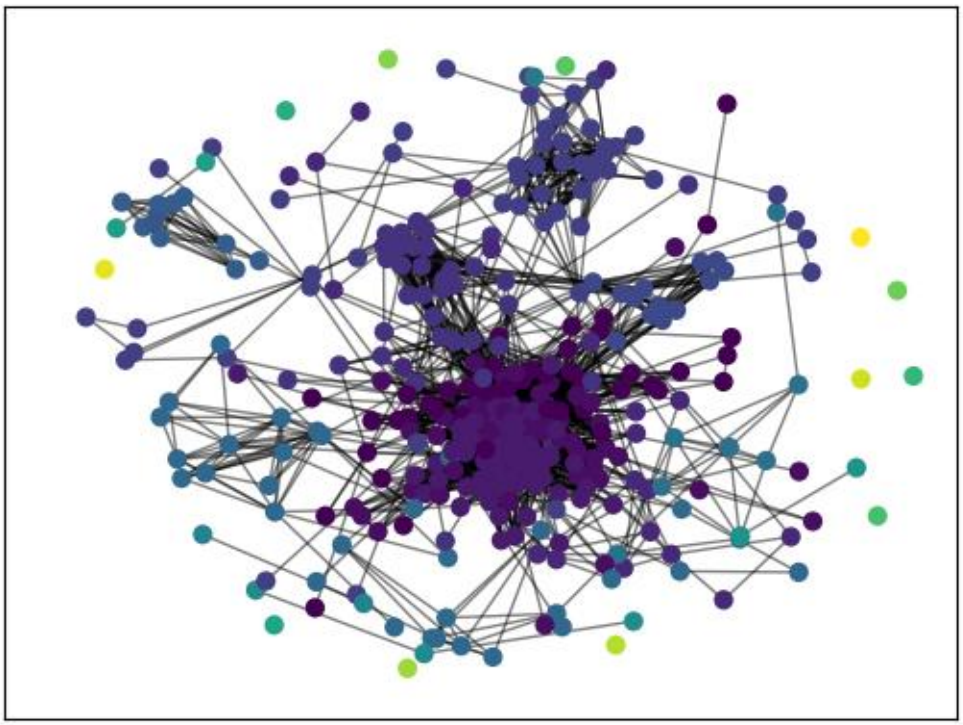
```
import networkx as nx
```

```
import community.community_louvain as community_louvain

# Create a graph and add author nodes and coauthored edges
G = nx.Graph()

# Assuming authors and coauthored relationships are loaded from your Neo4j database
authors = df['author_id']  # List of author IDs
coauthored_relationships = community_check()  # List of (author_id1, author_id2) tuples

for author_id in authors:
    G.add_node(author_id)

for author_id1, author_id2 in coauthored_relationships:
    G.add_edge(author_id1, author_id2)

# Perform community detection using the Louvain algorithm
partition = community_louvain.best_partition(G)

# Create a dictionary to group authors by community
communities = {}
for author_id, community_id in partition.items():
    if community_id not in communities:
        communities[community_id] = []
    communities[community_id].append(author_id)

# Print the authors in each community
for community_id, author_list in communities.items():
    print(f"Community {community_id}: {author_list}")
```

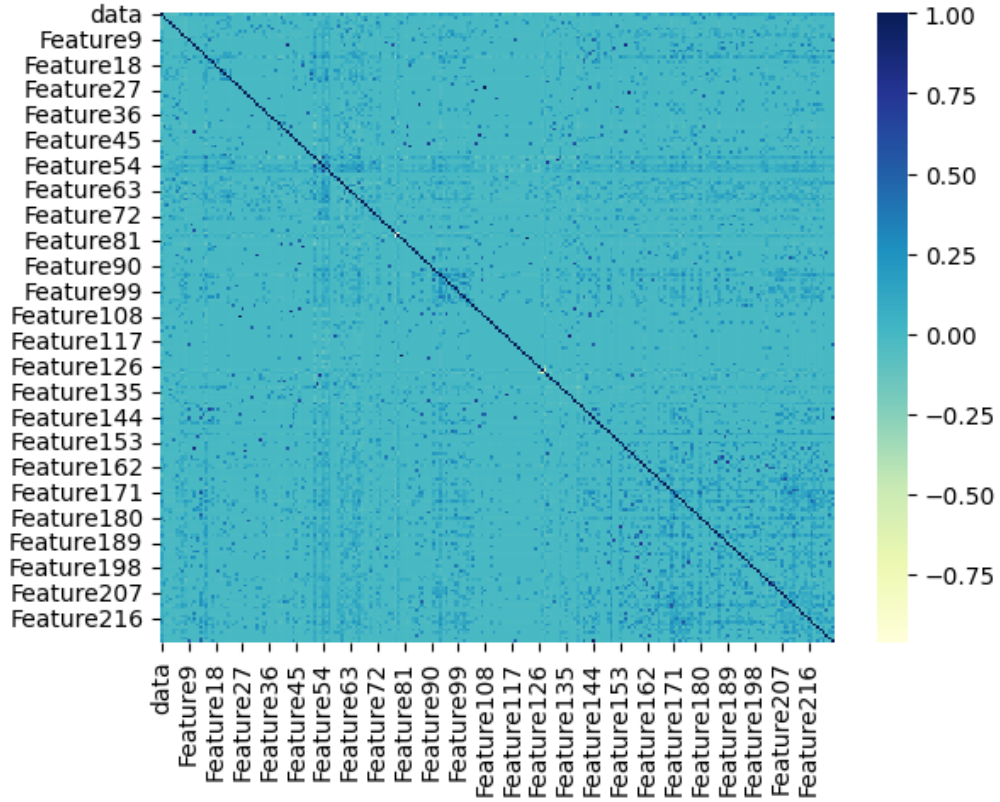
Create an empty graph G using nx.Graph() to represent the network of authors and their co-authored relationships. Since we have author data preloaded from your Neo4j database by community check function, you iterate through the list of author IDs and add each author as a node in the graph using G.add_node(author_id). You iterate through these relationships and add them as edges between corresponding authors in the graph using G.add_edge(author_id1, author_id2).

You then perform community detection using the Louvain algorithm by calling community_louvain.best_partition(G), which assigns each author to a community based on the network structure. You create a dictionary communities to group authors by their assigned community. The keys are community IDs, and the values are lists of author IDs belonging to each community.

We observed there were 27 communities. Output was a huge collection so I rather plotted it to make it visualizable.



My Third and last observation was to check for any correlation between features1 to 224. It was observed that majorly features were independent which made it easier to not to handle any derived features added more power to the model. This was the plot.



My Work can be seen in balkanid_task.ipynb

Part 2: Crafting Tomorrow's Chapters

For this phase, given an author (who is already present in the graph), your task is to predict the 5 most likely authors from the graph, who could co-author with the given author. You are required to use graph neural networks for this section. This section will be evaluated based on the quality of results.

I have used PyTorch geometric. PyTorch Geometric is a library built upon PyTorchto write and train GNNs there are various functions in torch_geometric.nn, torch.nn, torch_geometric.data the training part is visible in the balkanid_task.ipynb

```
# Create a mapping of author IDs to their corresponding indices
x = {author_id: idx for idx, author_id in enumerate(authors)}

# Create the edge_index from coauthored_relationships
edge_index = torch.tensor([[x[author_id1], x[author_id2]]
                           for author_id1, author_id2 in coauthored_relationships],
                          dtype=torch.long).t().contiguous()

authors = df['author_id']

author_features = {}
for index, row in df.iterrows():
    author_id = row['author_id']
    features = row[lis].tolist()
    author_features[author_id] = features

# Create the author_features tensor
author_features_tensor = torch.tensor([author_features[author_id] for author_id in authors],
dtype=torch.float)

data = Data(x=x, edge_index=edge_index)
```

This part of code is data preprocessing pipeline for a graph-based machine learning task, likely related to author recommendation or graph analysis. x represents the part where each author ID is mapped to its corresponding index. This mapping is useful for indexing into tensors and efficiently representing graph data.

Then we construct an edge index tensor where each row contains two author indices representing a co-authorship relationship. coauthored relationship is collection of all edges representing the relation b/w 2 authors. then we define author features here we map each author to its corresponding features 1 to 224.

Now we can create a data object. Data object using PyTorch Geometric. Data objects are commonly used to encapsulate graph data for graph neural networks (GNNs). lis is a list containing the column names from feature 1 to 224

```
class GCNRecommender(torch.nn.Module):
    def __init__(self, num_features, hidden_dim, num_classes):
        super(GCNRecommender, self).__init__()
        self.conv1 = GCNConv(num_features, hidden_dim)
```

```
        self.conv2 = GCNConv(hidden_dim, num_classes)

    def forward(self, x, edge_index):
        x = F.relu(self.conv1(x, edge_index))
        x = self.conv2(x, edge_index)
        return x
```

In PyTorch, neural network models are typically defined as subclasses of torch.nn.Module to leverage its functionalities for managing model parameters, gradients, and other essential components.
The init method is the constructor for the GCNRecommender class. It is called when you create an instance of the class. This method initializes the model's architecture and defines its layers and parameters.

self.conv1: This is the first GCN layer. It takes num_features as input features and projects them into a hidden_dim-dimensional space. It performs the first round of message passing over the graph. self.conv2: This is the second GCN layer. It takes the hidden_dim-dimensional features from the first layer and further projects them into the final output space with num_classes dimensions. This layer may incorporate information from neighboring nodes, eventually producing node embeddings that can be used for recommendation.

The forward method defines the forward pass of the neural network. It specifies how input data should propagate through the layers defined in the constructor. x: This parameter represents the input features for each node in the graph. It's assumed to be a tensor of shape (num_nodes, num_features). edge_index: This parameter represents the graph's edge connections, typically given as a tensor containing pairs of nodes that are connected in the graph.

The forward pass consists of the following steps:

1. self.conv1(x, edge_index): The input features x are passed through the first GCN layer (self.conv1). This step performs the first round of message passing and applies a ReLU activation function (F.relu) to introduce non-linearity to the model. The result is a new set of node embeddings.
2. self.conv2(x, edge_index): The node embeddings obtained from the first layer are further processed by the second GCN layer (self.conv2). This layer doesn't apply an activation function since it's producing the final output.
3. The final node embeddings (representing features in the num_classes-dimensional space) are returned as the output of the forward pass.

now we can can create a model object.

```
# Instantiate the model and define an optimizer
num_features = author_features_tensor.shape[1]
hidden_dim = 64
num_classes = len(authors)

model = GCNRecommender(num_features, hidden_dim, num_classes)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

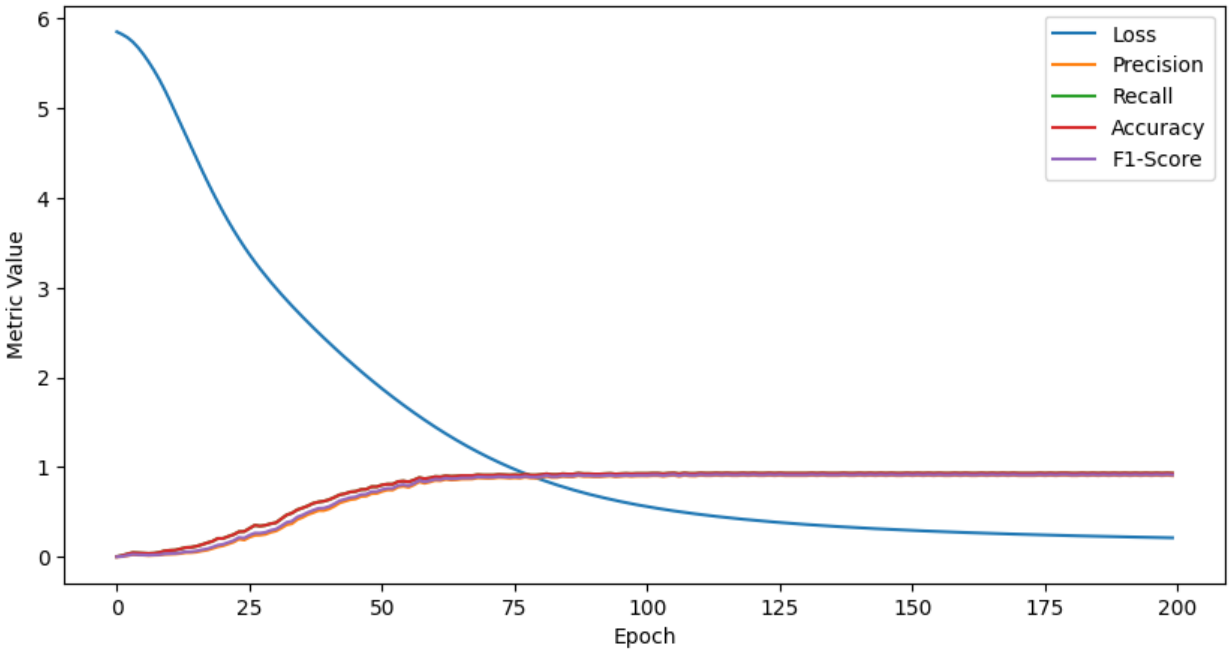
This is the train function.

```
# Train the Model and Print Loss and Additional Data for Each Epoch
def train():
```

```
    model.train()
    optimizer.zero_grad()
    out = model(author_features_tensor, edge_index)
    loss = F.cross_entropy(out, torch.tensor(int_authors, dtype=torch.long))
    loss.backward()
    optimizer.step()

    # Calculate predicted class labels
    _, predicted_labels = torch.max(out, 1)

    # Calculate precision, recall, accuracy, and F1-score
    precision = precision_score(int_authors, predicted_labels, average='macro')
    recall = recall_score(int_authors, predicted_labels, average='macro')
    accuracy = accuracy_score(int_authors, predicted_labels)
    f1 = f1_score(int_authors, predicted_labels, average='macro')

    return loss.item(), precision, recall, accuracy, f1
```

This is the final metrics.

Loss: 0.21200713515281677
Precision: 0.9048991354466859
Recall: 0.930835734870317
Accuracy: 0.930835734870317
F1-Score: 0.9124468231096472



here is an example of testing the model.

```
import math
# Author Recommendations
def recommend_authors(query_author_idx, num_recommendations=5):
```

```
    model.eval()
    with torch.no_grad():
        out = model(author_features_tensor, edge_index)
        query_embedding = out[query_author_idx]
        distances = torch.norm(out - query_embedding, dim=1)
        likeliness = 1.0 / (1.0 + distances)  # Likeliness inversely proportional to distance
        sorted_indices = torch.argsort(likeliness,descending=True)
        recommendations = [authors[idx.item()] for idx in sorted_indices if idx !=
query_author_idx][:num_recommendations]
        return recommendations, sorted_indices

author_id = input("Enter Author Id ")
query_author_idx = x[author_id]
recommended_authors, sorted_indices = recommend_authors(query_author_idx)

print("Recommended Authors in Increasing Order of Likeness:")
for idx, author in enumerate(recommended_authors, start=1):
    likeness = 1.0 / (1.0 + math.log(1 + idx))
    print(f"{idx}. Author: {author}, Likeness: {likeness:.4f}")
```

The above code is used to run the model that i trained earlier. This is example run

Enter Author Id authorID_2ac87_8b0e2_18061_6993b_4b6aa
Recommended Authors in Increasing Order of Likeness:
1. Author: authorID_a88a7_902cb_4ef69_7ba0b_6759c, Likeness: 0.5906
2. Author: authorID_aea92_132c4_cbeb2_63e6a_c2bf6, Likeness: 0.4765
3. Author: authorID_01d54_579da_446ae_1e75c_da808, Likeness: 0.4191
4. Author: authorID_482d9_673cf_ee5de_391f9_7fde4, Likeness: 0.3832
5. Author: authorID_8d1ed_e4f88_9e0ed_6f082_3d8c1, Likeness: 0.3582

Finally I saved the model as model.pt

torch.save(model.state_dict(), "model.pt")

References: https://pytorch-geometric.readthedocs.io/en/latest/ ---> Pytorch Geometric Docs

Part 3: Cloud Chronicles

This phase of the task requires you to deploy your model to the cloud. When a GET request is done to the deployed endpoint, the endpoint must return a JSON output as shown below. If your method of calculating potential coauthors does not provide likeliness, the output can contain a 1 in place of the value.

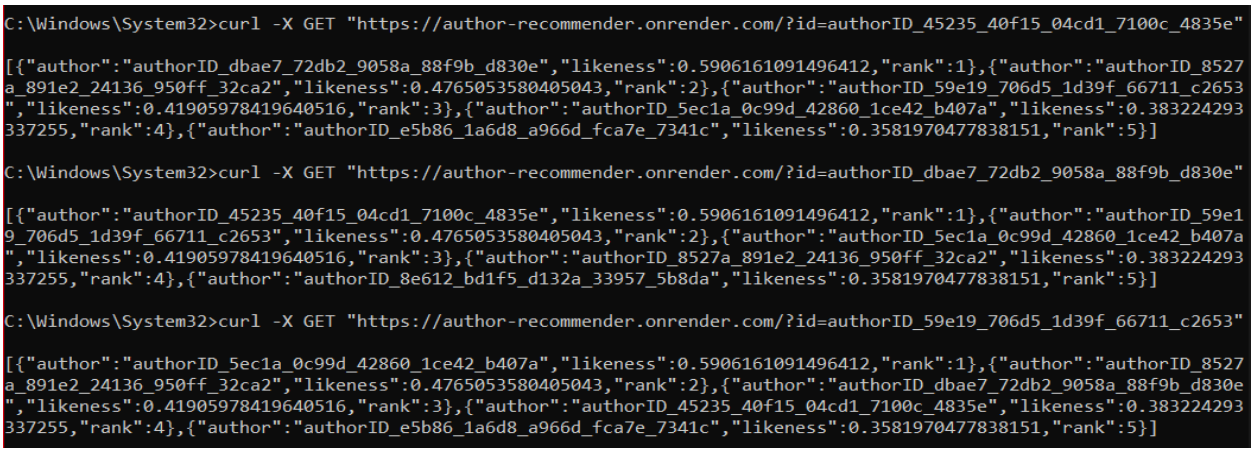
This part has been implemented in app.py

This was probably the hardest part that I faced as I dont have an upperhand in deployment as I am still learning. I tried various Services like Heroku, GCP, Azure but at the end I used render as it provided me with free hosting and finally with some working i finally deployed the work.

The likeness since we were not provided by any metrics so we have used the logarithmic of the index of the prediction.

The output can be Access as curl -X GET "https://author-recommender.onrender.com/?id=<author_id>"

Here I am attaching screenshots of the outputs



This marks the End of the Task.

Thank You.