

Data Science Task

Task: Providing Authors with Co-author suggestions

SUBMISSION BY :

ANJALI KEDIA

anjali.kedia2021@vitstudent.ac.in

+91 9560427614

Part 2: Crafting Tomorrow's Chapters (Forecasting Potential Writing Collaborations)

-> What you can expect here:

- I will be using PyTorch Geometric to implement my GNN.
- I explain various code blocks present in my collab file, AnjaliKedia_BalkanID_task.ipynb (uploaded in the repo). I have also uploaded the csv file('updated_authors.csv') that I created.
- Finally, I will show the results of 3 authors' recommended co-authors.
(If you'd like to see them first, skip to page -> 9)

Step 1: I first import the necessary libraries, pandas and torch, and utilize the torch_geometric library to work with graph data. It also loads a CSV file named 'updated_authors.csv' into a Pandas DataFrame called df.

Step 2: I define a function, extract_author_id, which uses regular expressions to search for the pattern 'author_id: "value"' within a given 'author_instance' string. If found, it returns the extracted 'value'; otherwise, it returns None. Then, it applies this function to a DataFrame column 'a' to extract author IDs and creates a new column 'author_id.' Finally, it prints the 'author_id' values from the DataFrame. This code appears to be extracting author IDs from text data and adding them to a DataFrame for analysis or processing.

(Note: I have used the author to be predicted for co-authors as a label, so you insert the index of the author that you want to predict for)

```
# Define the author for which you want to predict co-authors (Author_X)
# author_x = 'authorID_766cb_53c75_3baed_ac5dc_78259'
index_to_select = 100 # Change this to the desired row number

# Get the author_id from the selected row
author_x = df.loc[index_to_select, 'author_id']
# Create a label column based on whether each row's 'author_id' matches Author_X
df['label'] = (df['author_id'] == author_x).astype(int)
labels = df['label'].tolist()

# Now, the 'label' column contains binary labels (1 for Author_X, 0 for others)
```

Step 3: Next, I prepare data for a graph-based machine learning task. It focuses on a specific author, 'Author_X,' and performs the following tasks:

Select a row from a DataFrame based on an index to determine 'Author_X' and assign it to the variable 'author_x.'

Create a binary label column ('label') indicating whether each row's 'author_id' matches 'Author_X.' Map author IDs ('a' and 'coauthors') to unique numerical indices, creating a mapping dictionary ('author_id_to_index'). Convert the DataFrame into a numerical edge list and create a PyTorch Geometric Data object ('data') with edge indices and labels. Define a placeholder for node features ('data.x') with a specified number of nodes and features.

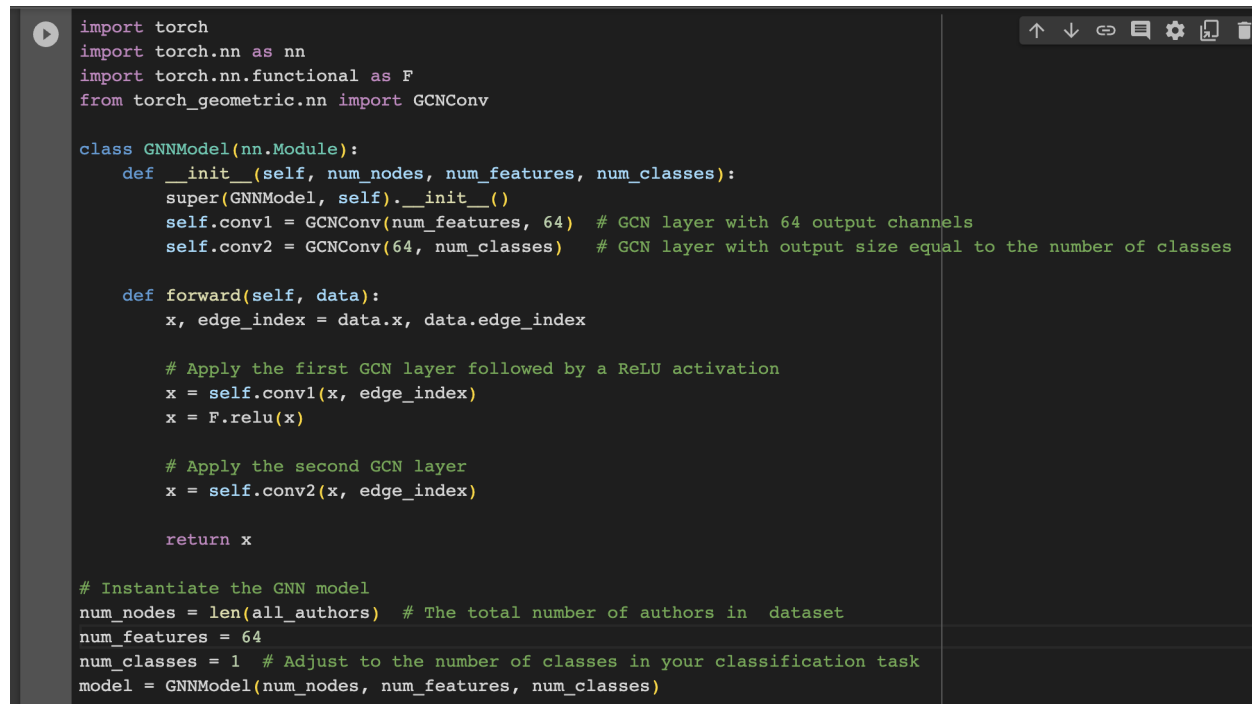
This code sets up the data structure necessary for graph-based machine learning, where nodes represent authors, edges represent co-authorships, and 'Author_X' is singled out for prediction or analysis.

Step 4: Create a NetworkX Graph: I convert the edge list into a NetworkX graph, which is a common format for working with graphs in Python. NetworkX can be used to perform various graph operations.

Step 5: GNNModel is a custom GNN model class that inherits from nn.Module. The model consists of two GCN layers (self.conv1 and

self.conv2), but you adjust the number of layers and hidden units as needed. The forward method defines the forward pass of the model. It applies the first GCN layer, a ReLU activation, and then the second GCN layer.

(Note: This is the GNN Model I created, although it has node_features listed, we further don't significantly make use of node_features)



```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GCNConv

class GNNModel(nn.Module):
    def __init__(self, num_nodes, num_features, num_classes):
        super(GNNModel, self).__init__()
        self.conv1 = GCNConv(num_features, 64) # GCN layer with 64 output channels
        self.conv2 = GCNConv(64, num_classes) # GCN layer with output size equal to the number of classes

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        # Apply the first GCN layer followed by a ReLU activation
        x = self.conv1(x, edge_index)
        x = F.relu(x)

        # Apply the second GCN layer
        x = self.conv2(x, edge_index)

        return x

# Instantiate the GNN model
num_nodes = len(all_authors) # The total number of authors in dataset
num_features = 64
num_classes = 1 # Adjust to the number of classes in your classification task
model = GNNModel(num_nodes, num_features, num_classes)
```

Step 6: This code defines a Graph Neural Network (GNN) model for a graph-based machine learning task using PyTorch Geometric. The GNNModel class inherits from nn.Module and consists of two Graph Convolutional Network (GCN) layers.

The forward method computes the forward pass of the model. It takes node features (x) and edge indices (edge_index) as input, applies the first GCN layer followed by a ReLU activation, and then applies the second GCN layer. This architecture is suitable for tasks like node classification in graphs.

The model is instantiated with the specified number of nodes, features, and classes. It is designed to be adaptable to different graph-based classification tasks by adjusting the number of features and classes accordingly.

Step 7: Preparation for the training of a graph-based machine learning model.

1. Loss Function and Optimizer:

- It defines a binary cross-entropy loss (`nn.BCEWithLogitsLoss()`) suitable for binary classification tasks. The model aims to predict whether nodes belong to a specific class (1) or not (0).
- It uses the Adam optimizer (`optim.Adam`) to update the model's parameters during training. Adam is a popular optimization algorithm that adapts the learning rate for each parameter based on its past gradients, which often leads to faster and more stable convergence.

2. Data Splitting:

- It splits the dataset into training, validation, and test sets with specified ratios.
- Randomly shuffles the indices of nodes to ensure randomness in each split.
- Extracts labels for each split, creating `train_labels`, `val_labels`, and `test_labels`.

```
import random

# Define the split ratios
train_ratio = 0.8
val_ratio = 0.1
test_ratio = 0.1

# Create a list of indices for all nodes
num_nodes = len(data.y)
all_indices = list(range(num_nodes))

# Determine the number of nodes for each split
train_size = int(train_ratio * num_nodes)
val_size = int(val_ratio * num_nodes)
test_size = num_nodes - train_size - val_size

# Ensure that train_size does not exceed the maximum valid index
train_size = min(train_size, num_nodes - 1)

# Shuffle the indices randomly
random.shuffle(all_indices[:train_size])

# Split the indices into train, validation, and test sets
train_indices = all_indices[:train_size]
val_indices = all_indices[train_size:train_size + val_size]
test_indices = all_indices[train_size + val_size:]

# Extract the labels for each set
train_labels = [labels[i] for i in train_indices]
val_labels = [labels[i] for i in val_indices]
test_labels = [labels[i] for i in test_indices]
```

3. Usage in Training Loop:

- After splitting the data, you can use these masks and labels during the training loop. For example, you can use `train_labels` to compute the loss during training and use `val_labels` to monitor the model's performance on a validation set to prevent overfitting.

(Adam is an optimization algorithm that adapts learning rates for each parameter, helping the model converge efficiently during training.)

Step 8: In this code, a training loop for a machine learning model is run. It runs for a specified number of epochs (100 in this case) and performs the following steps:

```
num_epochs = 100
train_labels_tensor = torch.zeros((data.size(0),))

for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()

    # Forward pass
    output = model(data)
    output = torch.squeeze(output)

    # Print a subset of output and train_labels_tensor for debugging
    print("Output:", output[:10]) # Print the first 10 elements
    print("Train labels:", train_labels_tensor[:10]) # Print the first 10 labels

    # Ensure that train_labels_tensor matches the size of output
    train_labels_tensor = train_labels_tensor.view(-1)

    # Debugging: Print a subset of train_indices
    # print("Train indices:", train_indices[:10]) # Print the first 10 indices

    # Ensure that train_indices are within bounds
    train_indices = [idx for idx in train_indices if idx < output.size(0)]

    # Calculate the binary cross-entropy loss using train_labels_tensor
    train_loss = criterion(output[train_indices], train_labels_tensor[train_indices])

    # Backpropagation
    train_loss.backward()
    optimizer.step()

    # Print loss for monitoring
    print(f'Epoch [{epoch + 1}/{num_epochs}], Training Loss: {train_loss.item()}')
```

```
Train labels: tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
Epoch [100/100], Training Loss: 0.06338164955377579
```

1. Model Training:

- `model.train()` sets the model in training mode, enabling features like dropout or batch normalization.

- `optimizer.zero_grad()` clears the gradients of the model's parameters before the forward pass.

2. Forward Pass:

- The model is used to make predictions (``output``) for the given input data (``data``).
- The ``output`` is squeezed to remove dimensions of size 1.

3. Debugging Outputs:

- A subset of the model's output and the training labels (``train_labels_tensor``) is printed for debugging purposes. This helps to observe how the model's predictions compare to the actual labels.

4. Data Preparation:

- The size of ``train_labels_tensor`` is adjusted to match the size of the ``output``.
- The ``train_indices`` are checked to ensure they are within bounds.

5. Loss Calculation:

- The binary cross-entropy loss is calculated using the model's output and the training labels, but only for the specified ``train_indices``.

6. Backpropagation and Optimization:

- The gradients of the loss are computed during backpropagation (``train_loss.backward()``) and used to update the model's parameters using the optimizer (``optimizer.step()``).

7. Monitoring:

- The training loss for the current epoch is printed to monitor the training progress.

(Note: This is the prediction code)

```

import numpy as np
model.eval()

# Forward pass to generate predictions for all nodes
with torch.no_grad():
    predictions = model(data)
    predictions = torch.sigmoid(predictions) # Apply sigmoid activation for probability scores

# Convert predictions to a numpy array for easier sorting
predictions = predictions.cpu().numpy()

# Sort predictions in descending order
sorted_indices = np.argsort(predictions[:, 0])[::-1] # Sort by the first column (probability of being similar)

# Ensure that sorted_indices do not exceed the length of data.y
sorted_indices = sorted_indices[sorted_indices < len(data.y)]

```

Step 9: In this code, the trained model is evaluated to predict potential co-authors for a given author. It performs the following steps:

1. Model Evaluation:

- The model is set to evaluation mode using `model.eval()` to disable features like dropout, which are active during training but not during evaluation.

2. Forward Pass for Predictions:

- The model is used to make predictions (`predictions`) for all nodes in the dataset.
- A sigmoid activation is applied to these predictions to obtain probability scores.

3. Sorting Predictions:

- Predictions are converted to a NumPy array for easier sorting.
- Predictions are sorted in descending order based on the first column (probability of being similar) using `np.argsort`.

4. Handling Index Bounds:

- To avoid index out-of-bounds errors, the sorted indices are filtered to ensure they do not exceed the length of `data.y`.

5. Top Author Selection:

- If there are at least 5 elements in `data.y`, the top 5 authors with the highest predicted probabilities are selected.
- The selected authors are printed as potential co-authors.

6. Handling Insufficient Data:

- If there are fewer than 5 authors in `data.y`, a message is printed indicating that there are not enough authors to predict the top 5.

This code demonstrates how to use a trained model to predict and rank potential co-authors based on their likelihood of collaboration, making it a useful tool for recommendation systems or network analysis.

```
if len(data.y) >= 5:
    num_top_authors = min(5, len(top_authors_indices))
    print("For author:", author_x)

    print("Top 5 Authors Who Could Co-Author:")
    for author_index in top_authors_indices:
        if author_index < len(df):
            author_id = df.loc[author_index, 'author_id']
            print(f"Author Index: {author_index}, Author ID: {author_id}")
        else:
            print(f"Author Index: {author_index}, Author ID: Not found in DataFrame")
    else:
        print("Not enough authors in data.y to predict the top 5.")
```

```
For author: authorID_3c152_85c04_fff40_024bb_8714b
Top 5 Authors Who Could Co-Author:
Author Index: 305, Author ID: authorID_82c01_ce15b_431d4_20eb6_alfeb
Author Index: 293, Author ID: authorID_8ede6_b2634_3305e_05c3c_0029f
Author Index: 291, Author ID: authorID_0e17d_aca5f_3e175_f448b_acace
Author Index: 319, Author ID: authorID_090d3_859ff_6840b_2280f_4708c
Author Index: 345, Author ID: authorID_6db6e_b4af1_e18ab_81d38_78e44
```


OUTPUT:

```

[➔] For author: authorID_3c152_85c04_fff40_024bb_8714b
Top 5 Authors Who Could Co-Author:
Author Index: 288, Author ID: authorID_011af_72a91_0ac4a_cf367_eef9e
Author Index: 312, Author ID: authorID_d029f_a3a95_e174a_19934_857f5
Author Index: 281, Author ID: authorID_d6a40_31733_610bb_080d0_bfa79
Author Index: 297, Author ID: authorID_a4e00_d7e6a_a8211_15754_38c5e
Author Index: 326, Author ID: authorID_f8809_aff4d_69bec_e79da_be35b

```

```

[➔] For author: authorID_7688b_6ef52_55596_2d008_fff89
Top 5 Authors Who Could Co-Author:
Author Index: 309, Author ID: authorID_8acc2_3987b_8960d_83c44_541f9
Author Index: 304, Author ID: authorID_5316c_alc5d_dca8e_6cecc_fce58
Author Index: 305, Author ID: authorID_82c01_ce15b_431d4_20eb6_alfeb
Author Index: 284, Author ID: authorID_3f980_7cb9a_e9fb6_c3094_2af61
Author Index: 312, Author ID: authorID_d029f_a3a95_e174a_19934_857f5

```

(Note: The following author is an author who had no co-authors earlier)

```

For author: authorID_6db6e_b4af1_e18ab_81d38_78e44
Top 5 Authors Who Could Co-Author:
Author Index: 315, Author ID: authorID_41cfc_0d1f2_d127b_04555_b7246
Author Index: 281, Author ID: authorID_d6a40_31733_610bb_080d0_bfa79
Author Index: 298, Author ID: authorID_8d23c_f6c86_e834a_7aa6e_ded54
Author Index: 303, Author ID: authorID_08490_29548_8a118_90997_51ebe
Author Index: 286, Author ID: authorID_a88a7_902cb_4ef69_7ba0b_6759c

```

You can refer to how I proceed with my attempt on deploying the model to cloud using this link:

<https://docs.google.com/document/d/140vppz1Dx-ls8KdkFCNbQLAQdw7bDB9Y3Y7su66UoXA/edit?usp=sharing>

Or, you may refer to the repo, where I have uploaded the documentation for that

Thank you for your patience, and an opportunity to explore the world of graph neural networks! :)