NCSR DEMOKRITOS
UNIVERSITY OF PIRAEUS

MSc in Artificial Intelligence

Link Prediction with Graph Neural Networks

Author :
Andreas Sideras

Supervisor:
Maria Halkidi

June 28, 2023

Abstract

In this exercise, our primary objective is to comprehensively examine a given graph, thoroughly analyzing its structure and properties. Building upon this analysis, we delve into the challenging task of link prediction utilizing the power of Graph Neural Networks (GNNs). With an aim to enhance predictive accuracy, we explore and compare various GNN architectures. By conducting an evaluation of these architectures, we aim to gain insights into their strengths and weaknesses, ultimately advancing our understanding of effective link prediction techniques in graph analysis.

Contents

1 Introduction

Graph representation is important because it allows us to model and analyze complex relationships and dependencies between entities or data points. The structure of graphs captures relationships between entities or data points. Nodes represent entities, while edges represent connections or relationships between them. This structure enables us to understand the connections and dependencies within a system, revealing valuable insights that may be overlooked in other data representations. Graph structures facilitate pattern recognition by highlighting the arrangement and connectivity of nodes. By analyzing the relationships and patterns formed by nodes and edges, we can identify clusters, communities, or recurring motifs within the graph. These patterns can provide valuable information about the underlying data or system being represented. Graph neural networks (GNNs) are a type of deep learning model designed to operate on graph-structured data. Unlike traditional neural networks that work well with grid-like data, GNNs excel at capturing relational information and dependencies within complex graphs. They are particularly effective in tasks involving node classification, link prediction, and graph-level tasks like graph classification. GNNs leverage message passing algorithms to propagate information across the nodes in a graph, allowing each node to update its representation based on the information received from its neighbors. This enables GNNs to encode both local and global structural patterns, making them well-suited for analyzing and learning from graph data. In this exercise we will perform link prediction using GNNs.

2 Graph Analysis

The dataset we used is a subset of "Statistics and Social Network of YouTube Videos" `https://netsg.cs.sfu.ca/youtubedata/`. Namely, our dataset comprises of 9064 nodes and 43963 edges. Each node represents a YouTube video and it is described with 7 attributes. An integer number of days between the date when the video was uploaded and Feb.15, 2007 (YouTube's establishment), a string of the video category chosen by the uploader, an integer number of the video length, an integer number of the views, a float number of the video rate, an integer number of the ratings and an integer number of the comments. Each video is correlated with some other videos. This relation is represented with directed edges in our graph. If a video b is in the related video list (first 20 only) of a video a, then there is a directed edge from a to b.

By using networkx library we are able to analyze our graph. A network can be an exceedingly complex structure, as the connections among the nodes can exhibit complicated patterns. One challenge in studying complex networks is to develop simplified measures that capture some elements of the structure in an understandable way. One such simplification is to ignore any patterns among different nodes, but just look at each node separately. If one zooms in onto a node and ignores all other nodes, the only thing one can see is how many connections the node has, i.e., the degree of the node. In the figures below we plotted the degree distributions in a logarithmic scale.

As the regards the in-degree distribution we can see [1] that is seems to follow the power-law distribution. If the in-degree distribution of a graph follows a power-law distribution, it indicates that the graph exhibits a *scale-free* property. In a scale-free network, a few nodes have a significantly higher number of incoming connections (high in-degree), while the majority of nodes have relatively fewer incoming connections. This means that a small number of nodes, known as "hubs," play a crucial role in the connectivity and structure of the graph. On the other hand, the out-degree is a bit more uniformly distributed [2]. These plots lead us to believe that our graph is an *asymmetric* graph. In an asymmetric graph, the connectivity patterns of nodes differ between incoming and

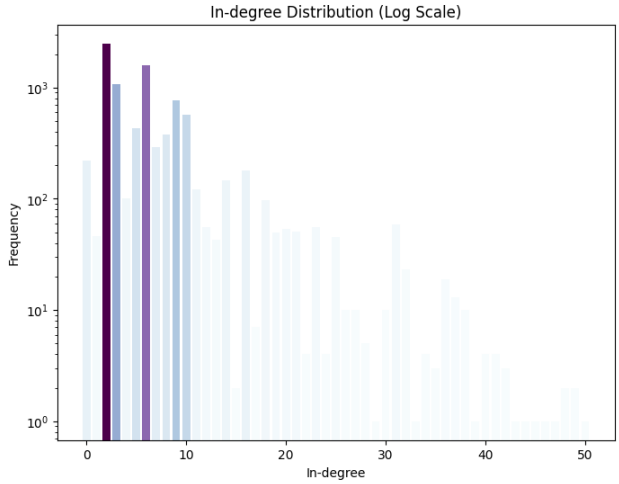

Fig. 1: Incoming edges distribution

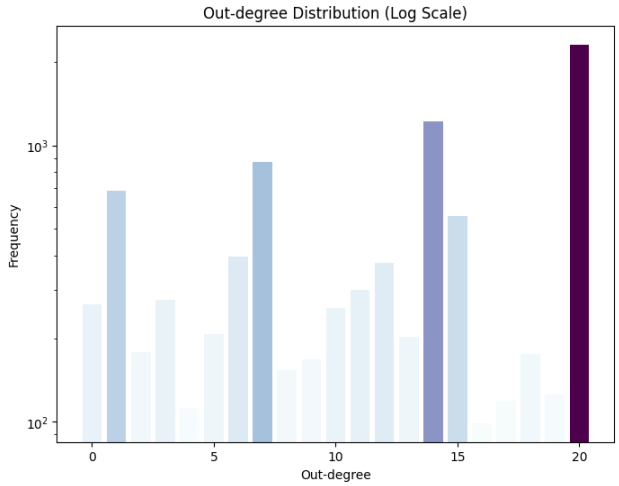

Fig. 2: Out-coming edges distribution

outgoing connections. This means that some nodes receive a large number of incoming connections (resulting in a power-law in-degree distribution), while the outgoing connections from those nodes are more evenly distributed (resulting in a more uniformly distributed out-degree distribution).

The figure [4] illustrates the shortest path distribution of our graph.

The same scenario applies here, a power-law distribution. If a graph exhibits a power-law distribution in its shortest path length distribution, it implies that the graph has a *small-world* property. The shortest path length distribution describes the distribution of the minimum number of edges or steps required to travel between pairs of nodes in the graph. In a power-law distribution of shortest path lengths, the frequency of short path lengths is relatively high, while the frequency decreases gradually as the path length increases. This means that there are many pairs of nodes that can be connected by a short path, while a few pairs of nodes are connected by significantly longer paths. The small-world property suggests that even in a large graph, the average distance between nodes is relatively small. In other words, most nodes in the graph can be reached from any other node

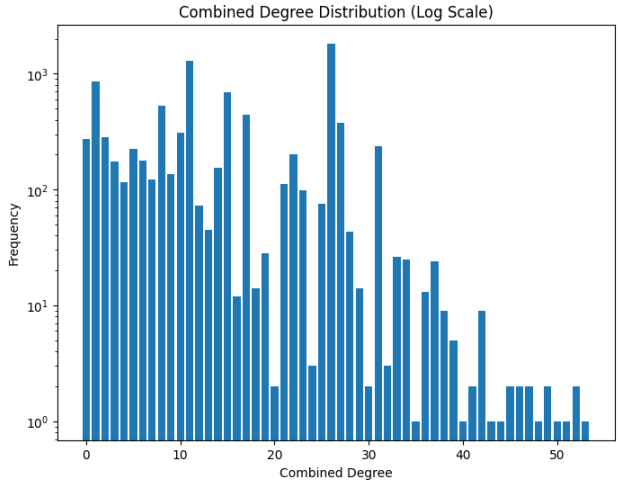

Fig. 3: Undirected connections distribution

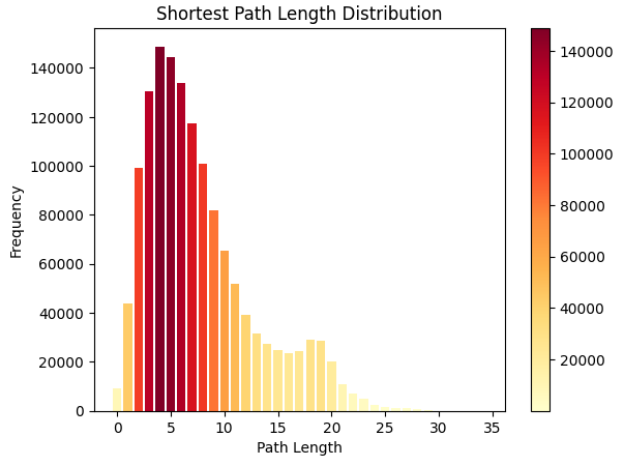

Fig. 4: Shortest Path distribution (Linear scale)

through a relatively small number of steps or edges.

In the figure [5] below we can see the betweeness, closeness and (in) degree centrality distributions. entrality measures used to quantify the importance or centrality of nodes in a graph. The centrality of a node in a graph is a measure that quantifies the importance or prominence of that node within the graph. Centrality measures help identify nodes that play critical roles or have significant influence in various aspects of the network, such as connectivity, information flow, or control. A power-law distribution in these centrality measures indicates that the graph follows a scale-free property in terms of node importance. Closeness centrality measures how easily a node can reach other nodes in a graph. If the closeness centrality follows a power-law distribution, it implies that there are a few nodes with high closeness centrality values, indicating that they can reach other nodes in the graph more efficiently than other nodes. These highly central nodes act as important communication or information flow channels within the graph. Betweenness centrality quantifies the extent to which a node lies on the shortest paths between other pairs of nodes. If the betweenness centrality follows a power-law distribution, it suggests the presence of a few nodes that act as

important "bridges" or intermediaries between other nodes. These nodes have a high influence on the flow of information or resources within the graph. In-degree refers to the number of incoming edges or connections that a node receives in a directed graph. If the in-degree distribution follows a power-law distribution, it suggests that a few nodes (hubs) have a significantly higher number of incoming connections compared to the majority of nodes. These hub nodes play a crucial role in the graph's connectivity and information flow.

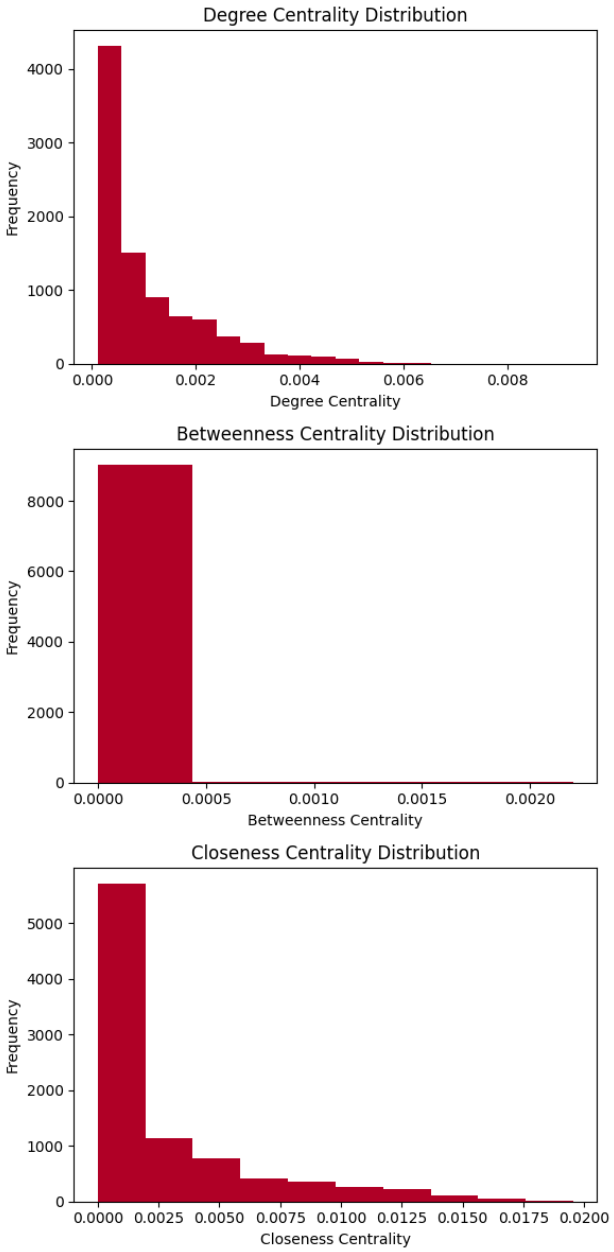

Fig. 5: Centrality distributions

3 Link Prediction and Graph Neural Networks

Link prediction refers to the task of predicting missing or potential links between entities in a network. It is widely used in various fields, such as social network analysis, recommender systems, biological networks, and knowledge graphs. The goal is to infer the likelihood of a connection between two nodes based on the observed network structure and other relevant attributes. Deep learning and graph neural networks (GNNs) [1] have proven to be effective approaches for solving link prediction problems. GNNs are a class of neural networks specifically designed to operate on graph-structured data. They capture both the structural information of the network and the node features, allowing them to learn complex patterns and dependencies among nodes. The network data is typically represented as a graph, where nodes represent entities, and edges represent connections or relationships between them. Each node can have associated features or attributes. In order to apply deep learning techniques, the graph needs to be converted into a continuous vector representation, known as graph embedding. This embedding captures the structural information of the graph in a low-dimensional space, where nodes and edges are represented as numerical vectors. GNNs operate by iteratively aggregating information from neighboring nodes and updating node representations. This process allows nodes to gather information from their local neighborhoods and propagate it through the graph (*message passing*). A subset of the graph is used for training the GNN model. The model is trained to predict whether a link exists between pairs of nodes based on their representations and other relevant attributes. Various loss functions, such as binary cross-entropy, can be used for training. After training, the model can be used to predict the likelihood of links between any pair of nodes in the graph. The node representations learned by the GNN capture the structural patterns and can be used to estimate the probability of a link between two nodes.

The architecture we used to build our link prediction model is an *Autoencoder* GNN [2]. An autoencoder is a type of neural network that aims to reconstruct its input data as accurately as possible. It consists of two main components: an encoder and a decoder as illustrated in figure [6]. The encoder compresses the input data into a lower-dimensional representation, while the decoder reconstructs the original input from the encoded representation. The model is trained to minimize the reconstruction error, encouraging it to learn a compact representation that captures the essential features of the input. When combined with a GNN, an autoencoder GNN incorporates the graph structure and node attributes into the encoding and decoding processes. It learns to encode each node's local neighborhood information in the latent space and decodes it to reconstruct the original graph. This process effectively learns a compressed representation that captures the structural patterns and node attributes of the graph. To solve link prediction tasks, an autoencoder GNN can be trained using a graph with known links as the input and with the goal of reconstructing the original graph structure. During training, a subset of the graph is randomly sampled, and some of the links are removed. The model is then trained to predict the missing links based on the encoded representations of the nodes and the reconstructed graph.

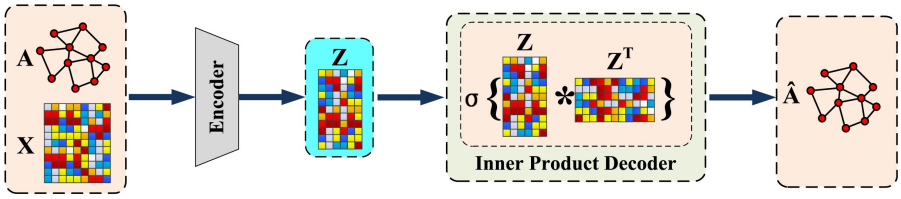

Fig. 6: Graph AutoEncoder

Apart from the 7 available features for each node, we computed 8 more based on their structural instance. For each node we computed the degree, betweenness, eigenvector and closeness centrality, the pagerank score, the in and out degree and the average neighbor degree. We concatenated these features into a single representation. As we said the goal is to transform/create a representation for each node in a latent space while incorporating information about their position in graph and their feature vector. In these latent space, we would like the representations of nodes that contain an edge between them to be close to each other. So there should be a decoder that tries to encapsulate this similarity. We could say that the decoder tries to reconstruct the graph because it learns to use the embeddings (representations) created in a way that links back together connected nodes (by providing high scores to their relation) and even to suggest the missing ones.

Our encoder starts with linear layers (linear1, linear2, and linear3) that map the input features of the graph to intermediate feature representations. These linear layers apply a linear transformation to the input features and output higher-level feature representations in a higher dimensional space. Up to this point we just have transformed the node attributes for each node. After each linear layer, a hyperbolic tangent activation function (tanh) is applied to introduce non-linearity and capture complex patterns in the data. It squashes the values between -1 and 1. Then, the encoded intermediate feature representations obtained from the linear layers and activation functions are then fed into a GNN layer (conv) based on the specified GNN version (we will elaborate on this later). The GNN layer performs *message passing* and *aggregation operations*, allowing nodes to exchange information with their neighboring nodes in the graph. The GNN layer refines the feature representations by incorporating local graph structure. After the GNN layer, another linear layer (linear4) is used to further transform the feature representations and prepare them for decoding. Overall, the encoder in the GAE model combines linear transformations, activation functions, and a GNN layer to encode the input features of the graph into a lower-dimensional representation that captures important graph structure and patterns. This encoded representation serves as a compact and informative representation of the graph, which can then be used for decoding and link prediction tasks.

Our decoder in the provided code takes the encoded representation obtained from the encoder and performs operations to decode and predict the existence of edges between nodes in the graph. The decoder takes as input the encoded representation, for a subset of indeed existing edges and for a **sampled** subset of non existing edges. The decoder computes the logits for each edge by taking the dot product between the embeddings of the starting and ending nodes and then by applying a sigmoid function, we get the likelihood of an edge existing between the corresponding nodes. The loss then is computed using the binary cross-entropy loss function. It measures the dissimilarity between the predicted logits (decoder's output) and the ground truth labels. The loss value is then used to compute the gradients of the model's parameters through backpropagation, using the Adam optimizer.

As we said earlier we can use different GNN types. A *GCN (Graph Convolutional Network)* [3]is a type of graph neural network designed to operate on graph-structured data. It leverages the structural information present in the graph to learn node representations by propagating information across the graph. A graph is typically represented by an adjacency matrix, which captures the connections between nodes. Each node in the graph is associated with an initial feature vector. These feature vectors can represent node attributes or embeddings. The key idea behind GCN is to iteratively propagate information between neighboring nodes to update their representations. This is done by aggregating and transforming the features of neighboring nodes. Each layer of the GNN allows information to propagate beyond the immediate neighbors of a node. As you stack multiple

GNN layers, each layer has the ability to capture information from increasingly distant nodes in the graph.

The update rule for a single node embedding in GCN can be expressed as follows:

$$h_v^{(l+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \frac{1}{|\mathcal{N}(v)|} W^{(l)} h_u^{(l)}\right) \tag{1}$$

In this equation:

- $h_u^{(l)}$ represents the feature vector of a neighboring node u of v at layer l.
- $\mathcal{N}(v)$ denotes the set of neighboring nodes of v.
- $|\mathcal{N}(v)|$ represents the degree of node v, which indicates the number of neighbors of node v.
- $W^{(l)}$ is the weight matrix associated with the l-th layer.
- σ is a non-linear activation function applied element-wise to introduce non-linearity (e.g., ReLU or sigmoid).

The next GNN architecture we experimented with is *GraphSAGE* [4].

The single node update rule of GraphSAGE is defined as follows:

Given a node v with its neighborhood $N(v)$, we can compute the updated representation $h_v^{(l+1)}$ of the node at layer $l+1$ as follows:

$$h_v^{(l+1)} = \sigma\left(W^{(l)} \cdot \text{CONCAT}\left(h_v^{(l-1)}, \text{AGGREGATE}\left(\{h_u^{(l)} \mid u \in N(v)\}\right)\right)\right) \tag{2}$$

where:

- $h_v^{(l)}$ represents the representation of node v at layer l.
- σ is the activation function (e.g., ReLU, sigmoid) applied element-wise.
- $W^{(l)}$ is a learnable weight matrix for layer l.
- AGGREGATE is an aggregation function that combines the representations of neighboring nodes. It can be a simple mean, max, or sum aggregation, or more complex functions like attention mechanisms.

Lastly, we experimented with *Graph Attention Networks (GAT)* [5].

The single node update rule of Graph Attention Networks (GAT) is defined as follows:

Given a node v with its neighborhood $N(v)$, we can compute the updated representation $h_v^{(l+1)}$ of the node at layer $l+1$ as follows:

$$h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu}^{(l)} \cdot W^{(l)} \cdot h_u^{(l)}\right) \tag{3}$$

where:

- $h_v^{(l)}$ represents the representation of node v at layer l.
- σ is the activation function (e.g., ReLU, sigmoid) applied element-wise.
- $\alpha_{vu}^{(l)}$ represents the attention coefficient between nodes v and u at layer l. It is computed as:

$$\alpha_{vu}^{(l)} = \text{softmax}\left(\vec{a}^{(l)} \cdot \left[W^{(l)} \cdot h_v^{(l)}, W^{(l)} \cdot h_u^{(l)}\right]\right) \tag{4}$$

  where $\vec{a}^{(l)}$ is a learnable weight vector specific to layer l and $[\cdot, \cdot]$ denotes concatenation.
- $W^{(l)}$ is a learnable weight matrix for layer l.

4 Results

We trained and compared the GNNs described in the previous section. Our training set size consists of 80,204 edges, the validation set consists of 2,358 edges, and the test set consists of 4,716 edges. We tuned the capacity and hyperparameters of the model by maximizing the f1 score on the validation set. Then we tested and compared the results. To ensure comparability, we kept the layers other than GNNs fixed for all three models. We used the Adam optimizer with a learning rate of 0.001 in a full batch setting. Namely, in each epoch we make a single step based on the loss from all positive and negative edges. We trained the models for 1000 steps.

GCN

Listing 1: Encoder Architecture

```
GAE(
  (linear1): Linear(in_features=15, out_features=64, bias=True)
  (tanh): Tanh()
  (linear2): Linear(in_features=64, out_features=256, bias=True)
  (tanh): Tanh()
  (linear3): Linear(in_features=256, out_features=1024, bias=True)
  (tanh): Tanh()
  (conv): GCN(1024, 256, num_layers=2)
  (linear4): Linear(in_features=256, out_features=64, bias=True)
)

The model has 953408 trainable parameters
```

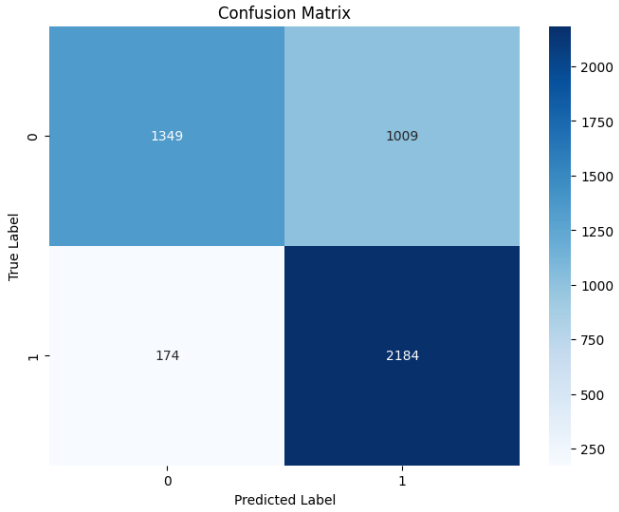

Fig. 7: GCN confusion matrix

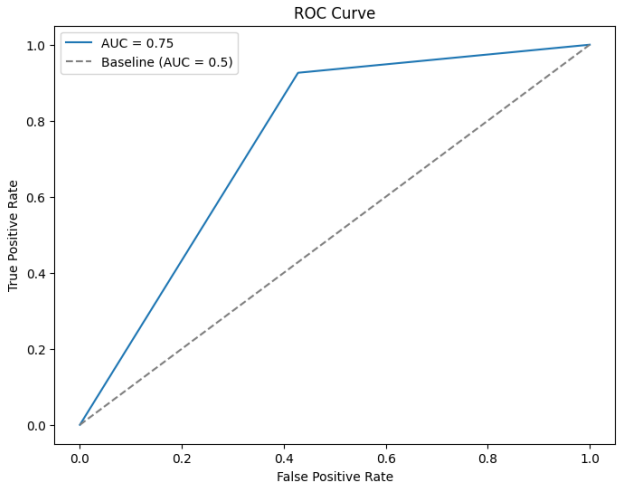

Fig. 8: ROC Curve

Table 1: Final GCN Test Performance

Metric	Score
Precision	0.684
Recall	0.926
F1 score	0.787
ROC AUC Score	0.749

GAT

Listing 2: Encoder Architecture

```
GAE(
  (linear1): Linear(in_features=15, out_features=64, bias=True)
  (tanh): Tanh()
  (linear2): Linear(in_features=64, out_features=256, bias=True)
  (tanh): Tanh()
  (linear3): Linear(in_features=256, out_features=1024, bias=True)
  (tanh): Tanh()
  (conv): GAT(1024, 256, num_layers=2)
  (linear4): Linear(in_features=256, out_features=64, bias=True)
)
The model has 954944 trainable parameters
```

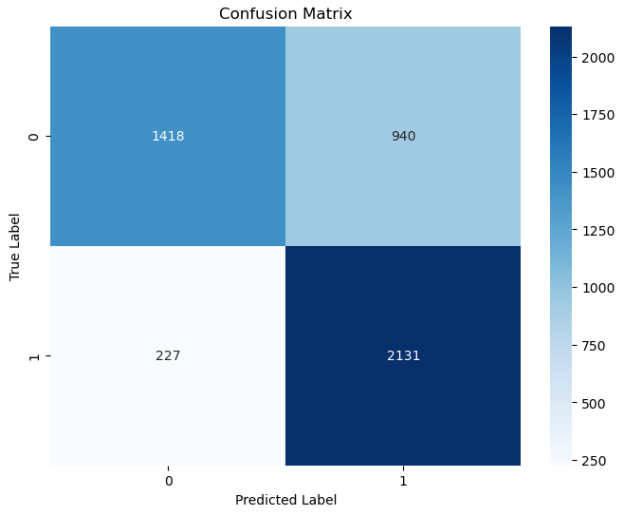

Fig. 9: GAT confusion matrix

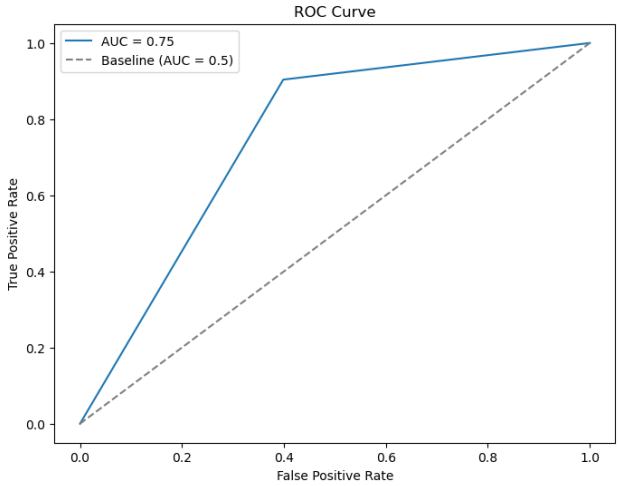

Fig. 10: ROC Curve

Table 2: Final GAT Test Performance

Metric	Score
Precision	0.694
Recall	0.904
F1 score	0.785
ROC AUC Score	0.753

SAGE

Listing 3: Encoder Architecture

```
GAE(
  (linear1): Linear(in_features=15, out_features=64, bias=True)
  (tanh): Tanh()
  (linear2): Linear(in_features=64, out_features=256, bias=True)
  (tanh): Tanh()
  (linear3): Linear(in_features=256, out_features=1024, bias=True)
  (tanh): Tanh()
  (conv): GraphSAGE(1024, 256, num_layers=2)
  (linear4): Linear(in_features=256, out_features=64, bias=True)
)
The model has 1608768 trainable parameters
```

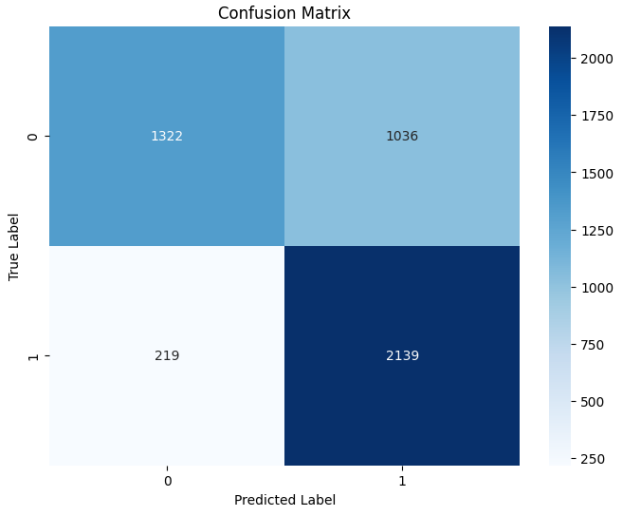

Fig. 11: SAGE confusion matrix

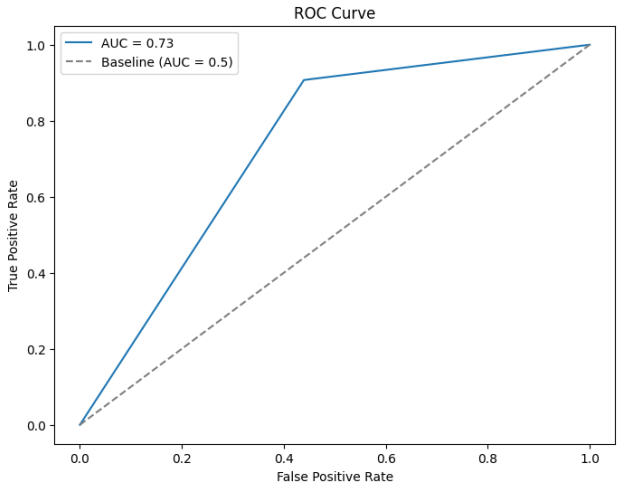

Fig. 12: ROC Curve

Table 3: Final SAGE Test Performance

Metric	Score
Precision	0.674
Recall	0.907
F1 score	0.773
ROC AUC Score	0.734

All three models we pretty close in terms of performance. It's important to note that the performance of different GNN architectures can vary depending on the specific dataset, graph structure, and task at hand. Though, some points could be made. Firstly, SAGE has quite much more parameters rather than the other two GNNs, something that made it a lot of slower to train. This extra capacity of the model did not have any positive impact on the results. Considering the comparable performance of the three models based on the evaluation metrics, it appears that the simpler architectures, GCN and GAT, were successful in capturing the relevant information and relationships required for the task at hand. It is worth noting that SAGE, unlike GCN and GAT, explicitly concatenated the node features with those of its neighbors during the aggregation process. This feature concatenation can be advantageous when the node features contain crucial information or when the characteristics of the central node significantly impact the link prediction task. In this particular scenario, it seems that utilizing the neighborhood's information alone was sufficient to achieve accurate predictions. The inclusion of explicit node feature concatenation, as done in SAGE, did not provide any noticeable improvement in performance.

In the context of link prediction, when the recall is higher than precision, it suggests that the models are better at capturing the true positive links that actually exist in the network. However, this may come at the cost of including some false positive predictions, meaning that the models might identify links that do not actually exist.

References

[1] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008. 6

[2] Muhan Zhang, Pan Li, Yinglong Xia, Kai Wang, and Long Jin. Revisiting graph neural networks for link prediction. 2020. 6

[3] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 7

[4] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017. 8

[5] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017. 8