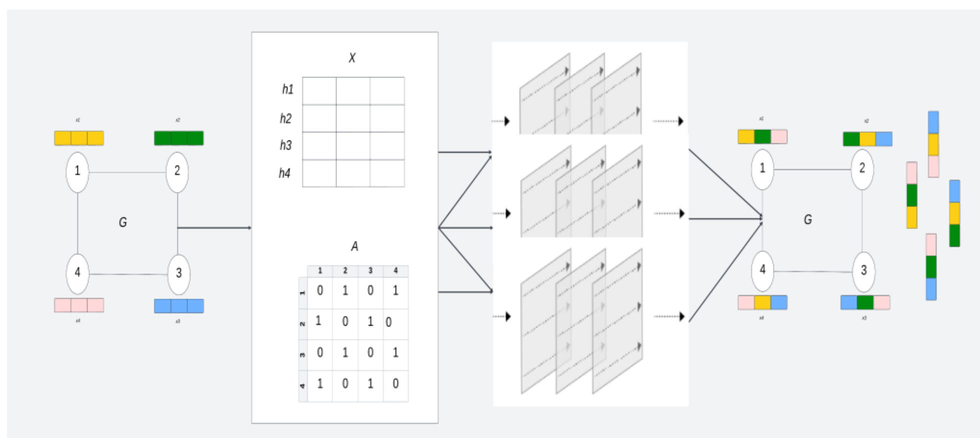
University of
South-Eastern Norway

www.usn.no

FMH606 Master's Thesis 2024

Industrial IT and Automation

Graph Neural Networks for outlier detection



Sarthak Lamsal

Faculty of Technology, Natural sciences and Maritime sciences
Campus Porsgrunn

University of South-Eastern Norway

www.usn.no

Course: FMH606 Master's Thesis, 2024

Title: Graph Neural Networks for outlier detection

Number of pages: 113

Keywords: Graph neural network, deep learning, machine learning, outlier detection, anomaly detection, graph representation, graph embedding, message passing, graph attention, graph convolution

Student: Sarthak Lamsal

Supervisor: Leila Ben Saad

Summary:

The thesis extensively explored state-of-the-art Graph Neural Networks (GNNs) for node-level outlier detection within graph data. A comprehensive review of various GNN architectures and outlier detection algorithms was conducted. Using PyTorch and the PyGOD library, the performance of four node-level outlier detection algorithms, DOMINANT, AnomalyDAE, CoLA, and GAAN was evaluated on the Cora and CiteSeer datasets, which were manually injected with 50 node-level outliers.

The models were assessed based on their AUC scores derived from ROC curves. AnomalyDAE and DOMINANT exhibited the highest performance, achieving AUC scores of ~0.81 and ~0.83 for the Cora dataset, and ~0.80 and ~0.83 for the CiteSeer dataset, respectively. CoLA followed closely with AUC scores of ~0.78 for Cora and ~0.80 for CiteSeer while GAAN demonstrated comparatively lower performance, with AUC scores of ~0.74 for Cora and ~0.78 for CiteSeer. Detection in node-level outliers where only 100 features were altered presented challenges across models, with variations observed in AUC scores. However, all models identified every node-level outliers where every features were altered.

The University of South-Eastern Norway takes no responsibility for the results and conclusions in this student report.

Preface

This work was written as a part of the FMH606 2024 Master's thesis course at the University of South-Eastern Norway (USN), campus Porsgrunn, Norway from 15th of January to 15th of May 2024. It is the final assessment of the Industrial IT and Automation (IIA), Master of Science program at USN under the department of Electrical Engineering, Information Technology and Cybernetics at campus Porsgrunn. The work was done on the study of Graph Neural Network and its application on Outlier Detection under the supervision of Associate Professor Leila Ben Saad. The task description provided is included in Appendix A.

I would like to express my sincere gratitude to my supervisor Associate Professor Leila Ben Saad for her continuous support, feedback, and positive encouragement. Her guidance has played a pivotal role in shaping this work. Also, I am equally thankful to the university for giving me the topic of my interest to work and do the research. Lastly, this work would not have been possible without the blessings of my parents, sister, and wife. I am also indebted and grateful to all the great souls that have contributed directly and indirectly to writing this thesis.

Porsgrunn, May 15, 2024

Sarthak Lamsal

Contents

Nomenclature vi
List of Symbols viii
List of Figures x
List of Tables xiii
1 Introduction 1
1.1 Background 1
1.2 Research gap and motivation 2
1.3 Objectives 2
1.4 Report Structure and Outline 3
2 An Overview of Graph Theory and GNNs 4
2.1 Graph Basics 4
2.1.1 Graph Connectivity and Representation 7
2.2 Graph Neural Networks 9
2.2.1 Learning Process 10
2.2.2 Applications of GNN 12
2.2.3 Challenges in GNN 13
2.3 Review of GNN Architectures 13
2.3.1 Graph Convolutional Network (GCN) 14
2.3.2 Graph Attention Network (GAT) 16
2.3.3 Graph Sample and Aggregation (GraphSAGE) 20
2.3.4 Graph Autoencoder (GAE) 21
2.3.5 Other GNNs 23
3 Review of GNN Algorithms for Outlier Detection 24
3.1 Outliers in GNNs 24
3.2 Outlier Detection 25
3.2.1 Review of Outlier Detection Algorithms in GNNs 26
3.3 GNN-based Node-level Outlier Detection in Static Graph 27
3.3.1 GCN-based method 27
3.3.2 GCN-based GAE method 29
3.3.3 GAT-based GAE method 31
3.3.4 Other GNN-based algorithms for outlier detection 32
3.4 Summary of GNN-based outlier detection methods 34
4 Implementation of GNN Algorithms for Outlier Detection 35
4.1 Introduction to PyTorch 35
4.1.1 Introduction to PyTorch Geometric 35
4.2 Introduction to PyGOD 36
4.3 Datasets 37
4.3.1 Cora Dataset 37
4.3.2 CiteSeer Dataset 38
4.3.3 Injection of outliers in the datasets 39
4.4 Model Implementation 42
4.4.1 Outlier Detection with DOMINANT 43
4.4.2 Outlier Detection with AnomalyDAE 44
4.4.3 Outlier Detection with GAAN 44

4.4.4 Outlier Detection with COLA *45*

5 Evaluation and Comparison of GNN-based algorithms **46**

5.1 Results of DOMINANT **46**
5.2 Results of AnomalyDAE **50**
5.3 Results of GAAN **53**
***5.4* Results of CoLA** **56**
5.5 Comparison of results **60**
5.6 Discussion **61**

6 Conclusion and Future work **63**

References **65**

Appendices **69**

Nomenclature

ALARM	Deep multi-view framework for anomaly detection
AnomalyDAE	Dual Autoencoder for Anomaly Detection on Attributed Networks
ARGA	Adversarially Regularized Graph Autoencoder for graph embedding
AUC	Area Under ROC Curve
BCE	Binary Cross-Entropy
CoLA	Contrastive self-supervised Learning
CNN	Convolutional Neural Network
DOMINANT	Deep Anomaly Detection on Attributed Network
FPR	False Positive Rate
GAE	Graph Auto Encoder
GAN	Generative Adversarial Network
GAAN	Generative Adversarial Attributed Network Anomaly Detection
GAT	Graph Attention Network
GCN	Graph Convolution Network
GIN	Graph Isomorphism Network
GNN	Graph Neural Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
LINE	Large-scale Information Network Embedding
LOF	Local Outlier Factor
ML	Machine Learning
MLP	Multi-Layer Perceptron
MPNN	Message Passing Neural Network
MSE	Mean Squared Error
ResGCN	Residual GCN
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
ResNet	Residual Network
PyG	PyTorch Geometric
PyGOD	Python Graph Outlier Detection
STrGNN	Structural-Temporal Graph Neural Network

TCN	Temporal Convolutional Network
TGN	Temporal Graph Network
TPR	True Positive Rate
USN	University of South-Eastern Norway
VAE	Variational Auto Encoder
VGAE	Variational Graph Auto Encoder

List of Symbols

G	Plain graph
G	Attributed graph
$G_{(t)}$	Dynamic graph
V	Set of vertices or nodes of plain or attributed graph
E	Set of edges of plain or attributed graph
$V_{(t)}$	Set of vertices or nodes of dynamic graph at time t
$E_{(t)}$	Set of edges of dynamic graph at time t
$\mathbf{X}$	Feature matrix of attributed graph
x	Transformed feature vector of attributed graph
$\mathbf{X}_{(t)}$	Feature matrix of dynamic graph at time t
u, v	Nodes
$\mathcal{N}$	Neighborhood
$\mathbf{A}$	Adjacency matrix of attributed graph
$\widehat{\mathbf{A}}$	Normalized adjacency matrix
$\mathbf{D}$	Degree matrix of attributed graph
$\widehat{\mathbf{D}}$	Modified degree matrix of attribute graph
$\mathbf{I}$	Identity matrix
d	Number of features of node
$\mathbf{W}$	Learnable weight matrix

$\mathbf{H}^{(k)}$	Feature Matrix at layer k
$\|\|$	Concatenation
e_{uv}	Attention score between node u and v
α_{uv}	Normalized attention score between node u and v
a	Learnable attention vector
σ	Non-linear activation function
α_{uv}	Normalized attention score between node u and v
$\mathbf{Z}$	Latent space representation
h_v^k	Feature vector of node v at layer k

List of Figures

Figure 1: A toy example of plain graph G consisting of four nodes (1,2,3 and 4). Here, node 1 and 2 are connected by edge e1, node 2 and 3 by edge e2, node 3 and 4 by edge e3 and node 4 and 1 by edge e4. $V = \{1,2,3,4\}$ is the set of nodes and $E = \{e1,e2,e3,e4\}$ is the set of edges for a plain graph G 5

Figure 2: A toy example of undirected and directed edge. Edge e1 connecting node 1 and 2 is an undirected edge, edge e2 connecting node is an undirected edge. In e2, node 4 is a source node and node 3 is the destination node. 6

Figure 3: A toy example of unweighted and weighted edge. Edge e3 connecting nodes 5 and 6 is an unweighted edge, edge e4 connecting nodes 7 and 8 with weight 1 is a weighted edge. . 6

Figure 4: A toy example of attributed graph G consisting of four nodes 1,2,3,and 4. Here, *x1*, *x2*, *x3*, and *x4* are the transformed node feature vectors for nodes 1,2,3, and 4 respectively.... 7

Figure 5: Adjacency matrix **A** for attributed graph *G*. For existence of edge between two nodes, the corresponding element in the matrix **A** is populated with 1 or else 0. The diagonal elements would be 1 if self-loops in all the nodes are present. 8

Figure 6: Feature matrix **X** for an attributed graph *G* with nodes 1,2,3,4, and their respective transformed node feature vectors *x1*, *x2*, *x3* and *x4*. 9

Figure 7: Pictorial representation of GNN in [1] consisting of 3 layers building GNN blocks where graph is fed as input to produce transformed graph as an output 10

Figure 8: GCN represented as image in [20]. Input is a graph with nine nodes and thirteen edges. Two hidden layer followed by activation function ReLU are GCN layers where graph convolution happens in the input graph. 14

Figure 9: GAT architecture with attention mechanism in [24]. Attention coefficients ij for all the neighboring nodes of node x_i are calculated. Self-attention coefficient of node x_i and neighborhood attention coefficient for all the neighboring edges, y_{ij} are calculated and combined where softmax activation function is applied to obtain all the attention coefficients ij of all nodes x_{ij}. 18

Figure 10: Pictorial representation of how attention coefficients are utilized in GAT taking node 1 of the graph as an example in [25] 19

Figure 11: A general GAE architecture, comprising of an encoder and decoder with **X** as the input to the encoder and **X′** the output generated by the decoder. **Z** is the latent space representation of the input **X**. Typically input is adjacency matrix, feature matrix or both.... 22

Figure 12: Examples of outliers in a graph in [37]. Nodes A and C are node-level outliers, nodes A and B are edge-level outliers. Node A is both node-level and edge-level outlier. 25

Figure 13: CoLA framework presented in [46] 29

Figure 14: Architecture of DOMINANT shown in [58]. DOMINANT has one attributed network encoder made of GCN where adjacency matrix **A** and feature matrix **X** of input graph is fed which produces latent space representation matrix **Z** as the output. Through structure

reconstruction decoder and attribute reconstruction decoder, DOMINANT produces estimated adjacency matrix **A** and estimated feature matrix **X**. 30

Figure 15: AnomalyDAE architecture presented in [49]. The framework is made up of two autoencoders, structure autoencoder and attribute autoencoder. Reconstructed adjacency matrix **A** and feature matrix **X** are calculated respectively from structure decoder and attribute decoder which are combinedly used to calculate the anomaly score and predict the outlier. . 32

Figure 16: GAAN framework in [50]. GAAN consists of three parts, generator, encoder and discriminator. Generator uses noise level to produce approximated node feature matrix $\mathbf{X'}$, encoder transforms both original feature matrix **X** and generated feature matrix $\mathbf{X'}$ into latent space **Z** and $\mathbf{Z'}$. Discriminator calculates loss from two latent space inputs and assigns the loss scores to all the nodes. 34

Figure 17: Graphical representation of Cora dataset with 2708 nodes and 10556 edges. Different colors indicate the nodes belonging to 7 different class of the dataset. Labels of classes are indicated in numbers from 0 to 6. 38

Figure 18: Graphical representation of CiteSeer dataset with 3327 nodes and 9104 edges. Different colors indicate the nodes belonging to 6 different class of the dataset. Labels of classes are indicated in numbers from 0 to 6. 39

Figure 19: A graphical representation of Cora dataset after injecting 128 outliers (50 contextual, 78 structural and 2 both). Different colors denote different types of nodes, inliers and types of outlier nodes. 41

Figure 20: A graphical representation of Cora dataset after injecting 128 outliers (50 contextual, 80 structural and 2 both). Different colors denote different types of nodes, inlier and types of outlier nodes. 41

Figure 21: ROC Curve for *model_Dominant* for contextual outlier predictions in Cora dataset containing injected outliers. TPR and FPR of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination. 48

Figure 22: ROC Curve for *model_Dominant* for contextual outlier predictions in CiteSeer dataset containing injected outliers. TPR and FPR of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination. 49

Figure 23: ROC Curve for *model_AnomalyDAE* for contextual outlier predictions in Cora dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination. 51

Figure 24: ROC Curve for *model_AnomalyDAE* for contextual outlier predictions in CiteSeer dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination. 53

Figure 25: ROC Curve for *model_GAAN* for contextual outlier predictions in Cora dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted

against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination. 54

Figure 26: ROC Curve for *model_GAAN* for contextual outlier predictions in CiteSeer dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination. 55

Figure 27: ROC Curve for *model_CoLA* for contextual outlier predictions in Cora dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination. 57

Figure 28: ROC Curve for *model_CoLA* for contextual outlier predictions in CiteSeer dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination. 59

Figure 29: Line plots for true predictions of models, Dominant, AnomalyDAE, GAAN, and CoLA for contextual outlier detections in Cora and CiteSeer dataset carried out for 10 experiments keeping threshold value top 50 outliers 60

List of Tables

Table 1 Summary of GNN based node-level outlier detection methods 34

Table 2 Summary of datasets used... 42

Table 3 Status of outliers predicted by DOMINANT model in Cora dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments. 47

Table 4 Status of outliers predicted by DOMINANT model in CiteSeer dataset with outliers injected. The model is run for 10 times in the dataset. Average is the average value of each column for 10 experiments. 49

Table 5 Status of outliers predicted by AnomalyDAE model in Cora dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments. 50

Table 6 Status of outliers predicted by AnomalyDAE model in CiteSeer dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments. 52

Table 7 Status of outliers predicted by GAAN model in Cora dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments. 54

Table 8 Status of outliers predicted by GAAN based model in CiteSeer dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments. 56

Table 9 Status of outliers predicted by CoLA model in Cora dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments. 58

Table 10 Status of outliers predicted by CoLA model in CiteSeer dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments. 59

Table 11 True Positive and False Positive of models for contextual outlier detection in Cora and CiteSeer dataset with outliers injected for threshold value of top 50 outliers 60

Table 12 AUC score of all the models in Cora and CiteSeer dataset for contextual outlier predictions 61

1 Introduction

This chapter presents the background information of graph theory, GNN and outlier detection. It includes the objectives of this work, research gap, motivation, and the structure of the report. It sets the stage for the exploration and analysis of GNNs for outlier detection in the subsequent chapters.

1.1 Background

As the digital connectivity of the world is rapidly growing, so does the complexity of data generated across various domains, prompting the need for sophisticated analytical tools and mechanisms. In the era of interconnected data, networks have been ubiquitous across various aspects, influencing everything from social interactions to financial transactions and beyond. The surge in network-based data has naturally led to increased academic and practical interest in novel analytical methods that cater to the unique attributes of such data. The prevalence of graph-structured data in several essential applications, such as social networks, financial systems, molecular science, and sensor arrays, lays the foundation for significant research and development efforts in this area. Among the emerging frontiers in this space is GNN, a significant extension of deep learning approaches, shaped specifically to navigate and interpret the wealth of data inherent in graph structures. The complex network pattern of data can be modeled effectively using GNNs, which extend deep learning techniques to graph-based data, allowing for the extraction of intricate patterns and features not readily apparent through traditional data analysis methods [1], [2].

GNNs have proven particularly adept at discovering hidden patterns within network data, pertinent not only in understanding complex systems but also in identifying abnormalities that deviate from expected behavior—outliers that could signify critical security concerns. GNNs are capable of learning rich and transferable representations by aggregating information within the graph, thus offering unparalleled insights into the data's underlying structure. Traditional methods often treat data points as isolated instances, which limits their effectiveness when the data's structure and relationships are essential for analysis. These methods are typically designed for scenarios where data points are assumed to be independent and identically distributed. GNNs, on the other hand, represent a paradigm shift from these traditional methods in several keyways, data structure, aggregation scheme and message passing, relational inductive bias, learning from topology to name a few.

Simultaneously, outlier detection in graph data has emerged as a critical challenge, particularly in security-sensitive fields. The ability to identify anomalies in a wide range of disciplines can help uncover not limiting to fraudulent transactions, detect network intrusions, reveal social network manipulations, toxicity predictions, disease pathway analyses. It is in this context that GNNs are being explored as a viable solution for efficient outlier detection, distinguishing themselves from other analytical approaches in handling relational data and their adaptability to the unique properties of each application domain. The utilization of GNNs for outlier detection embraces the essence of both disciplines – the structural intuition provided by graph theory and the predictive power of neural networks. This integration is pivotal for enhancing

security measures and for fostering trust in the digital infrastructures that increasingly underpin our society [3].

Therefore, this master thesis seeks to tap into the synergy between GNNs and the pressing requirement for robust outlier detection mechanisms within graph-structured data. By implementing and examining the latest GNN algorithms, specifically tailored for anomaly detection, this research aims to offer a comprehensive overview of the current landscape through a methodical evaluation of these algorithms using state-of-the-art libraries such as PyTorch [4] and PyGOD [5], and by closely scrutinizing their performance within specific application-related datasets.

1.2 Research gap and motivation

The rapid evolution of digital connectivity has ushered in an era where vast amounts of interconnected data are generated across diverse domains, necessitating sophisticated analytical tools to break down its complexities. Within this landscape, GNNs have emerged as a promising avenue for analyzing graph-structured data, offering unparalleled capabilities in extracting elaborate patterns and features. However, while GNNs have demonstrated remarkable prowess in various applications, a critical research gap exists in their application to outlier detection within graph data.

The motivation to bridge this gap reduces from the clustering need for robust anomaly detection mechanisms in security-sensitive fields and beyond. Traditional outlier detection methods often falter in adequately capturing the relational intricacies inherent in graph data, thereby underscoring the relevance of GNNs in this domain. Moreover, as the prevalence of anomalies, ranging from fraudulent transactions to network intrusions, continues to pose significant threats, harnessing the potential of GNNs for outlier detection becomes imperative. By delving into this intersection between GNNs and outlier detection, this research endeavors to not only advance the theoretical understanding of GNNs but also to offer practical solutions that bolster the security and integrity of digital infrastructures. Through a systematic exploration of state-of-the-art algorithms, meticulous evaluation using real-world datasets, and the proposal of potential enhancements, this thesis aims to contribute to the enhancement of anomaly detection capabilities, thereby fostering trust in the digital ecosystem.

1.3 Objectives

The objectives of this master thesis are as follows:

1. Inspect the state-of-the-art of GNNs and understand the different GNN architectures.
2. Review the state-of-the-art of GNN algorithms for outlier detection.
3. Implement and study GNN algorithms for outlier detection using PyTorch and the Python library for graph outlier detection PyGOD
4. Evaluate the performance of GNN based algorithms for outlier detection in datasets related to a specific application.
5. Propose improvements of GNN based algorithms for outlier detection.
6. Prepare a detailed report of the research process, findings, and conclusions, including code and data used in the thesis work.

The task description of this master thesis is included in Appendix A.

1.4 Report Structure and Outline

The report is structured in the following way.

Chapter 2 presents an overview of graph theory and graph-based data. Here, the state-of-the-art inspection of GNNs is performed. The architecture of GNN is thoroughly studied along with its nuances and variations. This includes a detailed review of the foundational elements that constitute GNNs and outlines how various architectures have evolved to address the challenges specific to learning from graphs, highlighting key advancements, and identifying the architectural traits that have contributed to their successes in outlier detection and other graph-based tasks.

Chapter 3 delves into a systematic review of the state-of-the-art GNN algorithms specifically tailored for outlier detection encompassing a thorough analysis of the methodologies and frameworks that have been developed and employed in recent studies.

Chapter 4 focuses on the hands-on implementation and in-depth study of GNN algorithms for outlier detection using PyTorch, along with PyGOD.

Chapter 5 is dedicated to the evaluation of GNN-based algorithms' performance in detecting outliers within datasets relevant to a targeted application area. It also synthesizes and reflects on the findings gained from implementing and evaluating GNN-based outlier detection methods. Key points of discussion include the practicality of GNN models and their performances.

Chapter 6 delves into identifying and addressing potential shortcomings of current GNN-based algorithms in outlier detection. The aim is to propose methodological and technical improvements to GNN algorithms. The concluding chapter encapsulates the main contributions of the thesis, summarizing the key findings and reinforcing the significance of the research in advancing the understanding and application of GNNs for outlier detection.

2 An Overview of Graph Theory and GNNs

This chapter provides a comprehensive overview that is necessary to understand GNNs and their application to outlier detection. It begins by introducing graph theory and the elements of graph-based data, laying down the definitions, types of graphs, and properties that are foundational to subsequent discussions on GNNs. The chapter is divided into three sections. It starts with an introduction to graph theory and its associated aspects, and then the introduction to GNNs. Finally, the chapter discusses the state-of-the-art GNN architectures including an exploration of the most recent and impactful research in the area along with the applications and challenges associated with GNNs.

2.1 Graph Basics

Perhaps, graphs are everywhere, almost in every fabric of the cosmos. Graphs can be seen as a reflection of the intricate tapestry of interconnections that weave together the elements of universe. From the neurons in human brain to the social ties that bind us, the stars in the cosmos, and the cities on maps, everything is interconnected in a complex network of relations and dependencies. These connections pattern the raw fabric of chaos into ordered structures that can be analyzed, understood, and optimized. Translating this philosophical contemplation into the realm of mathematics, graph theory emerges as the study of these patterns. A graph in this context is a collection of points, called vertices or nodes, connected by lines, known as edges. This simple construct is deceptively powerful, capable of representing virtually any system where a set of discrete entities maintains some form of relationship with one another [2], [6].

In the language of graph theory, relationships become tangible, quantifiable, and open to inspection. Each node can represent an individual entity, and each edge can embody the relationship between entities, varying in nature – social, biological, computational, or conceptual. By mapping out the connections, graph theory enables to see a clearer picture of the underlying structures governing complex systems. In essence, graph theory gives form to the abstract concept of interconnectedness, allowing to navigate and make sense of the complex networks that define the world. It is a bridge between the philosophical idea of unity and the practical need to comprehend and manage the interconnected systems that underlie the essence of life and technology.

A plain graph G can be simply represented mathematically as $G = (V, E)$, where V represents the set of vertices or nodes, and E represents the set of edges. Figure 1 shows an overview of a typical plain graph.

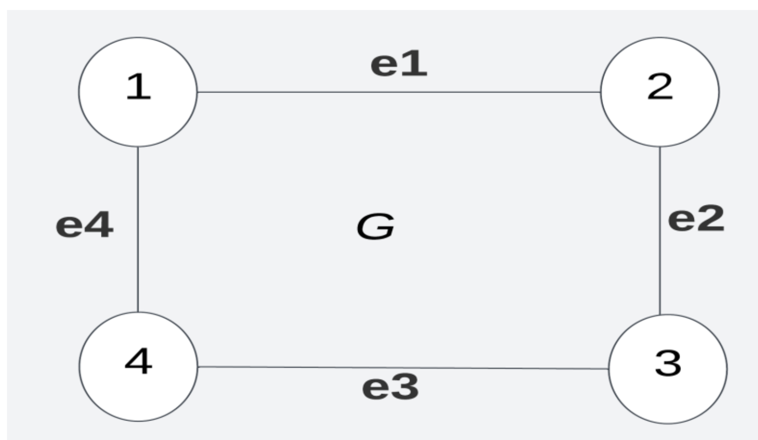


Figure 1: A toy example of plain graph G consisting of four nodes (1,2,3 and 4). Here, node 1 and 2 are connected by edge e1, node 2 and 3 by edge e2, node 3 and 4 by edge e3 and node 4 and 1 by edge e4. $V = \{1,2,3,4\}$ is the set of nodes and $E = \{e1,e2,e3,e4\}$ is the set of edges for a plain graph G

Nodes, also called vertices, represent entitles in the graph, such as people in a social network, paper or article in a citation network, cities in a transportation network, etc. Each node can have an identifier, and they can also carry attributes such as features or labels, commonly known as node features. Nodes are called plain nodes if they do not carry any additional information or attributed nodes if they carry additional information or attributes. In figure 1, the nodes, 1,2,3 and 4 indicate four nodes of a plain graph G.

Edges, on the other hand, represent relationships or connections between nodes. Edges can have various properties and types, depending on how they connect the nodes and whether they carry additional information. Edges can be undirected or directed based on the direction. An undirected edge simply represents a bidirectional relationship between two nodes without any inherent direction whereas a directed edge has a specific direction from one node to another representing a one-way relationship or flow from a source node to a target node.

Similarly, edges can also be weighted or unweighted based on the additional information carried. A weighted edge carries a numerical weight or value which can represent various properties such as distance or cost. Unweighted edge, on the other hand, does not contain any such numerical weight or value. Additionally, there are other instances of edges, such as multiple edges and self-loops. Two or more edges connecting the same pair of nodes are multiple edges and an edge that connects a node to itself is known as self-loops. Figure 2 illustrates undirected and directed edge and figure 3 illustrates unweighted and weighted edge.

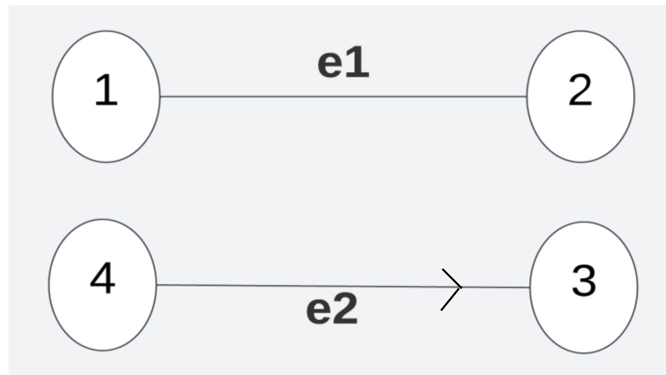


Figure 2: A toy example of undirected and directed edge. Edge e1 connecting node 1 and 2 is an undirected edge, edge e2 connecting node is an undirected edge. In e2, node 4 is a source node and node 3 is the destination node.

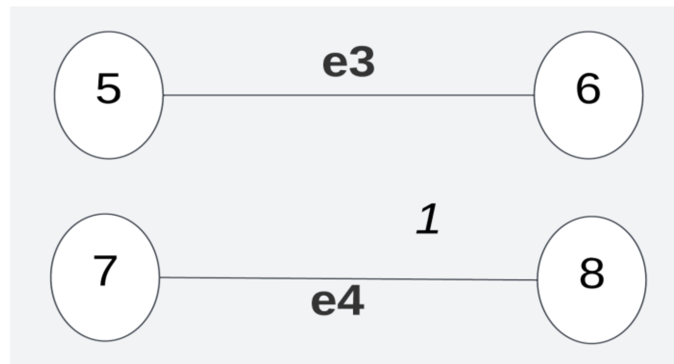


Figure 3: A toy example of unweighted and weighted edge. Edge e3 connecting nodes 5 and 6 is an unweighted edge, edge e4 connecting nodes 7 and 8 with weight 1 is a weighted edge.

An attributed graph, consequently, is another form of graph where nodes, edges or both are associated with additional features or attributes. Attributed graphs can represent enriched networks such as knowledge graphs, road networks, etc. These graphs are generally represented as $G = (V, E, \mathbf{X})$, where V represents the set of nodes, E represents the set of edges and $\mathbf{X}$ represents the feature matrix. Figure 4 is the illustration of attributed graph. In figure 4, the transformed node feature vectors *x1*, *x2*, *x3*, and *x4* are stacked row wise in the feature matrix $\mathbf{X}$ of the attributed graph G.

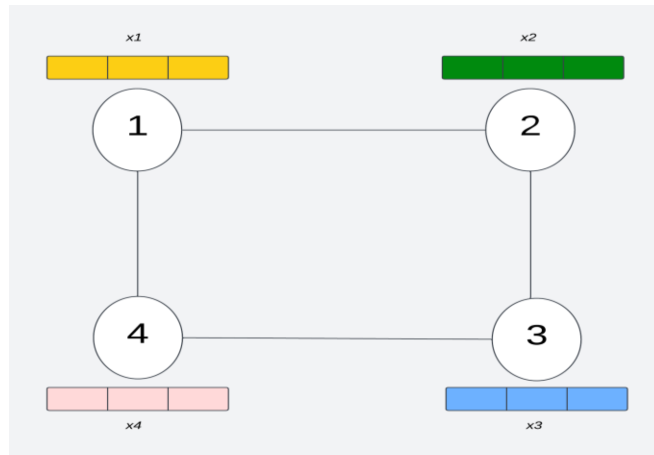


Figure 4: A toy example of attributed graph G consisting of four nodes 1,2,3,and 4. Here, *x1*, *x2*, *x3*, and *x4* are the transformed node feature vectors for nodes 1,2,3, and 4 respectively.

Based on the dynamism, a graph can be represented as static or dynamic graph. In static graph, nodes, edges, and attributes do not change or evolve and remains unchanged over time. Dynamic graph, also called spatial-temporal graph, however, is a graph where graph's structure including nodes, edges, or attributes evolve or change over time. A dynamic graph $G_{(t)}$ can be represented mathematically as $G_{(t)} = (V_{(t)}, E_{(t)}, \mathbf{X}_{(t)})$, where $V_{(t)}$ is the set of nodes at time t, $E_{(t)}$ is the set of edges at time t and $\mathbf{X}_{(t)}$ is feature matrix at time t.

2.1.1 Graph Connectivity and Representation

There are several important parameters used for graph connectivity and representation. Some of them are discussed below [1].

Adjacency Matrix (A): An adjacency matrix **A** is a common mathematical representation of a graph's structure, describing which nodes are connected to each other. It is basically a matrix representation of the graph and is a square matrix of size $|V|$ x $|V|$, where $|V|$ is the number of nodes in the graph. Each element $\mathbf{A}_{uv}$ in the matrix represents the connection between nodes u and v. In weighted graphs, the matrix includes the weight of the connections; in unweighted graphs, it is a binary matrix meaning if there is an edge from node u to v then $\mathbf{A}_{uv} = 1$; otherwise, $\mathbf{A}_{uv} = 0$. For undirected graphs, **A** is symmetric, meaning $\mathbf{A}_{uv} = \mathbf{A}_{vu}$. Figure 5 shows the adjacency matrix **A** for a graph G. Figure 5 has an attributed graph G with four nodes. **A** is the corresponding adjacency matrix of graph G, a symmetric matrix, where each element corresponds to the edge information. Graph G being an unweighted graph, the element in matrix **A** is filled with binary values 0 and 1 corresponding to the absence and presence of edges between two nodes respectively. For weighted graph, binary values are replaced by the corresponding weights of the edges.

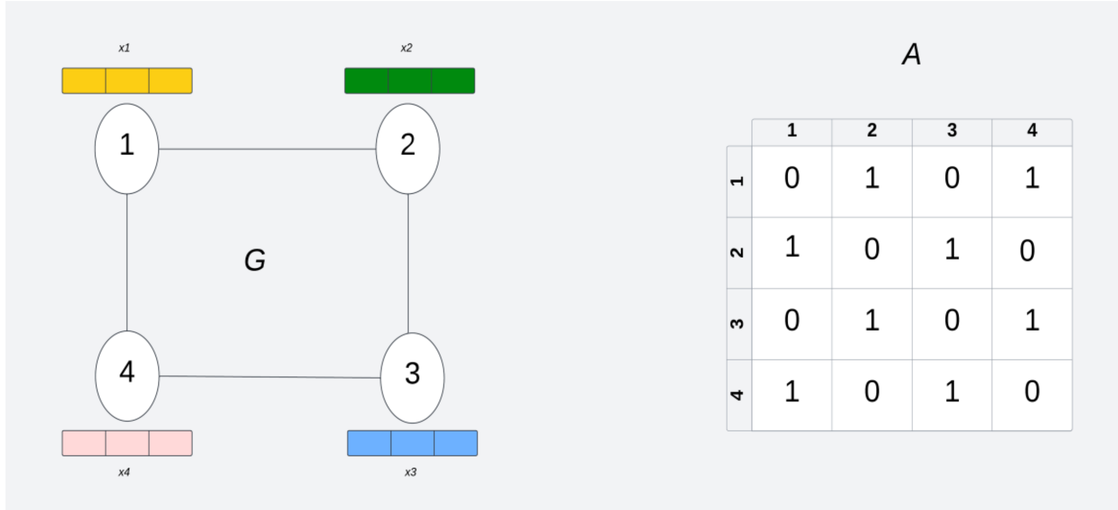


Figure 5: Adjacency matrix **A** for attributed graph *G*. For existence of edge between two nodes, the corresponding element in the matrix **A** is populated with 1 or else 0. The diagonal elements would be 1 if self-loops in all the nodes are present.

Likewise, the degree matrix **D** is a diagonal matrix, where the diagonal entry $\mathbf{D}_{uu}$ represents the degree (number of edges) of node *u* and is essential for normalization process. $\mathbf{D}_{uu}$ is calculated as $\mathbf{D}_{uu} = \sum_{v=1}^{|V|} \mathbf{A}_{uv}$, where $\mathbf{A}_{uv}$ is an adjacency matrix with nodes u and v.

The normalized adjacency matrix $\widehat{\mathbf{A}}$ is computed by first augmenting the adjacency matrix **A** with an identity matrix **I** to account for self-loops, resulting in the matrix $\mathbf{A}' = \mathbf{A} + \mathbf{I}$. Self-loops are edges in a graph that connect nodes to themselves.

Next, the degree matrix **D** is modified by adding an identity matrix **I** to each diagonal element, which increase each degree by one. This modified degree matrix is denoted as $\widehat{\mathbf{D}}$ and is calculated as $\widehat{\mathbf{D}} = \mathbf{D} + \mathbf{I}$.

To normalize the adjacency matrix, $\widehat{\mathbf{A}}$ is computed as $\widehat{\mathbf{A}} = \widehat{\mathbf{D}}^{-1/2}\, \mathbf{A}'\, \widehat{\mathbf{D}}^{-1/2}$, where $\widehat{\mathbf{D}}^{-1/2}$ is the inverse of square root of the modified degree matrix $\widehat{\mathbf{D}}$. By normalizing the adjacency matrix in this manner, the resulting matrix $\widehat{\mathbf{A}}$ ensures that the learned graph representations are more stable and less influenced by nodes with different connectivity levels.

Feature Matrix (X): A feature matrix **X** is a matrix, where each row corresponds to a node and each column corresponds to a feature. Each row of the feature matrix represents the feature vector for a node, containing the node's attributes such as labels, properties, or other data relevant to the graph. Thus, **X** is a matrix of size $|V|$ x d, where $|V|$ is the number of nodes, and d is the number of features. This matrix provides input data to GNN, representing each node's characteristics. It is the feature matrix in GNNs that undergoes transformations and aggregations to learn new representations of nodes, incorporating graph structure and feature information.

Feature Vector (*x*): A transformed feature vector *x* for node *u* is represented by the uth row of the feature matrix **X**. A feature vector captures the attributes of the node, which may include numerical, categorical, or binary values. Each row of the feature matrix **X** is a transformed

feature vector representing a single node's features. Feature vectors provide a representation of each node based on its characteristics or attributes. These features might include node attributes such as color, type, label, or any other information relevant to the specific graph and application. Figure 6 shows the concept of feature vector and feature matrix for an attributed graph $G = (V,E,\mathbf{X})$. The transform of feature vector x of all nodes stacked row wise produces the feature matrix $\mathbf{X}$ of graph G.

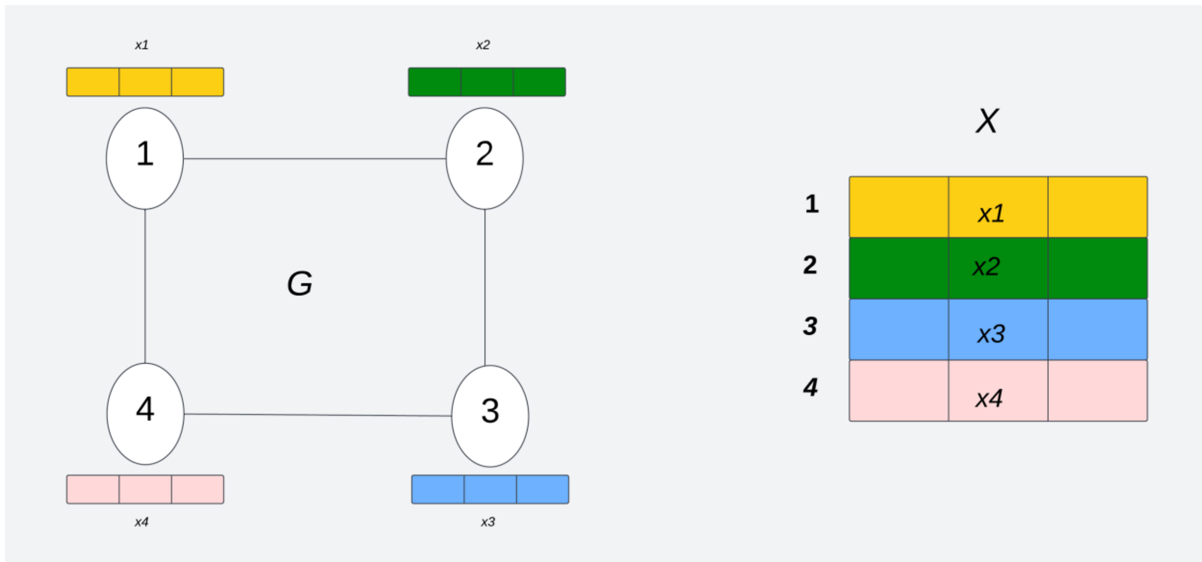


Figure 6: Feature matrix $\mathbf{X}$ for an attributed graph G with nodes 1,2,3,4, and their respective transformed node feature vectors $x1$, $x2$, $x3$ and $x4$.

These mathematical representations and concepts form the basis of graph data structures and their analysis in the context of GNNs and other graph-based algorithms are covered in the subsequent chapters.

2.2 Graph Neural Networks

Building on the philosophical understanding of interconnectedness, graph theory finds its tech-savvy incarnation in GNNs [6]. Just as graphs illustrate the complex web of relationships in nature and society, GNNs capture and harness these relationships within the realm of data.

GNNs are an innovative blend of graph theory and machine learning. While traditional neural network architectures excel at tasks with structured, grid-like data (such as images for Convolutional Neural Network (CNN) [7] or sequential text for Recurrent Neural network (RNN) [8]), GNNs are designed for the irregular, complex structures that graph data represents. They learn from the topological structure of graphs, accounting for the heterogeneous and rich relationships between nodes – a reflection of the real-world's complexity.

In the GNN paradigm, nodes aggregate information from their neighbors through neural network layers, effectively allowing for the direct application of machine learning to graphs.

This aggregation captures not only the individual node features but also the global structure of the data, enabling the model to infer deep insights about the nature of each node within the context of its surrounding network. Just as individual beings find identity in their social and environmental contexts, GNNs understand each node by its relational position and interactions within the network.

Furthermore, in applying GNNs to outlier detection, for instance, this powerful concept of GNNs is harnessed to identify the nodes or patterns that stand out as atypical or unexpected. GNNs, therefore, represent a synergy between age-old wisdom and cutting-edge technology, enabling a deeper understanding and interaction with the myriad networks that underpin the fabric of existence.

Now shifting the focus from philosophical point of view to scientifically and mathematically, GNNs are sophisticated computational models that operate on data represented as graphs. While traditional neural networks process fixed-size inputs and generate fixed-size outputs, which are not naturally adaptable to the variable-sized graphs characterized by an arbitrary number of nodes and connections, GNNs, however, are specifically engineered to handle this variability, capable of learning from data that is inherently relational and interconnected. Figure 7 is the pictorial representation of GNN.

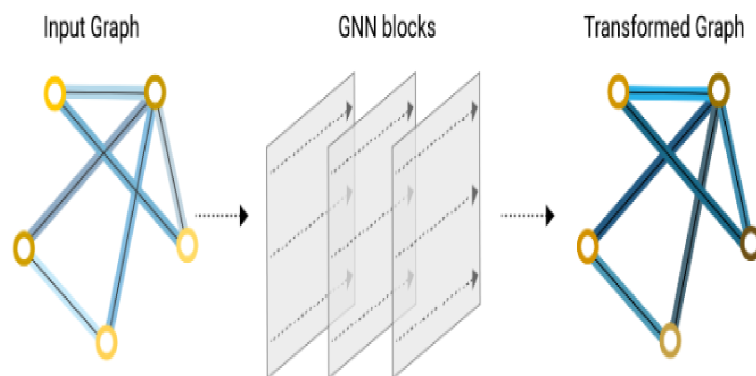


Figure 7: Pictorial representation of GNN in [1] consisting of 3 layers building GNN blocks where graph is fed as input to produce transformed graph as an output.

2.2.1 Learning Process

GNNs [26] extend the concept of neural networks to graphs by using the notion of message passing and neighborhood aggregation. The fundamental idea of GNNs is to learn suitable representation of graph data for a neural network. This is also called representation learning. Using all the information about the graph including the node features and the connections stored in an adjacency matrix, the GNNs output new representations which are also called embeddings for each of the nodes. These node embeddings contain the structural as well as the feature information of the other nodes in the graph. It means each node knows something about

the other nodes, the connection to this node and its context in the graph. The embeddings can finally be used to perform the predictions [1], [6], [9].

In a nutshell, for each node, GNNs aggregate information from its neighbors using an aggregation function. The aggregated information is then combined with the node's attributes using a transformation function. This process is repeated over several iterations or layers, allowing information to propagate across the network. This process is known as message passing. Message passing is a key concept in GNNs that essentially describes the process of transferring messages across nodes in the graph through their connections (edges).

The aggregation step includes a new representation for each node by combining its own features with the features of its neighbors. For a given node v, the aggregated feature vector $h_v^{(agg)}$ is computed as:

$$h_v^{(agg)} = aggregate\left((h_u^{(k-1)} : u \in \mathcal{N}(v))\right) \tag{1}$$

where, $h_u^{(k-1)}$ is the feature representation of node u at layer (k-1), u are the neighbor nodes belonging to the neighborhood $\mathcal{N}(v)$ of node v, and *aggregate* is the aggregation function. The neighborhood for the node v is defined as $\mathcal{N}(v) = \{ u \in V \mid (u, v) \in E \}$.

The *aggregate* function typically involves normalized summing, averaging, finding the maximum of the neighboring features or more advanced methods such as the use of neural network as well. Aggregation is often followed by normalization to ensure stable and consistent updates. A common normalization strategy is to normalize the adjacency matrix (**A**) to obtain $\widehat{\mathbf{A}}$. These aggregated neighbor features are then combined with the node's current features using the combine function. Mathematically, the kth layer of a GNN could be represented as:

$$h_v^{(k)} = combine^{(k)}\left(h_v^{(k-1)}, aggregate^{(k)}\left(\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\}\right)\right) \tag{2}$$

where, $h_v^{(k)}$ is the feature representation of node v at layer k, $h_v^{(k-1)}$ is the feature representation of node v at layer (k-1), $h_v^{(agg)}$ is the aggregated feature representation of neighbor nodes of node v and $combine$ is the function responsible for integrating the aggregated neighbor features with the node's current features.

In GNNs, the process of aggregating information from neighboring nodes and combining the aggregated information with the node's information is referred to pooling [1]. Pooling in GNNs is simply a technique used to compress the information of a graph or a node's neighborhood. Some common types of pooling are node pooling, edge pooling and graph pooling. In node pooling, the information is aggregated from a node and its neighboring nodes to produce a new representation of a node. Similarly, in edge pooling, the information is aggregated from the edges of a graph. This can involve pooling edge features or aggregating information from edges and their incident nodes as well. In graph pooling, pooling is often used to create a single representation of the entire graph. While node pooling is done at the node level, edge pooling can provide more nuanced insights into graph data by helping to capture important relationships and interactions within the graph from edge level and connectivity which analysis from the node level is not often guaranteed. The primary applications of pooling are feature aggregation

(aggregate the features of a node's neighbors to a single representation), graph summarization (create a summarized representation of the entire graph) and dimensionality reduction (reduce the dimensionality of the data).

The learning objective of GNNs is typically framed around node-level, edge-level, or graph-level tasks. Node-level tasks might involve classifying nodes into different categories, while edge-level tasks could include predicting the existence or properties of edges between nodes. Graph-level tasks often revolve around classifying entire graphs or predicting their properties.

For all these tasks, loss functions measure the discrepancy between the model's predictions and the true outcomes. During training, the loss is backpropagated through the network to update weights and improve the model's predictions. Backpropagation in GNNs accounts for the derivative of the loss with respect to the node representations, which further translates into gradient updates for the neural network parameters, moving in the direction that minimally reduces the loss.

2.2.2 Applications of GNN

Scientifically, GNNs have broad implications due to their ability to model relational data authentically. They allow the capturing of dependencies and interactions that would otherwise be lost in non-relational representations. This has led to far-reaching applications in fields like chemistry, where GNNs can predict molecular properties, bioinformatics for protein-protein interaction networks, social network analysis, and even infrastructure and urban planning using road and utility networks. From a mathematical lens, GNNs also present new frontiers in understanding how to optimally aggregate and transform information in complex systems. The ongoing development of GNN theory and methods continues to be an interdisciplinary effort, bringing together insights from graph theory, linear algebra, computer science, and statistical learning theory [3], [6], [26].

Some of the major applications of GNNs are discussed below:

- **Recommendation Systems:** GNNs are used to recommend products based on users' previous interactions and connections with other users [10].
- **Drug Discovery and Bioinformatics:** GNNs are applied in bioinformatics for modeling protein-protein interaction networks, gene regulatory networks, and drug-target interactions [11].
- **Financial Applications:** GNNs are used to analyze financial data modeled as graphs, such as transaction networks, for fraud detection, market analysis, and investment recommendations [12].
- **Traffic and Mobility:** GNNs are used to model traffic networks, such as road networks or transportation systems, for predicting traffic flow, optimizing routes, or managing mobility systems [13].
- **Cybersecurity:** GNNs are used to model and analyze network traffic as graphs for detecting anomalies or security threats, such as intrusion detection [14].

These are just a few examples of the many possible applications of GNNs. As the field continues to evolve, new applications are emerging in a variety of domains, where the

versatility and power of GNNs in processing and learning from graph-structured data are richly exploited.

2.2.3 Challenges in GNN

While GNNs have proven to be a powerful tool for learning from graph-structured data, they also present several challenges that needs to be addressed. These challenges arise from the complexity of graph data and the unique aspects of GNN architectures [6], [9], [15].

Some of the key challenges in GNNs are explained below.

- **Over-Smoothing:** Over-smoothing occurs when the node representations become increasingly similar as the number of layers in the GNN increases. This can lead to loss of distinctive features for nodes, making it difficult to differentiate between them.
- **Scalability:** Graphs can be large and complex, with millions or even billions of nodes and edges. Also, graphs come in different shape and size. Handling such large-scale graphs efficiently is a major challenge for GNNs.
- **Heterogeneity:** Many real-world graphs are heterogeneous, containing multiple types of nodes and edges with varying properties. Also, the non-Euclidean structure (no clear grid like structure) of graph makes the design of GNN architectures that can effectively handle such heterogeneity a challenge.
- **Dynamism:** Many graphs are dynamic, with nodes and edges being added or removed over time. Learning from such evolving graphs is challenging as the graph structure and features change over time.
- **Graph Sampling:** Efficiently sampling and selecting relevant subgraphs or neighbors for training is challenging, especially when dealing with large or dense graphs as poor sampling strategies can lead to biased or incomplete learning.
- **Isomorphism:** Graphs are inherently permutation invariant, meaning their structure does not change under reordering of nodes. This poses a challenge when using adjacency matrices directly in feed-forward networks because they are sensitive to the order of the nodes.

2.3 Review of GNN Architectures

GNNs over the past have been realized in several different architectures mainly designed to work with various types of graph-structured data and tasks. These architectures leverage the graph structure and its features in different ways. The state-of-the-art of GNNs is a rapidly evolving landscape with diverse architectures tailored to harness the power of graph data. These architectures address specific challenges such as scalability, inductive learning, expressiveness, and interpretability. The concept of neural networks for graphs started with work by Gori et al. in [16], where GNNs are introduced as a generalization of RNNs [8] to graphs. The idea was to update node features based on neighboring nodes using recurrent computations.

Message Passing Neural Networks (MPNNs) [17] are a framework that encompasses many GNN variants. Introduced by Gilmer et al in [17], MPNNs formalize GNN's operation into a

message passing phase, where edges deliver messages between nodes, and an update phase, where node representations are updated based on incoming messages. MPNNs provide a unifying framework that shows how different GNN architectures can be seen as specific instances of a general message-passing scheme. Each MPNN is characterized by how it defines the message functions and the update functions. The message function determines how information should be passed along edges, and the update function decides how the node's new state should be computed based on the aggregated messages. These functions often involve learnable parameters, allowing the MPNN to be trained end-to-end using gradient descent.

Some of the other prominent GNN architectures and their unique aspects are discussed in the following section.

2.3.1 Graph Convolutional Network (GCN)

GCN [18] is one of the most common GNN architectures and is based on convolutional operations on graph-structured data. It aggregates features from a node's neighbors, applies a linear transformation, and a non-linear activation function. GCNs extend the concept of convolution from regular grids (like images) to graph-structured data. They use a neighborhood aggregation approach where each node updates its representation by aggregating the representations of its neighbors. The key idea is to learn a transformation that is applied to the local neighborhood of a node. The simplicity and effectiveness of GCNs have made them a popular choice for node classification and link prediction tasks [18], [19], [20].

GCNs are a popular class of GNNs designed to handle graph data efficiently. Bruna et al. in [21] introduced the concept of spectral graph convolutional networks. These networks applied graph convolutions in the frequency domain using the graph Laplacian eigenvalues and eigenvectors. This work laid the foundation for the spectral approaches in GNNs. Kipf et al. introduced a simplified and scalable version of graph convolutions in [18]. In [18], two interesting ideas have been used. First, the aggregation is used for neighbor information as the normalized sum of the states. Additionally, it is incorporated with the update operation by adding a self-loop for a particular node including it into the summation meaning both update and aggregation is combined in one computation. The concept of GCN is shown pictorially in figure 8.

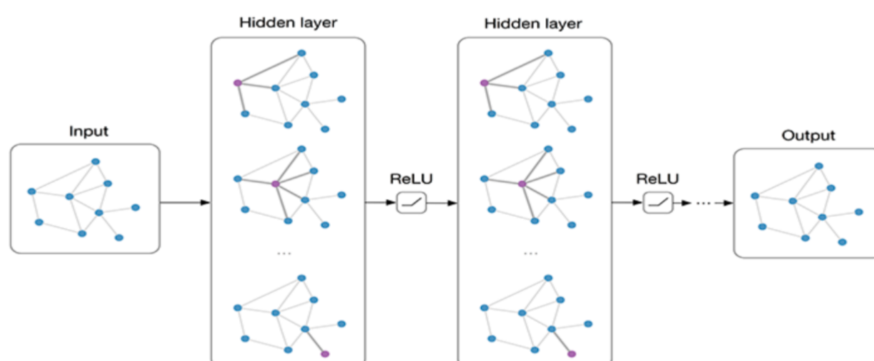


Figure 8: GCN represented as image in [20]. Input is a graph with nine nodes and thirteen edges. Two hidden layer followed by activation function ReLU are GCN layers where graph convolution happens in the input graph.

2.3.1.1 Learning Process

GCNs are designed to learn node representations through a process known as message passing or graph convolution. In each layer of a GCN, nodes aggregate information from their neighbors, transform it, and update their own features accordingly. The node-order is invariant or equivariant meaning the order of nodes are permutational invariant making the overall convolution node-order equivariant. GCNs perform graph convolutions by aggregating the features from the neighbors of a node and combining them with the node's own features [18]. The working process of GCN can be explained stepwise in the following ways.

Aggregation: The key idea behind GCNs is to perform convolutional operations directly on the graph structure. Unlike regular convolutions in grid-structured data (e.g., images), graph convolutions operate on irregular neighborhoods defined by the graph topology. For graph $G = (V, E, \mathbf{X})$ with feature matrix $\mathbf{X}$, the goal is to compute new feature representations for each node based on its local neighborhood. The graph convolution operation aggregates information from neighboring nodes and updates the node features using learnable parameters. For a node v, the aggregation in GCN is calculated as:

$$h_v^{(k+1)}=\sigma\left(\mathbf{W}^{(k)} \sum_{u\in\mathcal{N}(v)} \hat{\mathbf{A}}_{vu}\right) \tag{3}$$

where, $h_v^{(k+1)}$ is the updated feature vector for node v at layer k+1, $\mathbf{W}^{(k)}$ is the learnable weight matrix for layer k, $\hat{\mathbf{A}}_{vu}$ is the normalized adjacency matrix representing the connection between node u and v, $\mathcal{N}\ (v)$ is the neighborhood of node v and σ is the activation function to include non-linearity to the model.

In equation (3), the new updated representation of node v, $h_v^{(k+1)}$, is achieved by the convolution operation. The step $\sum_{u\in\mathcal{N}(v)} \hat{\mathbf{A}}_{vu}$ is where the feature vectors of all neighbors u of node v are aggregated and the learnable weight matrix $\mathbf{W}^{(k)}$ k transforms the aggregated features to a new updated feature vector for node v at layer k. This updated representation is passed through a non-linear activation function σ. This overall sums up the aggregation process in GCN.

Message Passing: GCNs typically employ a message passing scheme to propagate information across the graph. At each layer of the GCN, nodes exchange messages (information) with their neighbors, which are then aggregated to compute updated node representations. The equation (3) is performed for all the nodes present in the graph to obtain new node embeddings which contains the information aggregated from all the neighboring nodes. In general, the convolution operation or a message passing for a given layer in a GCN can be represented mathematically as:

$$\mathbf{H}^{(k+1)}=\sigma(\hat{\mathbf{D}}^{-1/2}\ \mathbf{A}'\ \hat{\mathbf{D}}^{-1/2}\ \mathbf{H}^{(k)}\mathbf{W}^{(k)}) \tag{4}$$

where, $\mathbf{H}^{(k)}$ is the feature matrix at layer k, with each row representing the transformed feature vector of a node, $\mathbf{A}' = \mathbf{A} + \mathbf{I}$ is the adjacency matrix with self-loops added (where $\mathbf{I}$ is the identity matrix), $\hat{\mathbf{D}}$ is the diagonal degree matrix of $\hat{\mathbf{A}}$, $\mathbf{W}^{(k)}$ is the learnable weight matrix at

layer k and σ is a nonlinear activation function. $\mathbf{H}^{(k+1)}$ is the updated feature matrix at the next layer.

Multi-Layer Perceptron (MLP): Each layer in GCN is responsible for aggregating information from the neighbors of each node and transforming it using a weight matrix and an activation function to generate new updated feature embeddings. The node feature representations are updated in each layer using linear transformations .This transformation is akin to the operations in a MLP but applied to the graph structure. This process continues through multiple layers, allowing information to propagate across the graph. The combination of linear transformations, non-linear activation functions, and weight matrices in each layer makes the GCN similar to an MLP in structure.

These three concepts, aggregation, message passing, and MLP, are central to the functioning of GCNs and enable the network to learn from graph-structured data. GCNs are trained using backpropagation and gradient descent methods to minimize a loss function. The loss function depends on the specific task, such as node classification or graph classification. During training, the model learns optimal parameters (e.g., weight matrices) to effectively propagate and aggregate information across the graph while minimizing the prediction error. In a classification task, the final layer's output can be used for node classification, graph classification, or other downstream tasks.

2.3.1.2 Limitations of GCNs

Some of the noticeable limitations of GCNs are discussed below [3], [19], [22].

- GCNs require knowledge of the complete graph for performing the convolution operation. which is very expensive computationally. New additions to the graph requires recalculation of the entire architecture parameters due to change in eigenvalues (λ) .
- GCNs use a fixed neighborhood size, which may not be optimal for different nodes or graphs.
- As the number of layers in a GCN increases, node features tend to become more similar (over-smoothed), leading to a loss of distinction between nodes.
- GCNs may not be expressive enough to differentiate between different graph structures due to their reliance on linear transformations and fixed aggregation functions.
- GCNs treat all neighbors equally during message passing, which may not capture the varying importance of different neighbors.
- GCNs leverage only transductive learning and cannot transfer knowledge from one domain to another.

2.3.2 Graph Attention Network (GAT)

GAT [23] uses attention mechanisms to weigh the importance of different neighbors when aggregating their features. This allows the network to focus more on relevant neighbors and less on less relevant ones. GATs introduce an attention mechanism to the aggregation step in GNNs. Nodes compute the coefficients of attention across their edges that indicate the importance of the neighboring node’s information. This allows the model to focus more on

relevant nodes and less on less relevant ones, enhancing the adaptability to different parts of the graph.

Veličković et al. in [23] introduced GATs, which applied attention mechanisms to graph data. This allowed the model to focus on specific neighbors more than others during message passing, providing a more flexible and interpretable approach compared to GCNs.

2.3.2.1 Learning Process

The learning process and working mechanism of GATs is summarized below [23]:

Attention Mechanism : GATs use an attention mechanism to compute a weight (attention score) for each neighbor of a node. The basic idea of attention mechanism is to additionally learn how important are the features of the neighboring nodes $u \in \mathcal{N}(v)$ for a node v. This importance is called the attention score or coefficient. Attention mechanism therefore allows the model to prioritize certain neighbors based on their importance. The attention score e_{uv} between node v and its neighbor u is computed as:

$$e_{uv} = \sigma\left(a^T[\mathbf{W}h_u || \mathbf{W}h_v]\right) \tag{5}$$

where, h_u and h_v are the feature vectors of nodes u and v, $\mathbf{W}$ is a learnable weight matrix, a is a learnable attention vector, $||$ denotes concatenation and σ is an activation function. Attention score essentially gives the weight of each neighbor fetching the information of how much attention should be paid to those specific nodes when updating the embedding.

The attention scores are often normalized using a softmax function over the neighbors of a node. This process of normalization is done to bring the attention coefficients of all the nodes in the same scale. A softmax function makes the value sum up to 1. It is done by :

$$\alpha_{uv} = softmax_v(e_{uv}) = \frac{\exp(e_{uv})}{\sum_{k \in \mathcal{N}(u)} \exp(e_{uk})} \tag{6}$$

where, α_{uv} is the normalized attention score between node u and node v and $\mathcal{N}(u)$ is the set of neighbors of node v.

There could be several different possibilities for calculating the attention coefficients. The explained approach is taken from [23] and is only one possibility. In [23], a shared single layer neural network is chosen. The input in this network are two transformed node feature vectors for an edge where the output indicates the importance of these nodes. This attention is effectively calculated for each node pair. The formula for full attention mechanism is expressed as :

$$\alpha_{uv} = = \frac{\exp\left(LeakyReLU(\overrightarrow{w_a^T}[\mathbf{W}h_u || \mathbf{W}h_v])\right)}{\sum_{k \in \mathcal{N}(u)} \exp\left(LeakyReLU(\overrightarrow{w_a^T}[\mathbf{W}h_u || \mathbf{W}h_k])\right)} \tag{7}$$

where, $\mathbf{W}h_u$ and $\mathbf{W}h_v$ are two node embeddings passed as inputs, $||$ the concatenation operation, $\overrightarrow{w_a^T}$ is a weight vector that is multiplied with $\mathrm{W}h_u \; || \; \mathrm{W}h_v$ when passed through a single layer neural network, *LeakyReLU* is an activation applied on each of the output to

emphasize the positive relationship between nodes and cutoff all the negative values and exp(.) is the softmax activation function.

Message Passing : The node features in GATs are then updated using a weighted sum of the neighbors' features as :

$$h_v' = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{uv} \mathbf{W} h_u\right) \tag{8}$$

where, h_v' is the updated node embedding for a node v, α_{uv} is the attention coefficient between node u and v. In equation 8, first, the node features h_u are transformed by multiplying them with shared weight matrix **W**. This learnable linear transformation ($\mathbf{W}h_u$) converts the node features into the higher-level features. The weight matrix W comes from the fully connected neural network, its input would be the shape of the node feature vector and the output would be the shape of the node embeddings. $\mathbf{W}h_u$ is multiplied with its respective attention coefficient α_{uv} and finally, this process is wrapped up by applying an activation function σ giving the updated node embedding h_v' for a node v . This is carried out for all the nodes in the graph where all the nodes now contain the information of their neighbors along with their own information. Essentially, feature vectors weighted with the importance (attention coefficient) for each node amplifies the important nodes and the less important ones are effectively suppressed. Figure 9 is the representation of GAT architecture that illustrates the attention mechanism used in GAT.

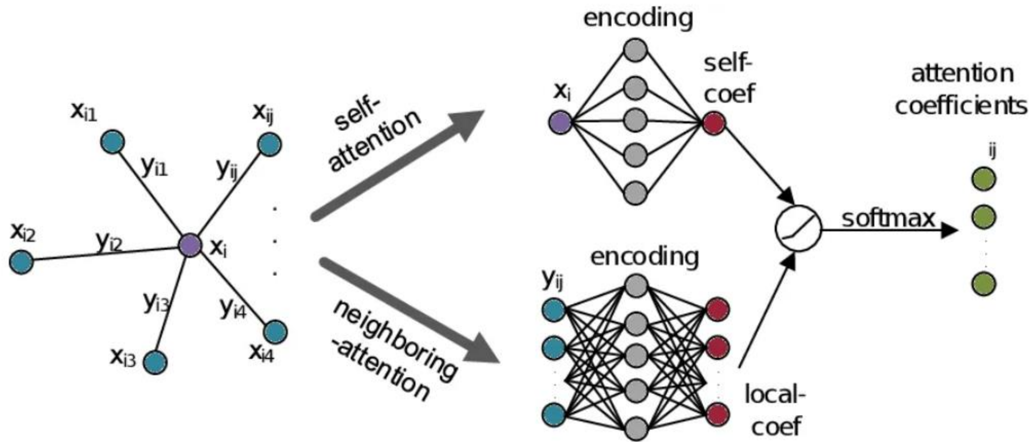


Figure 9: GAT architecture with attention mechanism in [24]. Attention coefficients ij for all the neighboring nodes of node x_i are calculated. Self-attention coefficient of node x_i and neighborhood attention coefficient for all the neighboring edges, y_{ij} are calculated and combined where softmax activation function is applied to obtain all the attention coefficients ij of all nodes x_{ij}.

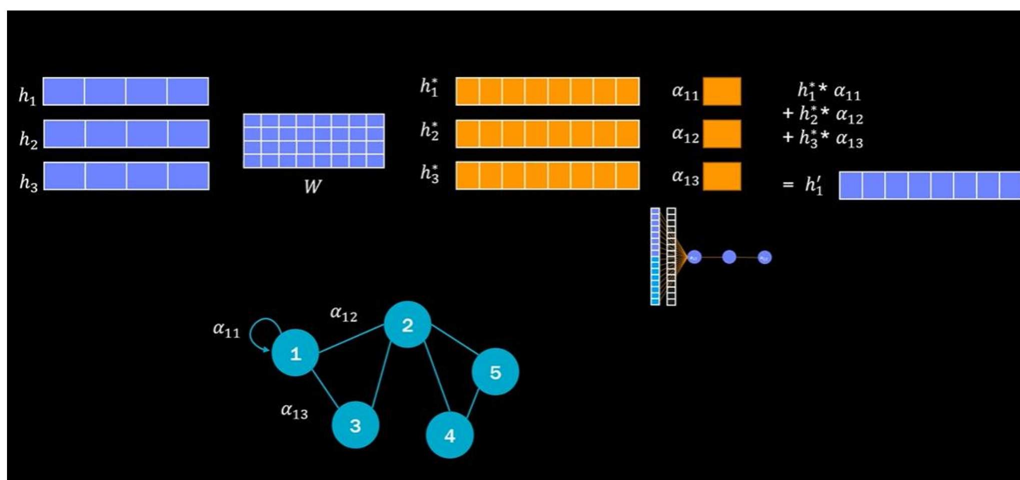


Figure 10: Pictorial representation of how attention coefficients are utilized in GAT taking node 1 of the graph as an example in [25]

Figure 10 explains the working mechanism of equation 7. In figure 10, a graph consisting of 5 nodes with self-loop at node 1 is considered as an example. For node 1, first, node embeddings of its neighboring nodes (h_2 and h_3) and its own (h_1) are collected, then they are all passed through a learnable linear transformation i.e. multiplying with W. The intermediate node states (h_1^*, h_2^* and h_3^*) are then obtained. Now, for each edge, two corresponding intermediate node states are passed through the shared single layer neural network giving the attention coefficients α_{11}, α_{12} and α_{13}. The intermediate node states are then summed up as a linear combination weighted with the corresponding calculated attention coefficients. The final updated node embedding (h_1') for node 1 is then obtained. This is done for all the nodes in the network to obtain final updated feature matrix of the graph thus completing the message passing layer [25].To stabilize the learning process of self-attention, multiheaded attention (having several independent attention mechanisms) is performed [3].

2.3.2.2 Limitations of GATs

The limitations observed in GATs are explained below [26].

- The use of attention mechanisms increases the computational complexity of GATs compared to GCNs. Attention computation requires calculating weights for each neighbor of each node, which can be expensive for large or dense graphs.
- The attention mechanism in GATs assigns weights to each neighbor of a node based on their importance. However, this can introduce a bias towards certain types of edges, which might not always be desirable. Also, GATs focus primarily on node features and their relationships, but do not inherently consider edge features. Integrating edge features into the attention mechanism can be complex and may require modifications to the GAT architecture.

2.3.3 Graph Sample and Aggregation (GraphSAGE)

Introduced by Hamilton et al. in [27], GraphSAGE is a popular GNN architecture. GraphSAGE was developed to address the limitations of then existing GNN models, particularly in handling large-scale graphs efficiently and learning node representations in an inductive manner (being able to generalize the unseen nodes and graphs during training). It uses a sampling method to select a subset of neighbors and create mini-batches for aggregation, improving scalability. It also supports different aggregation functions such as mean, max pooling, and LSTM-based methods. GraphSAGE enables inductive learning on graphs by learning aggregator functions that can induce the embedding of a new node based on its neighbors. Unlike transductive methods that necessitate knowledge of the entire graph (e.g., GCN), GraphSAGE generalize to unseen nodes, making it particularly useful for dynamic graphs where new nodes appear over time [3], [27].

2.3.3.1 Learning Process

The leaning process of GraphSAGE is explained below.

Neighbor Sampling: GraphSAGE introduces a sampling mechanism that enables efficient training on large graphs by sampling a fixed-size neighborhood for each node during training. Here, a fixed number of neighbors are randomly sampled for each node which alleviates the computational burden and memory constraints associated with large graphs. For each node v, GraphSAGE samples a fixed-size neighborhood $\boldsymbol{\mathcal{N}}(v)$.

Flexible Aggregation: GraphSAGE uses a flexible aggregation mechanism, allowing the model to use different aggregation functions (such as mean, max pooling) to combine information from neighboring nodes. The aggregated information is then transformed using a weight matrix and a non-linear activation function represented mathematically as:

$$h_v^{(k+1)}=\sigma\left(\mathbf{W}^{(k)} aggregate(h_u^{(k)}) \mid u \in \boldsymbol{\mathcal{N}}(v)\right) \tag{8}$$

where $h_v^{(k+1)}$is the representation of node v at layer k, $\mathbf{W}^{(k)}$ is the weight matrix at layer k , $aggregate(.)$ is the chosen aggregation function, σ is a non-linear activation function.

Node Representation Learning: GraphSAGE learns node representations through a multi-layer architecture. In each layer, nodes gather information from their neighbors using the specified aggregation function, update their representations, and pass the updated representations to the next layer. This continued process through multiple layers allow the model to learn increasingly abstract representations of the graph. Consequently, the final layer's output can be used for classification tasks or link predictions.

In general, GraphSAGE generates embeddings for previously unseen nodes by leveraging node feature information to efficiently generate node embeddings. Instead of training individual embeddings for each node, GraphSAGE learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood. The algorithm assumes that the model parameters are already learned and fixed. It incorporates node features in the learning algorithm, allowing it to simultaneously learn the topological structure of each node's neighborhood as well as the distribution of node features in the neighborhood. By training a

set of aggregator functions that learn to aggregate feature information from a node's local neighborhood, GraphSAGE can generate embeddings for entirely unseen nodes by applying the learned aggregation functions. The embedding generation (i.e., forward propagation) algorithm in GraphSAGE involves learning how to aggregate feature information from a node's local neighborhood, such as the degrees or text attributes of nearby nodes. This process allows GraphSAGE to efficiently generate embeddings for previously unseen data. The model parameters can be learned using standard stochastic gradient descent and backpropagation techniques, enabling the generation of embeddings for unseen nodes. The approach is designed to generalize to graphs with the same form of features and can be applied to graphs without node features as well.

2.3.3.2 Limitations of GraphSAGE

Some of the limitations in GraphSAGE are discussed below [3], [26].

- The random sampling of neighbors can introduce noise or bias in the aggregated representations. Also, the choice of neighbor sampling strategy can significantly impact the model performance.
- Although sampling helps with scalability, it can lead to a trade-off between efficiency and accuracy, as reducing the number of sampled neighbors can result in a loss of information.

2.3.4 Graph Autoencoder (GAE)

GAEs [28] are unsupervised GNNs that aim to learn graph representations through an encoding-decoding process. The encoder learns graph embeddings, while the decoder reconstructs the graph structure or features. GAEs are designed for unsupervised learning tasks on graphs. They work by encoding the nodes into a latent space and then reconstructing the graph's adjacency matrix or other graph properties from these embeddings. GAEs combine GNNs with the autoencoder framework to learn low-dimensional representations of graph data. These embeddings capture the structural properties of the graph and can be used for downstream tasks such as node clustering, community detection, and graph reconstruction. GAEs allow for unsupervised learning, making them useful for tasks where labeled data is scarce [26].

In [29], Kipf et. al proposed a variant of the autoencoder model, Variational Graph Auto-Encoders (VGAE). VGAE extends autoencoder architecture by learning probabilistic embeddings that can generate new graph structures. VGAE combines the principle of GNNs with variational inference to encode graph structures and node features into a latent space and then reconstructs the graph from the latent representation. The latent representation is sampled from the Gaussian distribution during training, a reparameterization method, that ensures the model to learn the uncertainty in the latent space. Likewise, in [30], Pan et al. proposed Adversarially Regularized Graph Autoencoder for graph embedding (ARGA). ARGA explores the combination of autoencoder architectures with adversarial training for better representation learning.

In general, the architecture of GAE typically consists of two main components, encoder, and decoder. The encoder maps the input graph into a lower-dimensional representation. This is usually done using a GNN such as but not limited to GCN and GAT. The encoder takes the

adjacency matrix of the graph and the feature matrix as input and produces a latent representation of the graph. The decoder, on the other hand, reconstructs the graph from the encoded latent representation. In practice, the decoder is often a simple function that reconstructs the adjacency matrix or node features from the latent representation. Decoder can use various reconstruction techniques, such as inner product or bilinear decoding, depending on the specific GAE architecture. Figure 11 shows the general GAE architecture with an encoder and decoder.

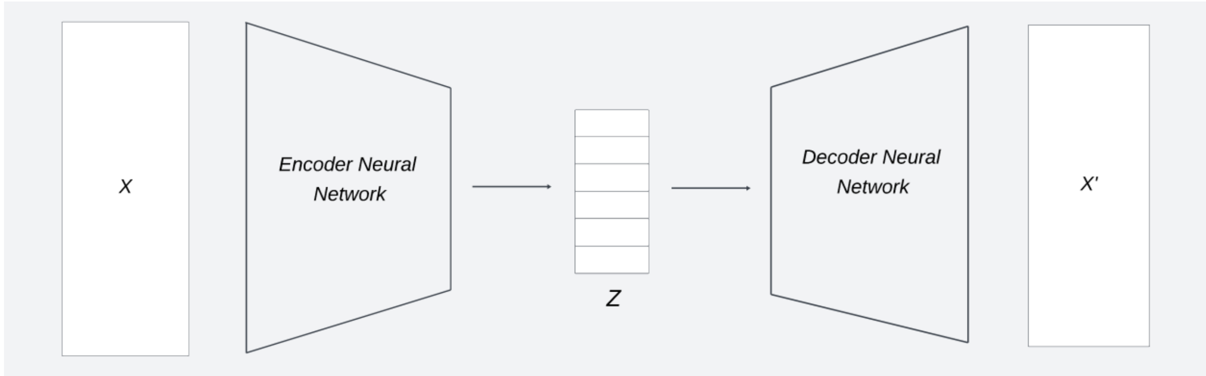


Figure 11: A general GAE architecture, comprising of an encoder and decoder with **X** as the input to the encoder and **X′** the output generated by the decoder. **Z** is the latent space representation of the input **X**. Typically input is adjacency matrix, feature matrix or both.

2.3.4.1 Learning Process

The generalized learning process of GAE can be summarized below.

Encoder: An encoder typically consists of a GNN, such as GCN or GAT, that takes an input graph represented by its adjacency matrix $\mathbf{A}$ and node feature matrix **X**. It then produces a lower dimension latent space representation **Z** of the graph given by **Z** = *encoder*(**A**, **X**) where *encoder* is any chosen GNN (GCN, GAT, etc.).

Decoder: A decoder reconstructs the graph from the latent representation **Z**. The reconstruction can be performed using various techniques, such as inner product or bilinear decoding. For example, the inner product decoding computes the reconstructed adjacency matrix $\widehat{\mathbf{A}}$ as $\widehat{\mathbf{A}} = \sigma(\mathbf{Z}\,\mathbf{Z}^T)$ where σ is a sigmoid function.

Loss function: The loss function in GAEs is typically based on the reconstruction error between the input and reconstructed adjacency matrices. Taking Binary Cross-Entropy (BCE) loss as an example of loss function, the loss function L is given by:

$$\mathrm{L} = -\sum_{i,j}[\mathbf{A}_{ij} log \mathbf{A}^*_{ij} + (1 - \mathbf{A}_{ij})(1 - \mathbf{A}^*_{ij})] \tag{9}$$

where, $\mathbf{A}_{ij}$ is the actual adjacency matrix element between node *i* and node *j*, $\mathbf{A}^*_{ij}$ is the predicted probability of edge between node *i* and *j*. GAE essentially learns by minimizing the loss function.

2.3.4.2 Limitations of GAE

Some of the limitations of GAE are discussed below [26], [28] .

- The learned node embeddings may lack interpretability, making it challenging to understand what the embeddings represent or how they relate to the original graph.
- The choice of loss function can impact the performance of GAEs. For example, using the imprecise loss function could result in poor graph reconstruction or suboptimal embeddings.
- Handling dynamic graphs (where the structure changes over time) can be challenging for GAEs, as the model may need to be retrained to accommodate new nodes and edges.

2.3.5 Other GNNs

While models like GCNs, GATs, and GAEs are designed to work with static attributed graphs, new GNN architectures have been introduced to handle dynamic graphs and other specific scenarios. Additionally, existing GNN architectures have also been refined and improved for better performance.

Xu et al. in [22] proposed Graph Isomorphism Network (GIN) that uses a sum aggregation function and a trainable parameter to avoid over-smoothing and improve representational power to recognize different graph structures. GIN proposes a family of architectures that can accurately capture the structure of the graph to determine if two graphs are isomorphic (essentially the same in structure) or not.

There are several dedicated GNNs to deal with dynamic graphs. As dynamic graphs pose unique challenges because they change over time, with nodes and edges being added or removed, several dedicated GNN architectures have been developed specifically for dynamic graphs to address the challenge. Temporal Graph Networks (TGNs) [31] and Structural-Temporal Graph Neural Networks (STrGNNs) [32] are some of the GNNs developed in the dynamic graph paradigm.

The scope of this thesis work is limited to GNNs based on static attributed graph and does not cover the study of GNNs based on dynamic graph.

3 Review of GNN Algorithms for Outlier Detection

This chapter provides an overview of the current state-of-the-art in GNN algorithms for outlier detection. It begins with a formal definition of outliers within the context of GNNs, understanding different types of outliers followed by the discussions of various categories of GNN algorithms tailored for detecting outliers in graph-structured data. It also examines the key approaches and methodologies employed in GNN-based outlier detection algorithms.

3.1 Outliers in GNNs

An outlier in any form of data is typically an anomaly, an observation that deviates significantly from the expected norm. Outlier could be anything from anomaly, rarity, peculiarity, novelty, or exception depending on several fields of applications that differ significantly from the mass in one form or the other and violate the norms and the standards. Much like a discordant note in a musical piece, outliers can disrupt the harmony of the system and affect its overall functioning. While traditional data analysis often views outliers as statistical anomalies, they can hold the key to understanding the complex dynamics within interconnected systems. Grubbs in [33] defined anomalies as ‘one that appears to deviate markedly from other members of the sample in which it occurs’.

In the context of GNNs, outliers take on a deeper significance due to the complex and interconnected nature of graph data. Graphs model relationships and interactions within various systems such as social networks, biological systems, and financial transactions. Anomalies that emerge within these relationships may indicate more than statistical oddities; they can signify critical shifts, disruptions, or irregularities within the system. Outliers are considered as data points in a graph such as nodes, edges, or subgraphs that exhibit atypical behavior or deviate significantly from the overall pattern or structure of the graph. This deviation could manifest in different forms, such as unexpected connections, unusual node features, or anomalous subgraph structures. Two important categories of outliers in graphs are node-level (contextual) outliers, and edge-level (structural) outliers [34],[35],[36].

- **Node-level (Contextual) Outliers:** Node-level outliers are nodes that deviate from expected behavior within a specific context or feature space. These outliers display unexpected attributes, or abnormal behaviors compared to other nodes. These are nodes whose attribute values (e.g., node features or properties) deviate significantly from the global distribution of attribute values in the graph.
- **Edge-level (Structural) Outliers:** As edges represent the relationships or connections between nodes, outliers at the edge-level involve anomalous interactions or connections. Structural outliers are nodes that deviate from the expected structure or relationships in the data and don't follow the typical patterns of connectivity or interaction. For instance, in a citation network, an academic paper cited by an unusually high number of other papers in a short time could be an outlier.

There is also a third category of outliers, community-level (sub graph-level) outliers. This type of outliers include a group of nodes and edges that deviate significantly from the main graph.

Figure 12 shows the concept of node-level and edge-level outliers in a graph.

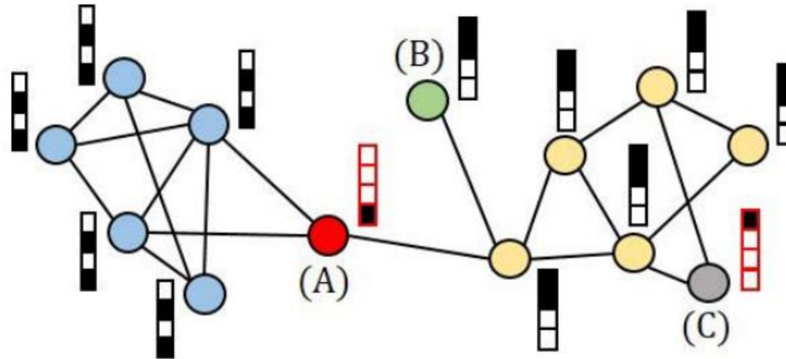


Figure 12: Examples of outliers in a graph in [37]. Nodes A and C are node-level outliers, nodes A and B are edge-level outliers. Node A is both node-level and edge-level outlier.

3.2 Outlier Detection

Outlier detection, in general, is a way to identify outliers from a given set of community or network. It can be seen as a data mining process that identifies the unusual patterns or behaviors or connections which deviate slightly or significantly from the majorities in an entire dataset. Outlier detection, also known as anomaly detection, is a critical application in the analysis of graph-structured data, where GNNs play a significant role in modeling complex relationships. This research area has garnered notable attention due to the widespread presence of outliers in various disciplines, including but not limited to security, finance, medicine, and social networks. GNNs provide a robust and adaptable framework for representing and interpreting graph data, which is a common data structure across many domains. By capturing intricate patterns and associations within graph data, GNNs facilitate the identification of anomalies that may signify potential risks or opportunities [34].

The challenge with outliers is the nuanced way in which these anomalies can manifest. Traditional outlier detection methods may not capture the complexity of relationships and structures in graph data. This is where GNNs come into the play, by learning representations of nodes, edges, and subgraphs, GNNs promise to provide more sophisticated detection and interpretation of outliers. They are able to capture intricate patterns and relationships, making them particularly adept at identifying outliers. This capability enables more precise detection of critical events such as financial fraud, network intrusions, social spam, spread of misinformation, and other detrimental occurrences.

Applications of Outlier Detection

Outlier detection using GNNs is instrumental in uncovering peculiar events such as financial fraud, network intrusions, and social spam. For instance, in the finance sector, GNN-based outlier detection can reveal suspicious transactions indicative of money laundering or insider

trading. Similarly, in network security, GNNs can identify unusual network traffic patterns signaling potential cyber-attacks. In the realm of social media, GNNs can help detect the spread of fake news or misinformation by identifying anomalous patterns in content sharing and user interactions. Additionally, in the field of healthcare, GNN-based outlier detection can aid in the early identification of emerging diseases or unusual health trends. By recognizing these outliers, GNNs enable proactive responses to emerging threats and insights into underlying systemic shifts [12], [14], [32], [34], [35].

3.2.1 Review of Outlier Detection Algorithms in GNNs

The state-of-the-art GNNs for outlier detection is a rapidly evolving area. Before the advent of deep learning, several non-deep learning techniques were used to identify outliers [33]. In such kind of approaches, graph outlier detection was initially transformed into traditional anomaly detection problem. OddBall [38] is one such approach that extracts the features from each node and its 1 hop neighbors to detect structural outliers. The identification of outlier nodes heavily relied on the selection of statistical features. However, it is not always possible to properly choose the most suitable features, especially from large datasets and hence these traditional approaches do not effectively capture structural information. The traditional way of manual feature engineering and building a tailored statistical model to detect outliers is also labor-intensive, expensive, time consuming and above all not always effective as graph data of a real-world network could easily contain millions of nodes and edges leading to computational overhead in both execution time and storage as well.

Subsequently, network representation-based methods have been exercised in order to capture information from the graph structure. In network representation-based methods, first, the graph structure is encoded to an embedded vector and outliers are detected through further analysis. By pairing the conventional anomaly detection methods such as density-based, distance-based with node embedding methods, outliers are identified (node present in low-density area or node far away from the majorities). Node2Vec [39] and LINE [40] are some of the earlier representative methods. Node2Vec uses biased random walks to explore the graph. To introduce bias in the random walks, Node2Vec uses two parameters, return parameter (p) to control how likely the walk is to revisit a node if it has already visited and in-out parameter (q) to control how far the walk is likely to stray from the starting node. The node embeddings are computed from these walks, and outliers can then be identified using metrics such as distance-based measures and clustering in the embedding space. LINE, on the other hand, learns node embeddings by preserving the proximity relationships in a graph, capturing both first-order (direct relationships between nodes i.e., edges) and second-order (similarity between nodes' neighborhoods i.e., the sets of nodes connected to them) proximity between nodes. The first-order and second-order proximities are then combined using an objective function to obtain the node embeddings and outliers are detected using distance-based methods.

While methods such as Node2Vec and LINE have been widely used for learning node embeddings from graph data, they do have certain limitations. These methods primarily focus on local neighborhood information through random walks or proximity measures making it difficult to capture global structures effectively. Also, these methods generate fixed embedding vectors for each node, which may not fully capture the intricate relationships in dynamic graph data where relationships and connections can change over time. Consequently, more advanced

techniques, particularly the use of deep learning-based architectures [41] were studied and have been brought to the implementation. Deep learning-based methods focus on learning complex representations from graph data, whereas traditional methods like Node2Vec and LINE primarily learn embeddings through fixed random walks or proximity measures. GNNs, in particular, have become a popular approach for efficiently detecting anomalies in graphs due to their ability to intuitively learn graph representations through message passing mechanisms. These networks handle graphs with complex structures and attributes as input data, making the process of learning and extracting anomalous patterns from graphs straightforward. This is especially useful when dealing with graphs that have intricate connections and diverse node attributes. In this approach, state-of-the-art graph anomaly detection methods often combine GNNs with existing deep learning approaches, such as RNNs [8] or transformers [42] . GNNs excel at simultaneously examining graph topology and node attributes, providing a comprehensive view of the graph's behavior.

Despite the novelty of this field, researchers have already covered a wide variety of graph types, such as static and dynamic graphs, as well as plain and attributed graphs. Structural anomalies in graph topology range from node and edge anomalies to subgraph-level anomalies but there appears to be a focus on outlier detection at node-level in static attributed graphs, leaving other areas with room for further exploration.

The scope of this thesis is limited to node-level outlier detection algorithms in static attributed graphs. The following subchapters outline key aspects of GNN-based node-level outlier detection methods.

3.3 GNN-based Node-level Outlier Detection in Static Graph

In static attributed graphs, where nodes and edges remain consistent over time, various GNN-based outlier detection algorithms have been developed. These algorithms often leverage GNN architectures such as GCN [18], GAT [23], GAE [28] either individually or in combination for detecting anomalies. While many algorithms focus on node-level outlier detection, there are also methods that address edge-level and graph-level outlier detection both on static and dynamic graphs.

3.3.1 GCN-based method

GCN-based methods [37] leverage the convolutional operation of GCNs [18], where information from a node's neighbors is aggregated through message passing. This process allows the creation of new node embeddings that capture the graph's structure and node features. These embeddings can then be analyzed to detect nodes that significantly deviate from the norm, indicating potential outliers. GCNs are often combined with GAE, where the GCN layers serve as the encoder component and are responsible for transforming the input graph data into a lower-dimensional latent space. The decoder component of the GAE then reconstructs the graph from these latent representations. This combination is particularly useful for outlier detection, as the model can be trained to capture the typical patterns in the graph during the encoding and decoding process. Anomalies can be identified by comparing the

reconstructed graph with the original input graph; significant discrepancies may indicate the presence of outliers [35].

- **Semi-GCN**: Kumagai et al. in [43] proposed a method called semi-GCN that combines the elements of supervised and unsupervised learning to efficiently utilize the available labeled data while still learning from the entire graph. In this method, the main idea is to extend GCNs to scenarios where only some nodes in the graph have labels. This is known as a semi-supervised learning approach, as it leverages both labeled and unlabeled data. Like a standard GCN, semi-GCN method uses layers of graph convolution. Each layer aggregates information from a node's neighbors and combines it with the node's own features to create new representations. The semi-GCN method is trained using the labeled nodes in the graph. This means the model learns from the features and labels of the nodes with available labels. The model uses the learned parameters to propagate information throughout the graph. This allows it to generate predictions for nodes that were originally unlabeled based on the patterns it learned from the labeled nodes. The loss function used in *semi-GCN* focuses on the difference between the predicted labels and the true labels of the labeled nodes. After passing through the graph convolutional layers, nodes are embedded in a latent space. These embeddings capture the structure and features of the graph. Finally, a reconstruction error is computed based on the difference between the original node features and the reconstructed features obtained from the latent embeddings. The reconstruction error for a node v is calculated as $reconstruction\ error_v = \| h_v - h_v^{(final)} \|^2$ where h_v is the initial node embeddings of node v and $h_v^{(final)}$ is the final node embedding of node v obtained from GCN layers. Nodes with a high reconstruction error are considered potential outliers, as they deviate significantly from the expected patterns learned from the graph. A threshold can be set to classify nodes as outliers based on their reconstruction error.
- **ResGCN:** In[44], Pei et al. proposed Residual GCN (ResGCN), where GCNs are extended that incorporated residual connections, similar to those used in ResNet [45] in the field of deep learning for images. This model aims to alleviate issues such as vanishing gradients, sparsity and over-smoothing and improve training stability using residual connections, allowing the model to capture more complex patterns in graph-structured data. In each layer, the ResGCN performs a graph convolution similar to a standard GCN, but with the addition of a residual connection. ResGCN is almost same as semi-GCN, but ResGCN has residual connections that allows for better training stability.
- **Contrastive self-supervised learning (CoLA):** Liu et al. in [46] proposed Contrastive self-supervised Learning (CoLA), a different method compared to semi-GCN and ResGCN in the sense that CoLA employed contrastive learning to generate representations of nodes that emphasize the differences between normal and anomalous patterns in the graph. CoLA is a variant of self-supervised learning, a technique used to pretrain neural networks on unlabeled data by creating supervised-like tasks from the data itself. In contrastive self-supervised learning, the model learns to encode data samples in such a way that similar samples are pulled closer together in the embedding space, while dissimilar samples are pushed apart. CoLA is particularly useful when labeled data is scarce or expensive to obtain.

The contrastive loss function is designed to maximize the similarity between similar node pairs and minimize the similarity between dissimilar node pairs. Outliers are detected by analyzing the similarity scores between node pairs. Nodes that have lower similarity scores with other nodes (based on embeddings) were considered outliers. While semi-GCN and ResGCN focused more on reconstructing the input graph or learning node representations, CoLA's contrastive learning focuses on maximizing differences between similar and dissimilar nodes. Figure 13 explains the framework of CoLA.

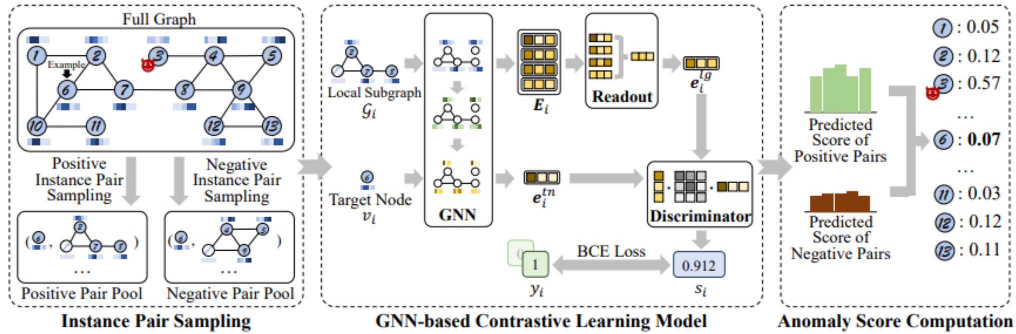


Figure 13: CoLA framework presented in [46]

The working process of CoLA can be summarized as follows.

Data Augmentation: CoLA often starts with data augmentation techniques to generate augmented versions of the input data. This helps in exposing the model to diverse variations of the input samples and improves its robustness.

Positive and Negative Samples: For each input sample, CoLA creates positive and negative pairs. Positive pairs consist of two augmented versions of the same input sample. Negative pairs consist of an input sample and another sample that is dissimilar to it, often randomly selected from the dataset.

Embedding Generation: The encoder network takes both positive and negative samples as input and generates embeddings (feature representations) for each sample.

Backpropagation and Parameter Update: The gradients of the contrastive loss are backpropagated through the network, and the model parameters are updated using optimization algorithms.

However, designing effective data augmentation strategies is crucial for the success of CoLA. Choosing appropriate hyperparameters, such as the contrastive loss margin and temperature parameter, can significantly impact the quality of learned representations [46], [47].

3.3.2 GCN-based GAE method

Autoencoder framework [28] has been extensively used in several of the GNN-based outlier detection methods. In general, nodes are often encoded into a latent space, and then the network tries to reconstruct the original graph structure from these embeddings. In outlier detection, nodes that are poorly reconstructed (high reconstruction error) are candidates for outliers, indicating that their pattern of connections differs significantly from the norm defined by the

rest of the graph. Combining the principles of autoencoders with GCNs, some models aim to reconstruct graph data using GCN-based encoding and decoding layers.

- **Deep Anomaly Detection on Attributed Network (DOMINANT)**: Ding et al. in [58] and is one such GCN-based GAE method where an anomaly score is calculated for each nodes using the network reconstruction errors. Here, GCN-based encoder is used to learn both the structure of the graph and the attributes of the nodes. A decoder is then used to utilize the reconstruction errors to detect outlier nodes in an unsupervised manner. DOMINANT uses GCN encoder, called as attributed network encoder, whose input is both topological structure and nodal attributes of the graph. The encoder consisting of multiple graph convolutional layers subsequently aggregates information from a node's neighbors and transform the input node features into a lower-dimensional latent space ultimately providing the latent space representations (embeddings) of the nodes along with the structural information of the graph. A decoder section, consisting of two decoders, structural reconstruction decoder and attribute reconstruction decoder, takes the latent representations as input and reconstructs both the original topological structure and nodal features. Finally, reconstruction errors for each node are computed by comparing the original node features with the reconstructed features obtained from the decoder and the nodes with high reconstruction errors are considered possible outliers. DOMINANT, unlike previously discussed models such as semi-GCN and ResGCN, spots anomalies by calculating the reconstruction errors from both nodal attributes and structural point of view in an unsupervised manner [58]. Figure 14 describes the architecture of DOMINANT.

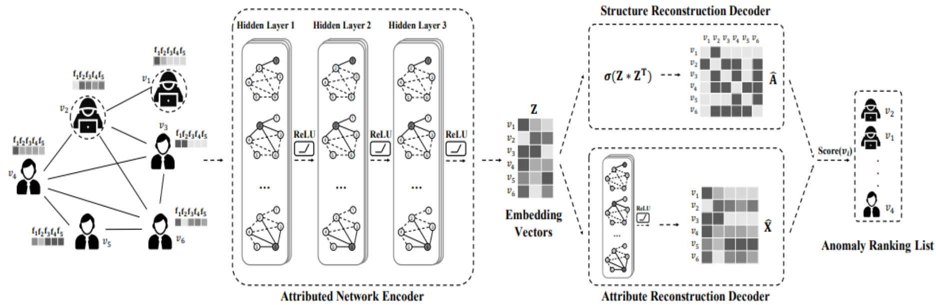


Figure 14: Architecture of DOMINANT shown in [58]. DOMINANT has one attributed network encoder made of GCN where adjacency matrix **A** and feature matrix **X** of input graph is fed which produces latent space representation matrix **Z** as the output. Through structure reconstruction decoder and attribute reconstruction decoder, DOMINANT produces estimated adjacency matrix $\widehat{\mathbf{A}}$ and estimated feature matrix $\widehat{\mathbf{X}}$.

In decoder section of DOMINANT, two reconstruction errors, structure reconstruction error and attribute reconstruction error are separately calculated. A structure reconstruction error is calculated as $R_s = \mathbf{A} - \widehat{\mathbf{A}}$ where, **A** is the original adjacency matrix and $\widehat{\mathbf{A}}$ is the estimated adjacency matrix for the learned latent representation matrix **Z**, an output of the attributed network encoder. Likewise, to calculate attribute reconstruction error, a link prediction layer is first trained based on the output of attributed network encoder **Z**. Here, $\widehat{\mathbf{A}}$ is calculated as $\widehat{\mathbf{A}} = sigmoid(\mathbf{Z}\,\mathbf{Z}^{\mathrm{T}})$ where *sigmoid* is the sigmoid activation function. The

estimated node feature matrix $\hat{\mathbf{X}}$ is then calculated as $\hat{\mathbf{X}} = f_{Relu}(\mathbf{Z}, \mathbf{A}|\mathbf{W})$ where f_{Relu} is the ReLU activation function. With this, attribute reconstruction error R_A is calculated as $R_A = \mathbf{X} - \hat{\mathbf{X}}$ where $\mathbf{X}$ is the original node feature matrix. Finally, the objective function L is obtained as L = (1-α) $R_s + \alpha R_A$, where α is the controlling parameter. The model works by minimizing the objective function L using gradient descent. After certain iterations, the outlier score of each node is then calculated and the ones with larger scores are considered as outliers.

- **Deep multi-view framework for anomaly detection (ALARM)**: Peng et al. in [48] proposed ALARM using a multi-view approach, leveraging different views of the graph to improve anomaly detection. It uses multiple views of the graph (e.g., different sets of node attributes or different graph structures) to improve the robustness and accuracy of anomaly detection. ALARM combines GCNs and autoencoders for learning embeddings and reconstructing node features from different views, and then uses an attention mechanism to integrate information from these views. Here, a learning process is typically multi-view learning utilizing multiple views of the graph. These views may include different sets of node attributes, graph structures, or combinations of both, each view providing a different perspective of the graph. For each view, ALARM uses a GCN-based GAE to learn latent space representations of the nodes and reconstruct the original features.

3.3.3 GAT-based GAE method

In GAT-based GAE method [37], GAT architectures are exploited to utilize their attention mechanisms to weigh the importance of each neighbor when aggregating the information for a given node. This attention-based approach allows the model to focus on the most relevant neighbors and better capture the complex relationships in the graph. This method essentially uses GAT layers as the encoder to generate the latent representations of the graph. These representations are then used by the decoder to reconstruct the graph. The reconstruction error between the original graph and the reconstructed graph are then used to identify outliers.

- **Dual Autoencoder for Anomaly Detection on Attributed Networks (AnomalyDAE)**: In [49], Fan et al. introduced AnomalyDAE, an end-to-end variant of GAT-based GAE method consisting of dual autoencoders, structure autoencoder and attribute autoencoder. These autoencoders are trained jointly to capture complex interactions between network structure and node attributes in attributed networks. The working mechanism is explained as follows:
  Structural Autoencoder: Here, the encoder first learns the node embeddings based on the network structure. GAT [23] layer is used to obtain attention mechanism in order to grasp important structural patterns. The structure encoder transforms the original node attribute matrix $\mathbf{X}$ into a lower-dimensional latent representation $\tilde{\mathbf{Z}}^v$ as $\tilde{\mathbf{Z}}^v = \sigma\ (\mathbf{X}\mathbf{W}^{v(1)} + b^{v(1)})$ where, $\sigma(.)$ is the activation function, $\mathbf{W}^{v(1)}$ and $b^{v(1)}$ are the weight and bias learned by encoder. Using the transformed node embeddings $\tilde{\mathbf{Z}}^v$, GAT layer aggregates information from neighbor nodes through a shared attention mechanism $e_{i,j} = attn(\tilde{\mathbf{Z}}_i^v, \tilde{\mathbf{Z}}_j^v)$ where, $e_{i,j}$ is the importance weight of node v_i to v_i and *attn*(.) is the GAT. The final importance weight $\gamma_{i,j}$ is computed using the softmax function. Consequently, the final node embedding

$\mathbf{Z}_i^v$ is then calculated by a weighted sum based on the learned importance weights as $\mathbf{Z}_i^v = \sum_{k\in\mathcal{N}(i)} \gamma_{i,k}\, \tilde{Z}_k^v$ where $\mathcal{N}(i)$ denotes the neighbors of node v_i.
The structure decoder uses the final node embeddings $\mathbf{Z}^v$ as inputs to reconstruct the original network structure $\widehat{\mathbf{A}} = sigmoid(\mathbf{Z}^v\,(\mathbf{Z}^v)^{\mathrm{T}})$ where $sigmoid(.)$ is the sigmoid activation function.
Attribute Autoencoder: In attribute encoder, two non-linear feature transformation layers are first utilized to map the observed attribute data to a latent attribute embedding $\mathbf{Z}^A$. The attribute decoder uses the node embeddings $\mathbf{Z}^v$ obtained from the structure encoder and the attribute embeddings $\mathbf{Z}^A$ to reconstruct the original node attributes $\widehat{\mathbf{X}}$ as $\widehat{\mathbf{X}} = (\mathbf{Z}^v\,(\mathbf{Z}^A)^{\mathrm{T}})$. In AnomalyDAE [49], the training objective is to minimize the reconstruction errors of both the network structure and node attributes. Finally, the anomaly score for nodes are calculated as the reconstruction error from both the network structure and node attribute perspectives. These scores can then be used to determine a threshold which allows for classification of nodes and anomalous based on their score, the one with highest scores often classified as anomalies. Figure 15 is the pictorial representation of AnomalyDAE algorithm.

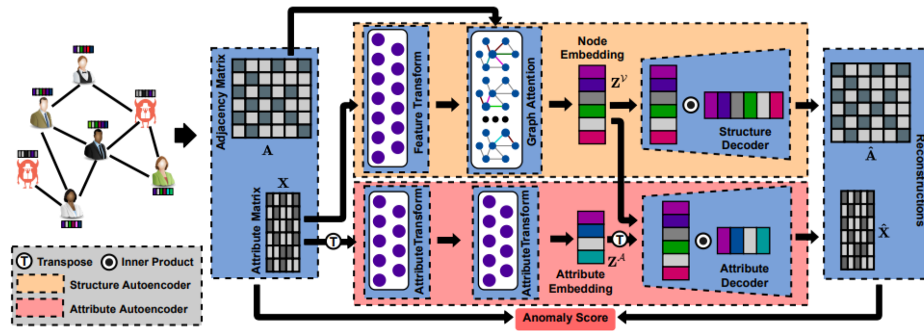


Figure 15: AnomalyDAE architecture presented in [49]. The framework is made up of two autoencoders, structure autoencoder and attribute autoencoder. Reconstructed adjacency matrix $\widehat{\mathbf{A}}$ and feature matrix $\widehat{\mathbf{X}}$ are calculated respectively from structure decoder and attribute decoder which are combinedly used to calculate the anomaly score and predict the outlier.

3.3.4 Other GNN-based algorithms for outlier detection

There are outlier detection algorithms that are based on different architectures other than GCN, GAT and GAE. Generative Adversarial Attributed Network Anomaly Detection (GAAN) [50] is one such algorithm which is based on Generative Adversarial Network (GAN) [51] . GANs are a class of machine learning models designed for generating new, synthetic data that resembles a given dataset. GANs use an adversarial framework and are known for their innovative approach to learning and generating new data.

- **Generative Adversarial Attributed Network Anomaly Detection (GAAN)**: Chen et al. (2020) in [50] proposed GAAN, a method to detect node level outlier based on GAN. The basic architecture of GAAN consists of three parts, generator, encoder, and discriminator.

The generator takes a Gaussian noise as input and generates synthetic graph nodes. The encoder then maps both real and fake graph nodes into a latent space and encodes graph structure information into the node's latent representation through the sample covariance matrix for both real and fake nodes. Its goal is to create data that is indistinguishable from real data. The discriminator finally takes the output generated by the encoder as input and predicts whether the connected nodes are real (from the graph) or fake (from the generator). The model uses the encoder output to assess anomalies by considering sample reconstruction error and real-sample identification confidence. The working mechanism is summarized as follows:

Generator: The generator, Gn, in the GAAN framework aims to approximate the distribution of the original feature matrix X using a low-dimensional prior Gaussian distribution. It employs MLP as the generator, which consists of layers that perform linear transformations and non-linear mappings. Here, for n^th MLP layer of generator Gn, the output $\mathbf{H}_{Gn}^{(n+1)}$ is calculated as $\mathbf{H}_{Gn}^{(n+1)} = f(\mathbf{W}_{Gn}^{(n)}\mathbf{H}_{Gn}^{(n)} + b_{G}^{(n)})$, where $\mathbf{H}_{Gn}^{(n)}$ is the input of nth MLP layer, $\mathbf{W}_{Gn}^{(n)}$ and $b_{Gn}^{(n)}$ are the nth layer parameter matrix and corresponding bias respectively. f (.) is ReLU activation function.

Encoder: The encoder, E, converts the original node feature matrix X and the generator's output, approximated node feature matrix, X' into a low-dimensional latent space with the same dimensions as the generator's prior data distribution. This transformation is achieved using a three-layer MLP with a ReLU activation function. The attribute matrix X and the generator's output matrix X' serve as the initial inputs for the first layer eventually resulting in latent representations $\mathbf{Z}$ and $\mathbf{Z}'$, respectively.

Discriminator: The discriminator, D, in GAAN captures graph structure information by estimating the adjacency matrix $\mathbf{A}$ using graph embedding. The adjacency matrix is estimated through the dot product of the embedding output, which is then passed through an entry-wise sigmoid function. For real data, the adjacency matrix estimate $\widehat{\mathbf{A}}$ is given as $\widehat{\mathbf{A}} = sigmoid(\mathbf{Z}\mathbf{Z}^{T})$, and for generated data by the generator, the adjacency matrix estimate $\widehat{\mathbf{A}}'$ is given as $\widehat{\mathbf{A}}' = sigmoid(\mathbf{Z}'\mathbf{Z}'^{T})$. Here, $\mathbf{Z}$ and $\mathbf{Z}'$ are embeddings encoded from original node attributes $\mathbf{X}$ and generator output $\mathbf{X}'$, respectively. For node pairs $<v_i,v_j>$ with an existing link in the original graph, the discriminator is trained to differentiate whether the dot product of embeddings is from the real data's $\widehat{\mathbf{A}}$ or from the generator's $\widehat{\mathbf{A}}'$.

GAAN is trained by minimizing the cross-entropy loss of the binary classifier. The optimization process in GAAN involves learning the encoder, generator, and discriminator to improve the model's ability to distinguish between real and fake data. The generator aims to confuse the discriminator, while the discriminator works to identify whether the data is from the real distribution or generated. After training, anomaly detection involves calculating an anomaly score for each node based on two loss components: context reconstruction loss and structure discriminator loss. The context reconstruction loss measures how well the generator can recreate the original attributes of a node, while the structure discriminator loss evaluates the model's accuracy in determining if connections between nodes are real. A higher anomaly score suggests the node is more likely to be anomalous. This scoring system helps identify potentially abnormal nodes in the graph

based on both attribute and structural inconsistencies. Figure 16 is the pictorial representation of GAAN.

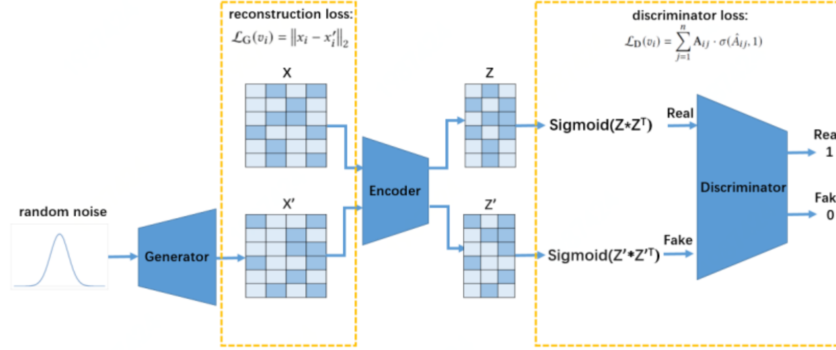


Figure 16: GAAN framework in [50]. GAAN consists of three parts, generator, encoder and discriminator. Generator uses noise level to produce approximated node feature matrix $\mathbf{X}'$, encoder transforms both original feature matrix $\mathbf{X}$ and generated feature matrix $\mathbf{X}'$ into latent space $\mathbf{Z}$ and $\mathbf{Z}'$. Discriminator calculates loss from two latent space inputs and assigns the loss scores to all the nodes.

3.4 Summary of GNN-based outlier detection methods

The summary of node-level outlier detection methods explained in this report is shown in table 1.

Table 1 Summary of GNN based node-level outlier detection methods

Graph Type	Network Architecture	Method	Measurement	Objective function
Static Attributed	GCN	Semi-GCN [43]	Outlier Score	$\| h_v - h_v^{(final)} \|^2$
		ResGCN [44]	Outlier Score	$(1\text{-}\alpha)\|\|(\mathbf{A} - \hat{\mathbf{A}})\|\|_F^2 + + \alpha \|\|\mathbf{X} - \hat{\mathbf{X}} - \lambda R\|\|_F^2$
		CoLA [46]	Outlier Score	$-\sum_{N=1} y_i \log(s_i) + (1\text{-}y_i)\log(1\text{-}s_i)$
	GCN-based GAE	DOMINANT [58]	Outlier Score	$(1\text{-}\alpha) R_s + \alpha R_A$
		ALARM [48]	Outlier Score	$L_s + L_a$
	GAT-based GAE	AnomalyDAE [49]	Reconstruction Loss	$\alpha\|\|(\mathbf{A} - \hat{\mathbf{A}})\theta\|\|_F^2 + + (1-\alpha)\|\|(\mathbf{X} - \widehat{\mathbf{X}})\eta\|\|_F^2$
	GAN	GAAN [50]	Outlier Score	$\alpha L_{Gn}(v_i) + (1-\alpha) L_D(v_i)$

4 Implementation of GNN Algorithms for Outlier Detection

In this chapter, the practical aspects of implementing various GNN algorithms for outlier detection are explored. The focus is on hands-on coding and the in-depth application of the discussed algorithms using PyTorch [4] and the PyGOD [5] library, a dedicated framework for outlier detection in graphs. The chapter is focused on outlier node detection and covers a range of algorithms to represent each of the methodologies, GCN-based, GCN-based GAE, GAT-based GAE, and GAN-based. Algorithms CoLA, DOMINANT, AnomalyDAE and GAAN are chosen to illustrate each method, respectively and are used for node-level outlier detection in static attributed graph network. The datasets used for implementing these algorithms are the Cora and CiteSeer datasets available in PyGOD.

4.1 Introduction to PyTorch

PyTorch [4] is an open-source machine learning library widely used for developing and deploying deep learning models. It offers a dynamic computational graph model, which allows for easier debugging and experimentation compared to static computational graph frameworks. PyTorch provides comprehensive support for neural networks, including efficient tensor operations, automatic differentiation, and GPU acceleration. These features make it a suitable choice for implementing GNNs and other machine learning algorithms. Some of the key features of PyTorch are explained below.

1. Tensor Manipulation: PyTorch is known for its flexible tensor operations, which serve as the foundation for deep learning models. It supports various data types and operations, including matrix multiplication, reshaping, slicing, and more.
2. Automatic Differentiation: PyTorch's ‘autograd’ feature automates the calculation of gradients for tensors, facilitating the training of neural networks.
3. Neural Network Modules: PyTorch's ‘nn’ package provides a range of pre-built neural network layers and modules, including convolutional layers, recurrent layers, activation functions, loss functions, and optimizers.
4. GPU Acceleration: PyTorch seamlessly supports running computations on GPUs, which significantly accelerates training and inference for large models.
5. Data Loading and Preprocessing: PyTorch provides tools for data loading and preprocessing through its ‘torch.utils.data’ package, including ‘Dataset’ and ‘DataLoader’ classes.

4.1.1 Introduction to PyTorch Geometric

PyTorch Geometric (PyG) [4] is an open-source library built upon PyTorch that provides a comprehensive framework for building graph-based machine learning models using the PyTorch deep learning framework. It offers a wide range of tools and utilities specifically designed for working with graph data. Some of the noticeable features of PyG are as follows:

1. Graph Data Handling: PyG provides efficient and flexible data handling capabilities for graph data. It includes data structures such as 'Data' and 'Batch' that allow for easy representation of graph data, including node and edge features, adjacency information, and other attributes. PyG supports various graph types, including undirected and directed graphs, as well as heterogenous graphs with different types of nodes and edges.
2. GNN Architectures: PyG offers a variety of GNN architectures, such as GCN, GAT, GraphSAGE, and more. These architectures are implemented as PyTorch Modules and can be easily customized and integrated even into larger models.
3. Graph Datasets: PyG includes a collection of commonly used graph datasets, such as Cora, PubMed, and others. These datasets are available through the 'torch_geometric.datasets' module and can be easily loaded for use in experiments. PyG also supports loading and working with custom graph datasets.
4. Graph Transformations and Utilities: PyG provides a variety of graph transformations and preprocessing utilities, such as adding self-loops, normalizing adjacency matrices, and performing random walks. These utilities facilitate data preparation and transformation for GNN models.
5. Integration with PyTorch: PyG is built on top of PyTorch and leverages its features, such as automatic differentiation, GPU support, and the extensive ecosystem of deep learning tools. This integration allows for seamless use of PyTorch optimizers, loss functions, and other modules within PyG.

4.2 Introduction to PyGOD

PyGOD (Python Graph Outlier Detection) [5] is a specialized library for outlier detection in graph data. It extends PyTorch to provide efficient and easy-to-use implementations of various graph-based anomaly detection algorithms. Some of the salient features of PyGOD are explained below:

1. Variety of Algorithms: PyGOD offers implementations of several GNN-based outlier detection methods, including DOMINANT, AnomalyDAE, CoLA, GAAN and more. These algorithms cater to different types of graph data and anomaly detection scenarios.
2. Unified Interface: PyGOD provides a unified interface for using different outlier detection algorithms, making it easier to experiment and compare their performance on the same dataset.
3. Customization and Flexibility: PyGOD allows users to customize hyperparameters and experiment with different model configurations. This flexibility is crucial for achieving optimal performance across different applications.
4. Evaluation Metrics: PyGOD supports a variety of evaluation metrics, such as precision, recall, F1-score, and area under the ROC curve (AUC), to assess the performance of outlier detection algorithms.
5. Documentation and Examples: PyGOD provides comprehensive documentation and examples to help users understand how to implement and apply the provided algorithms effectively.

4.3 Datasets

In this work, two datasets are used, Cora dataset and Citeseer dataset provided by Planetoid and are openly available under PyTorch Geometric dataset. Both datasets are used for algorithm implementation and model development. The Cora and CiteSeer datasets are popular citation network datasets, a collection of benchmark datasets, where nodes represent academic papers (documents), and edges represent citations between papers.

Although there are many datasets available within the library, the Cora and Citeseer datasets were selected for this study. The datasets are widely used for research purposes in graph-based machine learning and are well-suited for the needs of this study. These datasets serve as a well-known and frequently utilized resource for research in graph-based machine learning, particularly for node classification tasks.

4.3.1 Cora Dataset

The Cora dataset is characterized by the following attributes :

Nodes: Cora dataset consists of 2,708 nodes, each representing a scientific paper. The nodes contain features that describe the paper, such as word presence in the abstract.

Edges: There are 10556 edges in the dataset, which represent citation relationships between papers. If one paper cites another, an edge connects the two papers in the graph.

Classes: The papers are categorized into seven distinct classes based on their research topics. The dataset has 7 classes starting from 0 to 6 as the index number of classes.

Features: Each node in the graph is described by a feature vector of length 1,433. The feature vector is a numerical representation indicating the presence or absence of certain words in the paper's abstract. If any feature (which is word) is present, the feature value will have some numerical value or else 0.

Graph: Cora dataset represents one single static attributed graph consisting of 2708 nodes and 10556 edges with every node having 1433 features. The edges are directed and unweighted meaning any edge has a source and destination node but does not carry any weight.

The graphical representation of Cora dataset is presented in figure 17.

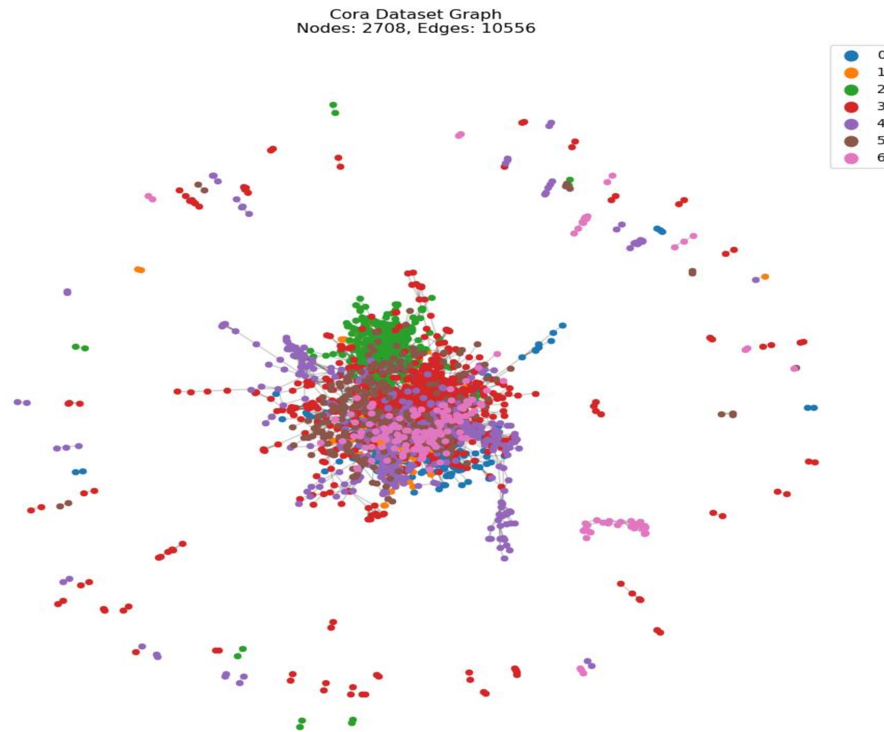


Figure 17: Graphical representation of Cora dataset with 2708 nodes and 10556 edges. Different colors indicate the nodes belonging to 7 different class of the dataset. Labels of classes are indicated in numbers from 0 to 6.

4.3.2 CiteSeer Dataset

The CiteSeer dataset contains the following attributes :

Nodes: There are 3327 nodes in Citeseer dataset, each representing an academic paper. Each node (paper) has a feature vector, typically a bag-of-words representation of the document's text.

Edges: Citeseer dataset has 9104 edges representing the citation relationships between papers. If one paper cites another, an edge connects the two papers in the graph.

Classes: Each papers are categorized into six distinct classes based on their research topics meaning the dataset has total of 6 classes starting from 0 to 6 as the index number of classes.

Features: Each node in the graph of CiteSeer dataset is described by a feature vector of length 3703. Here, the feature vector for each node is typically a bag-of-words representation of the document's text.

Graph: CiteSeer dataset is one single static attributed graph with 3327 nodes and 9104 edges with 3703 features for each node. Like Cora, the edges here are also directed and unweighted implying that any edge has a source and destination node but does not carry any weight.

The graphical representation of CiteSeer dataset is presented in figure 18.

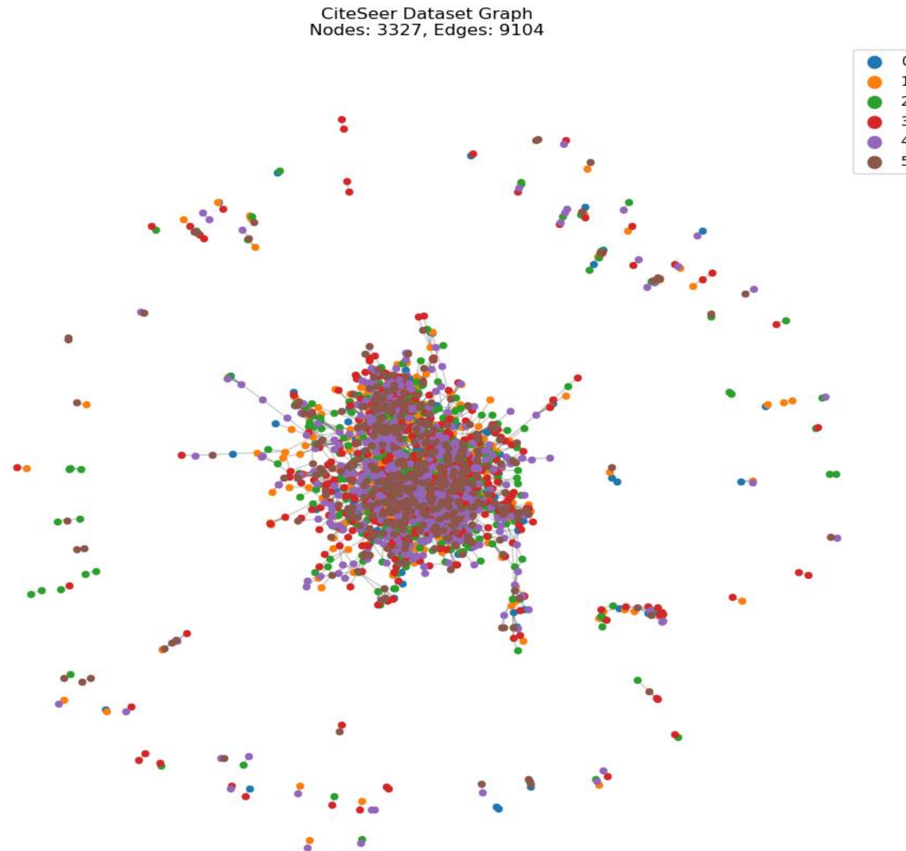


Figure 18: Graphical representation of CiteSeer dataset with 3327 nodes and 9104 edges. Different colors indicate the nodes belonging to 6 different class of the dataset. Labels of classes are indicated in numbers from 0 to 6.

4.3.3 Injection of outliers in the datasets

As both Cora and CiteSeer dataset do not contain any labeled outliers, manual injection of outliers is carried out to create a more realistic scenario for testing and benchmarking different outlier detection algorithms. By introducing known outliers into the dataset, the performance of various algorithms can be assessed and compared based on ground truth data. This setup also provides a basis for training and validating models in supervised learning settings, ensuring that models can be efficiently trained to identify and handle anomalous data points.

In both datasets, Cora and Citeseer, two types of outliers are manually injected, contextual outliers and structural outliers. Contextual outliers are injected into the graph by modifying the features of randomly selected nodes. Structural outliers, on the other hand, are injected by adding or removing the edges in the graph between randomly selected two nodes.

Contextual Outliers (node-level outliers): Contextual outliers are introduced by modifying the features of randomly selected nodes in the graph. This can also be seen as injecting the node-level outliers. The main motive here is to change the existing feature values of the selected nodes. Based on the number of features modified, the contextual outliers in the datasets are injected in two different ways, naming them as soft contextual outliers and hard contextual outliers.

- **Soft Contextual Outliers**: In soft contextual outliers, only 100 features out of all the node features are manipulated. To inject a soft contextual outlier into the dataset, first a node is randomly chosen. Again, 100 random features of that randomly chosen node are chosen. The randomly chosen features are altered. The alteration of features is performed by populating with any randomly generated values in the range of (0, 0.5). A total of 25 soft contextual outliers are injected into the dataset.
- **Hard Contextual Outliers**: In hard contextual outliers, all the features of node are modified. While injecting hard contextual outliers, a random node is chosen. For all the feature values of the randomly chosen node, the values are altered. A randomly generated value in the range of (0,0.5) is injected to all the features. A total of 25 hard contextual outliers are injected into the dataset.

This process of modification for 50 different random nodes alters the characteristics of the nodes, making them outliers within the graph. The nodes that are altered by this process are stored.

Structural Outliers (edge-level outliers): Structural outliers are introduced by manipulating the edges in the graph which involves adding or removing edges (based on the presence of edge) between pairs of randomly selected nodes. It is done so that adding new edges creates unexpected connections, while removing existing edges breaks connections in the graph. For each such outlier to inject, two nodes are randomly selected from the graph and the presence of edge between two selected nodes are checked. Based on the presence of edge, an edge is either added if previously not present or removed if previously present. Both the nodes involved in the edge modification are added to the set of structural outliers. This process is instantiated 40 times to get a set of 80 structural outliers.

After injecting both contextual outliers and structural outliers, the union of both the outliers are calculated to get a set of the total outliers from 50 contextual outliers and 80 structural outliers. As the process is random, total outliers might contain outliers which are both contextual and structural at the same time and also the nodes might get repeated which necessarily does not always produces 130 different outliers.

In Cora dataset after injecting the outliers, it yielded 126 total distinct outliers out of which 50 are contextual outliers(25 soft contextual and 25 hard contextual outliers), 78 are structural outliers and 2 are both contextual and structural outliers. The graphical representation of Cora dataset after manipulating 126 nodes as outliers is shown in figure 19.

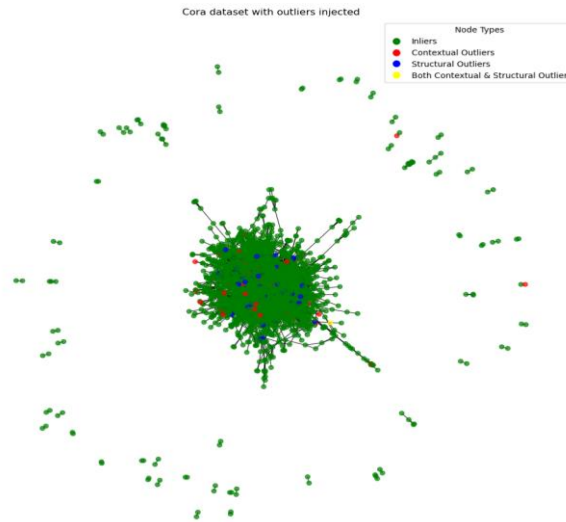


Figure 19: A graphical representation of Cora dataset after injecting 128 outliers (50 contextual, 78 structural and 2 both). Different colors denote different types of nodes, inliers and types of outlier nodes.

Likewise, in Citeseer dataset, the outlier injection process produced 128 total outliers injected in the graph. 50 of them are contextual outliers, 80 are structural outliers and 2 are both structural and contextual outlier. Citeseer dataset has 50 contextual outliers (25 soft and 25 hard contextual outliers). The graphical representation of Citeseer dataset with outlier injected is shown in figure 20 where each type of nodes (inliers, contextual outliers, structural outliers and both type of outliers) are represented with different colors.

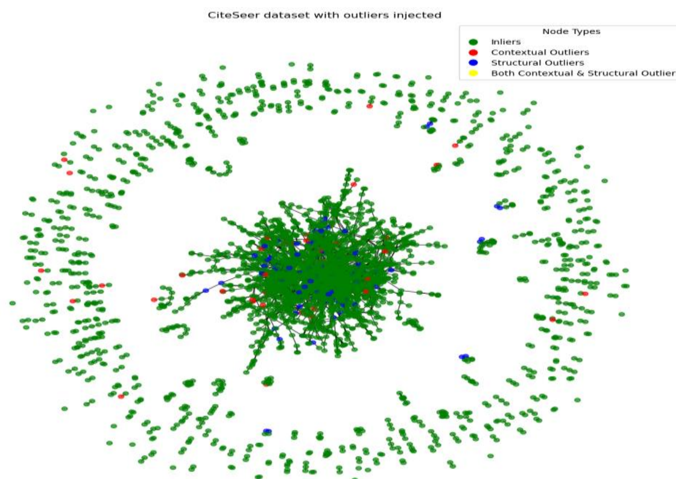


Figure 20: A graphical representation of Cora dataset after injecting 128 outliers (50 contextual, 80 structural and 2 both). Different colors denote different types of nodes, inlier and types of outlier nodes.

The summary of statistics of the datasets used is shown in table 2.

Table 2 Summary of datasets used

Datasets	Cora	CiteSeer
Number of nodes	2708	3327
Number of edges	10566	9104
Number of attributes	1433	3702
Number of contextual outliers	50	50
Number of structural outliers	78	80
Number of both contextual and structural outliers	2	2

4.4 Model Implementation

The Cora dataset with injected outliers and the CiteSeer dataset with injected outliers are used as base datasets where GNN algorithms are implemented to perform outlier detection. Both datasets are first transformed using '*NormalizeFeatures*()' as a parameter to normalize the feature matrix of the datasets. This transform essentially brings each feature in feature matrix to a consistent scale. There is also a class imbalance in the dataset, meaning 126 outlier nodes as compared to 2582 inlier nodes in Cora dataset and 128 outliers compared to 3199 inliers. However, both datasets do not have any labels associated with outliers and inliers. Also, GNN algorithms are designed to work in unsupervised learning where the models do not learn from the labeled examples but rather from normal patterns and relationships in the data. Therefore, class imbalances in both datasets (Cora dataset and CiteSeer dataset) are not addressed.

A total of 4 models are designed to implement outlier detection in Cora and Citeseer datasets. Though the datasets contain both node-level (contextual) and edge-level (structural) outliers, the primary focus of the model implementation is in the detection of node-level outliers. However, the prediction of structural outliers by the models are also considered for evaluation to see how the models behave in such context. All the models are built on node-level based outlier detection algorithms. The algorithms chosen in this work are DOMINANT, AnomalyDAE, GAAN, and CoLA, the respective models are named as *model_Dominant*, *model_AnomalyDAE*, *model_GAAN*, and *model_CoLA*. The motive here is to choose one algorithm from different GNN based methods in detecting node-level outliers in static attributed graph.

DOMINANT uses GCN-based GAE framework, AnomalyDAE uses GAT-based GAE framework, GAAN uses GAN-based GAE framework and CoLA uses GCN framework . The

detection of edge-level based outliers and dynamic graph-based outliers are out of the scope of this work.

Each model is run 10 times in both Cora and Citeseer dataset with outlier injected to detect the outliers. The average value of results of 10 such experiments is taken as the representative values for all the models in both datasets. The evaluation and comparison of all the designed models are based on the average value of 10 experiments.

4.4.1 Outlier Detection with DOMINANT

The algorithm DOMINANT [52] is readily available in PyGOD library [5]. The algorithm comes with several hyperparameters that need to be tailored for a specific task or dataset. Here, in this work, DOMINANT is imported from the library which is under *pygod.detector* [53]. A model (namely *model_Dominant*) is instantiated by creating the object of imported class DOMINANT.

The hyperparameters of the model are set manually to obtain the optimum performance. In hyperparameters settings, ReLU is set as the activation function and GCN is chosen as the GNN layer (backbone) of the model. The training batch size is set to the total number of nodes present in the datasets (2708 for Cora dataset and 3327 for CiteSeer dataset). Likewise, other hyperparameters, contamination is set to 0.1 and dropout rate (total fraction of inputs to drop to prevent overfitting) is set to 0.2. The hidden dimension (the dimension of latent space representation) is set to 128 while keeping the learning rate of the model (a hyperparameter that determines the size of the steps taken during the optimization process of training) to 0.04. The total number of GNN layers (which is GCN) in encoder is set to 2. The model is finally trained for 100 epochs. The hyperparameters were twigged manually by trying and testing the model and by referring to the documentation in [52] to get the best and optimum results.

The model is trained 10 times for both Cora and Citeseer datasets with outliers injected. The decision scores of the model for each trainings is then calculated. Decision scores are the scores assigned by the model to each node in the dataset. The higher the score, the more likely the node is to be considered as an outlier. As there are 50 contextual outliers injected in Cora dataset and CiteSeer dataset along with 126 total outliers in Cora dataset and 128 total outliers in CiteSeer outliers, two different sets of predicted outliers are calculated using two different thresholds. In Cora dataset, first (top) 50 outliers and top 126 outliers predicted by *model_Dominant* are extracted whereas in CiteSeer dataset, top 50 and top 128 outliers predicted by *model_Dominant* are extracted.

The outliers predicted by *model_Dominant* are finally checked with the ground truth, the real outliers injected in the respective datasets. Both contextual outliers and structural outliers are evaluated. In addition to this, Receiver Operating Characteristic (ROC) curve of True Positive Rate vs False Positive Rate (FPR) for all node-level outlier prediction of the model is also calculated for all threshold settings. The Area Under the ROC Curve (AUC) score of the model from ROC curve is calculated to generalize the performance of DOMINANT in outlier detection for Cora and CiteSeer dataset respectively.

4.4.2 Outlier Detection with AnomalyDAE

Similar to DOMINANT, AnomalyDAE [54] is also available in PYGOD [5] library. The method is imported and instantiated as *model_AnomalyDAE* with manually setting some of the hyperparameters referring to the documentation in [54].

The hyperparameters set during the training of *model_AnomalyDAE* are as mentioned. The activation function is chosen as ReLU, and the training batch size is set to the total number of nodes present in the datasets (2708 and 3327 for Cora and Citeseer dataset respectively). GAT is chosen as the GNN layers in encoder (backbone) while the number of GNN layers in encoder is set to 2. The learning rate of the model is set to 0.02 and the contamination parameter is set to 0.1. The hidden dimension (dimension of encoder output also called the latent space representation) is set to 128 and theta (decision threshold controlling hyperparameter) value of the model during the training is set to 1. [54]

The AnomalyDAE based model '*model_AnomalyDAE*' is then trained for Cora and CiteSeer dataset by *fit*(.) method for 10 times each. The decision scores for all the nodes in the network assigned by the model in each trainings are calculated and are checked against two different thresholds (top 50 and top 126 outliers for Cora dataset, top 50 and top 128 outliers for CiteSeer dataset) to generate the predicted outliers by the model. The outliers predicted by *model_AnomalyDAE* are finally evaluated against the real outliers injected in each of the dataset, Cora and CiteSeer. The evaluation for both contextual and structural outliers is performed in both datasets. ROC curve of TPR vs FPR for all node-level outlier prediction of the model is obtained for all threshold settings. AUC score from ROC curve is calculated.

4.4.3 Outlier Detection with GAAN

GAAN [55], another pre available algorithm for outlier detection in PYGOD [5], is imported, instantiated and trained for the datasets (Cora and CiteSeer dataset with injected outliers) for 10 times each. The decision scores calculated by GAAN-based model (*model_GAAN*) for all the nodes in the network in each trainings are checked against the thresholds (top 50 and top 126 outliers for Cora dataset, top 50 and 128 outliers for CiteSeer dataset) to produce the list of outliers predicted by the model. The outliers predicted by *model_GAAN* are then checked against the real outliers of the datasets.

GAAN based model is instantiated with several hyperparameters settings. The model's training batch size is set to 0 (0 for bull batch training [55]). The noise dimension of the model is set to 32. Similarly, hidden dimension (dimension of latent space representation) is set to128 while keeping the learning rate of the model to 0.02. The model has 4 GNN layers as 4, 2 layers for generator and 2 for discriminator. The contamination parameter of the model during training is set to 0.1 and weight (a hyperparameter for reconstruction of node feature and structure) is set to 0.5 during training of the model. The model is trained for 100 epochs. [55]

The trained model of GAAN is also used to calculate ROC curve. In ROC curve, TPR and TFR of the model for node-level outlier prediction are plotted against each other for all threshold settings. AUC score for ROC curve is calculated to generalize the outlier detecting performance of the model for both datasets, Cora and CiteSeer.

4.4.4 Outlier Detection with COLA

CoLA [47], an algorithm for outlier detection available in PYGOD [47], is implemented in a similar way to the other previously used algorithms DOMINANT [52], AnomalyDAE [54] and GAAN [55]. The CoLA model (*model_CoLA*) is imported, instantiated, and trained with Cora and CiteSeer dataset. The decision score of the model is checked with the thresholds (top 50 and top 126 for Cora dataset, top 50 and 128 for CiteSeer dataset). The outliers are predicted based on the decision score of *model_CoLA*. The model is run for 10 times for each Cora and CiteSeer dataset. The predicted outliers by the model in both datasets are checked against the real outliers.

CoLA based model is instantiated with different hyperparameters. The model has an activation function as ReLU and has 4 GCN layers in its encoder. The contamination rate of the model during training is set to 0.1 while the dropout rate is set to 0.2. The training batch size is set to the respective number of total nodes present in Cora and CiteSeer dataset. The model samples all the neighbors during training so 'num_neigh' hyperparameter is set to -1. The model is trained for 100 epochs. [47]

Once the model is trained, TPR and FPR of the model for node-level outlier detection are plotted against each other at different threshold settings to obtain ROC curve. From the ROC curve of the model, AUC score is calculated and is taken as a parameter to generalize the performance of CoLA based model for node-level outlier predictions for Cora and CiteSeer dataset.

5 Evaluation and Comparison of GNN-based algorithms

This chapter presents the results obtained from the GNN models that are designed for outlier detection. The results obtained for each of the models, *model_Dominant*, *model_AnomalyDAE*, *model_GAAN*, and *model_COLA* are discussed, analyzed, and compared.

5.1 Results of DOMINANT

A DOMINANT [52] based model, *model_Dominant,* has been implemented to predict the outliers in Cora and CiteSeer dataset with injected outliers. The outliers predicted by this model are checked against the real outliers injected into the datasets. As two different kinds of outliers are injected, contextual (node-level) and structural (edge-level) outliers, the result obtained from the model is evaluated against both the outliers.

Performance of DOMINANT model in Cora dataset

Table 3 shows the status of outliers predicted by *model_Dominant* in Cora dataset with outliers injected. As the model was ran for 10 times, the number of outliers predicted along with their types for each run is shown in table 1. The average values from 10 runs for all the predicted outliers is calculated and this average value is taken as the representative result for the model, *model_Dominant*. Table 1 shows the number of hard contextual and soft outliers, and structural outliers correctly predicted by DOMINANT model along with the number of false predictions of outliers for two thresholds (top 50 outliers and top 126 outliers). The threshold number 50 and 126 is chosen because of the fact that there are total 50 contextual outliers and 126 total outliers (contextual, structural and both combined).

As the primary goal is to evaluate the performance of model in node-level outlier detection, a list of top 50 outlier nodes predicted by the model based on the outlier scores assigned to each of the nodes during model training is calculated. This list of predicted outliers are checked with the list of real outliers injected in the dataset and it is observed that on average, 41 predicted outliers are true outliers and remaining 9 are not the true outliers. Furthermore, out of 41 correctly predicted outliers, all 25 hard contextual outliers are detected whereas 16 out of 25 soft contextual outliers are truly detected. However, in all 10 runs, *model_Dominant* predicted only 2 true structural outliers which were in reality both structural and contextual outliers. This result showed that the model is incapable of correctly predicting the structural outliers.

Likewise, considering the fact that Cora dataset has 126 total outliers injected, from a threshold value of 126, a list of top 126 outliers predicted by *model_Dominant* is extracted. It is found that upon checking upon with the list of real outlier injected in the dataset, on average 49 true outliers (22 soft contextual outliers, 25 hard contextual outliers, 2 structural outliers and 2 both contextual and structural outliers) are predicted leaving remaining 77 predicted outliers by the model as incorrect prediction.

Table 3 Status of outliers predicted by DOMINANT model in Cora dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments.

Experiment Number	Threshold : Top 50 Outliers Total number of prediction : 50				Threshold : Top 126 Outliers Total number of prediction: 126			
	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers
1	15	25	2	10	22	25	4	77
2	16	25	2	9	23	25	5	75
3	14	25	2	11	22	25	5	74
4	18	25	2	7	22	25	4	76
5	15	25	2	10	23	25	4	76
6	16	25	2	9	22	25	5	76
7	16	25	2	9	21	25	4	78
8	17	25	2	8	22	25	4	77
9	15	25	2	10	23	25	4	76
10	14	25	2	11	24	25	5	74
Average	**16**	**25**	**2**	**9**	**22**	**25**	**4**	**77**
***SCO: Soft Contextual Outliers, *HCO: Hard Contextual Outliers**. *Average is rounded up to the nearest highest integer value for all true prediction.								

In addition to the evaluation shown in table 3, ROC curve is also plotted to check the diagnostic ability of the model to predict the outliers in all threshold settings and AUC score is calculated. Figure 21 is the ROC curve for DOMINANT model taking only contextual outliers into consideration. The existence of structural outliers in the dataset is not considered. DOMINANT model has the AUC score of 0.80 in ROC curve for contextual outliers detection which is shown in figure 21.

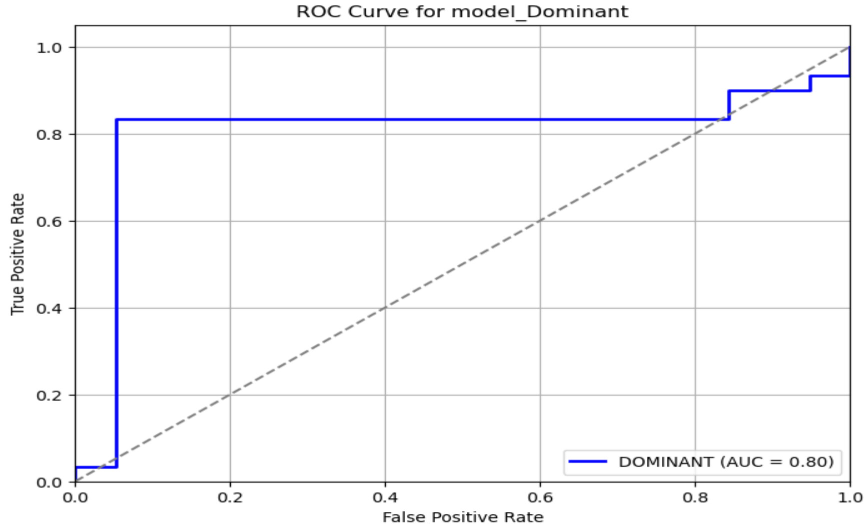


Figure 21: ROC Curve for *model_Dominant* for contextual outlier predictions in Cora dataset containing injected outliers. TPR and FPR of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination.

Performance of DOMINANT model in CiteSeer dataset

In table 4, the performance of the model in CiteSeer dataset is shown. DOMINANT model is trained for 10 times for each of the two thresholds (top 50 outliers and top 128 outliers based on the outlier scores assigned by the model). For each run, number of true soft contextual outliers, true hard contextual outliers, structural outliers, and false outliers predicted are recorded. The average value of all 10 runs for both the threshold values from table 4 suggests that *model_Dominant* is incapable of truly predicting the structural outliers. For threshold of top 50 outliers, on average, the model predicted 18 soft contextual outliers and 25 hard contextual outliers keeping the number of correct contextual outliers prediction to 43 whereas 7 are falsely predicted as outliers. Keeping the threshold value of top 128 outliers, the model is able to detect 24 soft contextual outliers, 25 hard contextual outliers essentially making the correct contextual outliers prediction number to 49 while 79 of the predictions made by the model as outliers are incorrect on average.

In figure 22, ROC curve of TPR plotted against FPR for all the threshold settings of DOMINANT model for the prediction of outliers in CiteSeer dataset is presented where the model has AUC score of 0.83. This ROC curve is only for the contextual outlier prediction of the model while the scenario of structural outliers and their detection is dropped off.

Table 4 Status of outliers predicted by DOMINANT model in CiteSeer dataset with outliers injected. The model is run for 10 times in the dataset. Average is the average value of each column for 10 experiments.

Experiment Number	Threshold : Top 50 Outliers Total number of prediction : 50				Threshold : Top 128 Outliers Total number of prediction : 128			
	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers
1	18	25	2	7	23	25	3	79
2	18	25	2	7	23	25	3	79
3	17	25	2	8	24	25	3	78
4	17	25	2	8	25	25	2	78
5	18	25	2	7	24	25	2	79
6	16	25	2	9	24	25	2	79
7	18	25	2	7	23	25	3	79
8	17	25	2	8	24	25	2	79
9	18	25	2	7	23	25	3	79
10	17	25	2	8	24	25	3	78
Average	**18**	**25**	**2**	**7**	**24**	**25**	**3**	**79**
*SCO: Soft Contextual Outliers, HCO: Hard Contextual Outliers. *Average is rounded up to the nearest highest integer value for all true prediction and to the nearest lowest integer value of false predictions.								

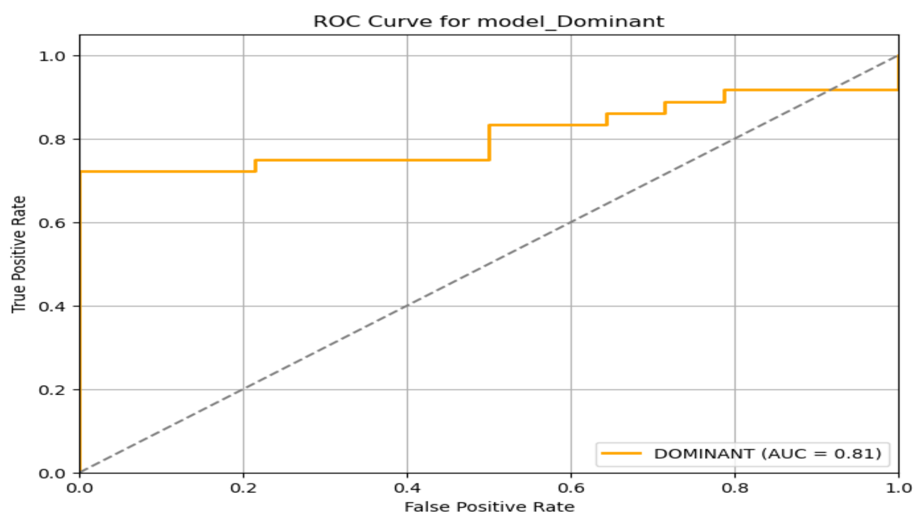


Figure 22: ROC Curve for *model_Dominant* for contextual outlier predictions in CiteSeer dataset containing injected outliers. TPR and FPR of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination.

5.2 Results of AnomalyDAE

The contextual and structural outliers predicted by *model_AnomalyDAE*, which is based on AnomalyDAE algorithm [54] are checked against the real outliers injected into the datasets, Cora and CiteSeer.

Performance of AnomalyDAE model in Cora dataset

Table 5 shows the status of outliers predicted by *model_AnomalyDAE* for two thresholds of top 50 outliers and top 126 outliers. For each run of model indicated by the experiment number in table 5, the number of true soft contextual outliers, hard contextual outliers, true structural outliers and false predicted outliers in each of the thresholds are tabulated. The average value in the table suggests that in both thresholds, the performance of the model in structural outliers prediction is negligible. In the top 50 outliers predicted by the model, 18 are true soft contextual outliers, 25 are true hard contextual outliers and 7 are falsely predicted outliers. Likewise, the model predicted 48 contextual outliers (23 soft contextual and 25 hard contextual) while 75 of the outliers predicted are not the real outliers for threshold of top 126 outliers on an average.

Ignoring the presence of structural outliers and their prediction by *model_AnomalyDAE*, ROC curve for contextual outlier detection is plotted and is shown in figure 23. TPR and FPR of outlier detection for various threshold values assigned by the model for the nodes in the graph of Cora dataset are plotted against each other to generate the ROC curve which gave the AUC score of 0.81.

Table 5 Status of outliers predicted by AnomalyDAE model in Cora dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments.

Experiment Number	Threshold : Top 50 Outliers Total number of prediction : 50				Threshold : Top 126 Outliers Total number of prediction : 126			
	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers
1	18	25	2	7	23	25	4	76
2	17	25	2	8	23	25	5	75
3	19	25	2	6	24	25	5	74
4	19	25	2	6	23	25	4	76
5	16	25	2	9	23	25	4	76
6	17	25	2	8	24	25	5	74
7	18	25	2	7	22	25	4	77
8	20	25	2	5	22	25	4	77
9	19	25	2	6	23	25	4	76
10	16	25	2	6	22	25	5	76
Average	**18**	**25**	**2**	**7**	**23**	**25**	**5**	**75**
*SCO: Soft Contextual Outliers, *HCO: Hard Contextual Outliers. *Average is rounded up to the nearest highest integer value.								

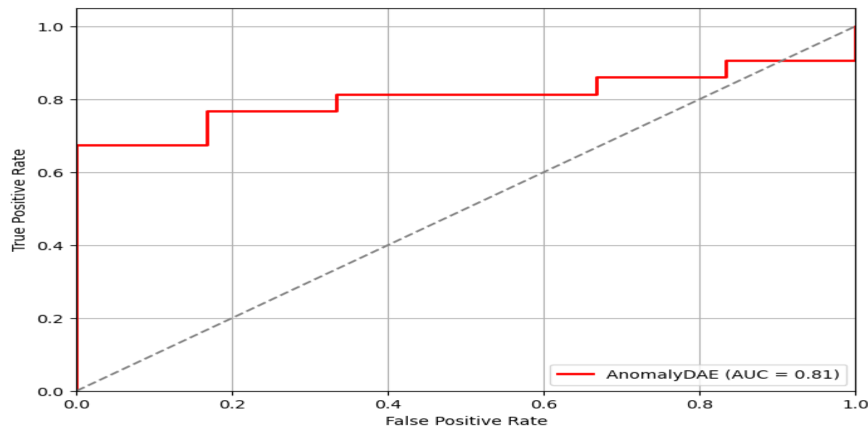


Figure 23: ROC Curve for *model_AnomalyDAE* for contextual outlier predictions in Cora dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination.

Performance of AnomalyDAE model in CiteSeer dataset

In CiteSeer dataset, the training of AnomalyDAE model for 10 runs is shown in table 6. With experiment number denoting the run of the model, the table highlights the predictions made by the model for two chosen thresholds, top 50 outliers and top 128 outliers predicted. The average value for all the predictions shows that in top 50 outliers threshold category, the model predicted 18 true soft contextual outliers and 25 hard contextual outliers which in combination is 43 true contextual outliers predictions. Remaining 7 predictions as outlier nodes by the model are incorrect. In top 128 outliers threshold section, the prediction made by the model reads 24 true soft contextual outliers, 25 true hard contextual outliers, (47 true contextual outliers prediction) and 79 incorrectly predicted outliers. In both of thresholds, the model is incapable of predicting the structural outliers. The concern of structural outlier detection is kept aside and only the contextual outlier prediction is considered to build a ROC curve. In ROC curve, true positive rate and false positive rate of AnomalyDAE model to predict the contextual outliers in all the threshold settings are plotted against each other to obtain the AUC score of the model for CiteSeer dataset. The AUC score of AnomalyDAE based model is obtained to be 0.84 as shown in figure 2. that represents the ROC curve of AnomalyDAE model.

Table 6 Status of outliers predicted by AnomalyDAE model in CiteSeer dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments.

Experiment Number	Threshold : Top 50 Outliers Total number of prediction : 50				Threshold : Top 128 Outliers Total number of prediction : 128			
	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers
1	17	25	2	8	23	25	3	79
2	18	25	2	7	23	25	3	79
3	18	25	2	7	24	25	3	78
4	17	25	2	8	25	25	2	78
5	16	25	2	9	24	25	2	79
6	19	25	2	6	24	25	2	79
7	17	25	2	8	23	25	3	79
8	18	25	2	7	24	25	2	79
9	18	25	2	7	23	25	3	79
10	17	25	2	8	24	25	3	78
Average	**18**	**25**	**2**	**7**	**24**	**25**	**3**	**79**
*SCO: Soft Contextual Outliers, *HCO: Hard Contextual Outliers. *Average is rounded up to nearest highest integer value for all true prediction and to the nearest lowest integer value of false predictions								

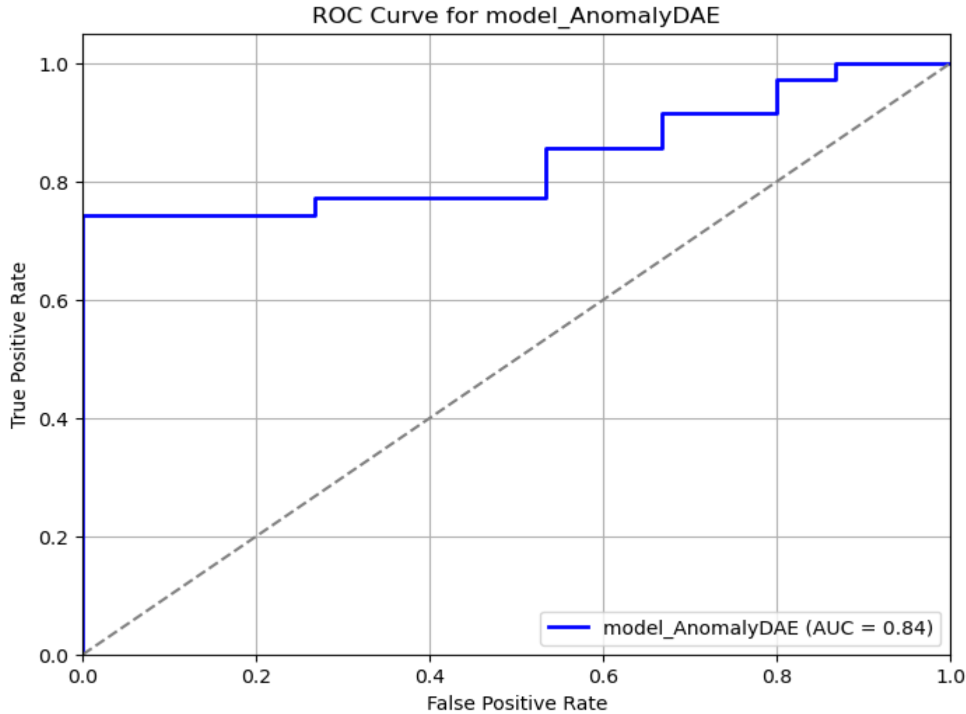


Figure 24: ROC Curve for *model_AnomalyDAE* for contextual outlier predictions in CiteSeer dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination.

5.3 Results of GAAN

In the datasets, Cora dataset and CiteSeer dataset where both of them contains manually injected outliers, GAAN [55] based model, *model_GAAN*, is trained. The model is trained for 10 times in both the datasets. For all of 10 trainings, the outliers predicted by the model is recorded.

Performance of GAAN in Cora dataset

Table 7 is the summary of the model performance in Cora dataset. The experiment number denotes the run of the model. Two different threshold values, top 50 outliers and top 126 outliers are set to produce the lists of outliers predicted by the model. The average values in table 7 shows that in Cora dataset, for threshold value of top 50 outliers, *model_GAAN* predicted 8 true soft contextual outliers, 25 hard contextual outliers, 2 structural outliers (which are essentially both contextual and structural outliers) leaving 17 as the false predictions on average. Similarly, on average, for threshold of top 126 outliers, *model_GAAN* predicted 21 true soft contextual outliers, 25 hard contextual outliers, 3 structural outliers while making 79 false outlier predictions. The model fails to predict structural outliers present in the dataset.

Keeping the focus on only contextual outlier predictions and taking out the equation of structural outliers, ROC curve of the model for contextual outlier detection in Cora dataset is plotted. TPR and FPR of the model in different threshold settings is plotted against each other in ROC curve as shown in figure 25. From figure 25, the AUC score of GAAN model is found out to be 0.74.

Table 7 Status of outliers predicted by GAAN model in Cora dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments.

Experiment Number	Threshold : Top 50 Outliers Total number of prediction : 50				Threshold : Top 126 Outliers Total number of prediction : 126			
	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers
1	8	25	2	17	22	25	3	78
2	9	25	2	16	22	25	4	77
3	8	25	2	17	21	25	3	79
4	7	25	2	18	20	25	3	80
5	7	25	2	18	22	25	3	78
6	8	25	2	17	20	25	3	80
7	10	25	2	15	21	25	3	79
8	7	25	2	18	22	25	4	77
9	8	25	2	17	21	25	3	79
10	7	25	2	18	22	25	3	78
Average	**8**	**25**	**2**	**17**	**21**	**25**	**3**	**79**

*SCO: Soft Contextual Outliers, HCO: Hard Contextual Outliers.

*Average is rounded up to nearest highest integer value for all true prediction and to the nearest lowest integer value of false predictions

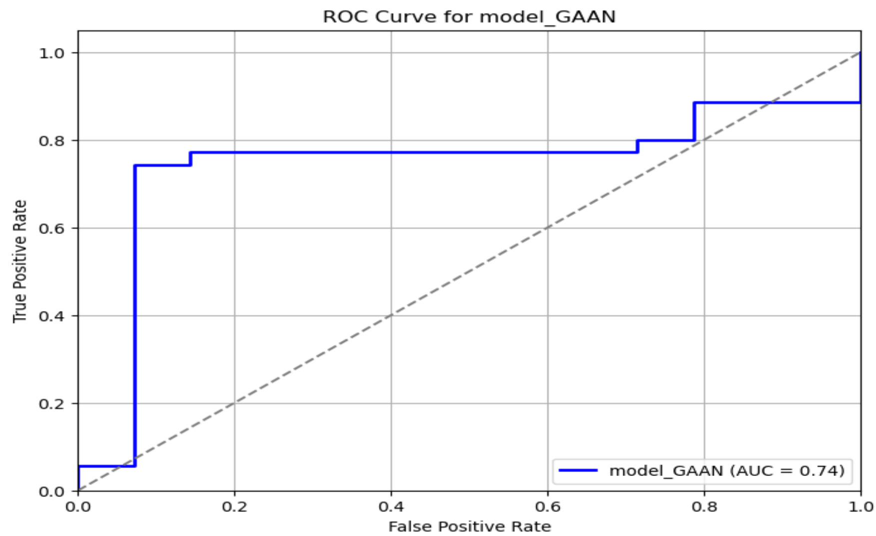


Figure 25: ROC Curve for *model_GAAN* for contextual outlier predictions in Cora dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination.

Performance of GAAN in CiteSeer dataset

In table 8, the summary of GAAN *based* model, *model_GAAN* in detecting the outliers in CiteSeer dataset with injected outliers is presented. Considering the average values of 10 runs of model from table 8, for threshold of top 50 outliers, GAAN model predicted 8 true soft contextual outliers, 25 hard contextual outliers, 2 structural outliers and 17 incorrectly predicted nodes as outliers. Also, for threshold of 128 outliers, there were total of 82 incorrectly predicted outliers, 21 true predicted soft contextual outliers, 25 true predicted hard contextual outliers, and 2 true predicted structural outliers. The model essentially predicted 33 true contextual outliers in top 50 outliers threshold and 46 true contextual outliers in top 128 outliers threshold. On the other hand, the model fails to predict structural outliers present in CiteSeer dataset.

For the model's ability to predict contextual outliers while keeping aside structural outliers, ROC curve is further plotted to generalize the performance. For all the threshold settings to determine the outliers in the dataset, model's TPR and FPR are plotted against each other to obtain the ROC curve of *model_GAAN* as shown in figure 26. *GAAN* model has 0.78 AUC score.

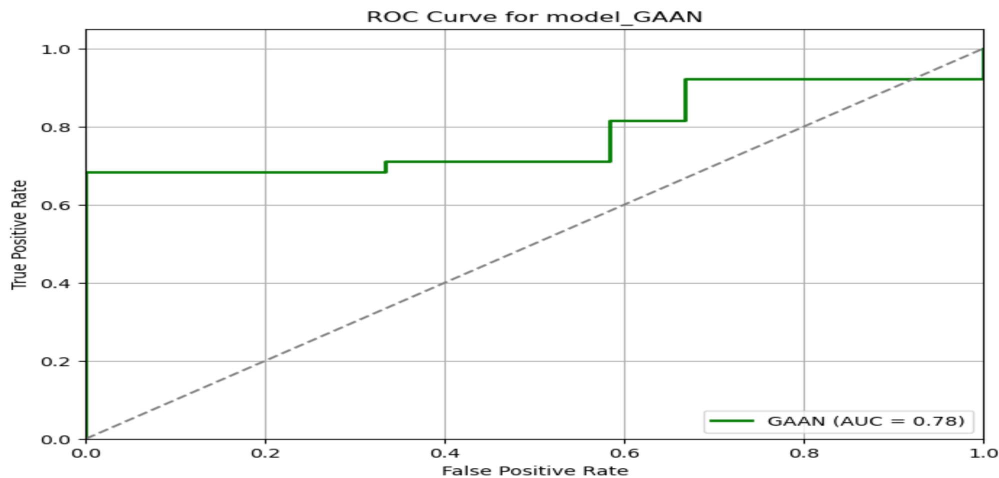


Figure 26: ROC Curve for *model_GAAN* for contextual outlier predictions in CiteSeer dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination.

Table 8 Status of outliers predicted by GAAN based model in CiteSeer dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments.

Experiment Number	Threshold : Top 50 Outliers Total number of prediction : 50				Threshold : Top 128 Outliers Total number of prediction : 128			
	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers
1	9	25	2	16	21	25	2	82
2	8	25	2	17	22	25	2	81
3	8	25	2	17	21	25	2	82
4	7	25	2	18	22	25	2	81
5	7	25	2	18	22	25	2	81
6	9	25	2	16	22	25	3	80
7	7	25	2	18	21	25	3	82
8	8	25	2	17	22	25	2	82
9	8	25	2	17	22	25	2	82
10	7	25	2	18	22	25	3	81
Average	**8**	**25**	**2**	**17**	**21**	**25**	**2**	**82**
*SCO: Soft Contextual Outliers, HCO: Hard Contextual Outliers. *Average is rounded up to nearest highest integer value for all true prediction and to the nearest lowest integer value of false predictions								

5.4 Results of CoLA

For *model_CoLA*, a CoLA [47] based model, Cora dataset and Citeseer dataset with manually injected outliers are used as two benchmark datasets. The model is trained 10 times for each of the datasets and the outliers in the datasets are predicted.

Performance of CoLA model in Cora dataset

Table 9 presents the status of outliers predicted by CoLA model in Cora dataset. The information of the real outliers injected in the dataset is used for the evaluation of the prediction. For each of 10 runs of model, the outliers predicted by the model is presented in table 9. The model is set for two threshold values, top 50 outliers and top 126 outliers based on the outlier scores assigned by the model. The model under threshold of top 50 outliers yielded 10 true soft contextual outliers, 25 true hard contextual outliers, 2 structural outliers and 14 false outliers on average in its top 50 predicted outliers list. Similarly, the model predicted 22 true soft contextual outliers, 25 hard contextual outliers, 4 structural outliers and 77 false outliers for its top 126 outliers threshold. In summary, *model_CoLA* had the true prediction of 35 contextual outliers in its top 50 outliers threshold and 48 true contextual outliers in its top 126 outliers threshold. The model fails to truly predict structural outliers.

ROC curve for contextual outlier prediction of CoLA model is shown in figure 27. TPR is plotted against FPR of the model for outlier detection in different threshold settings assigned by the model manually to draw the ROC curve which provides the AUC score of the model as 0.78.

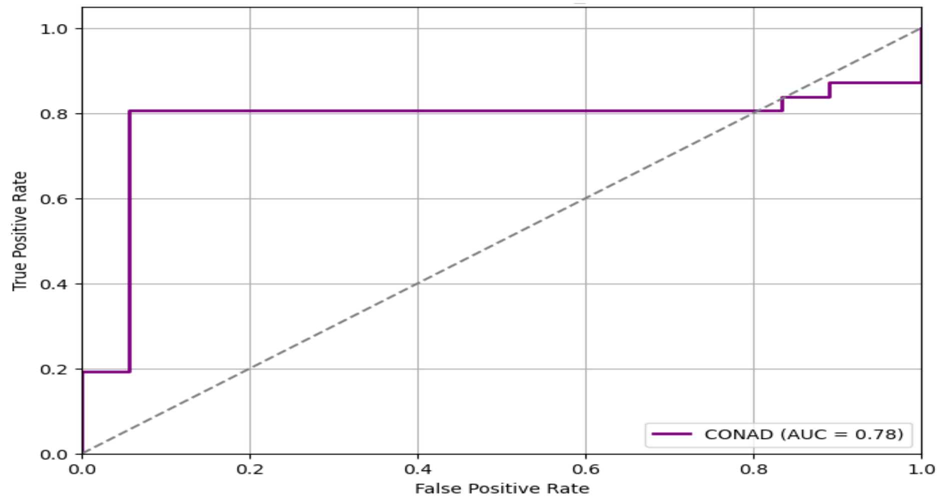


Figure 27: ROC Curve for *model_CoLA* for contextual outlier predictions in Cora dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination.

Table 9 Status of outliers predicted by CoLA model in Cora dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments.

Experiment Number	Threshold : Top 50 Outliers Total number of prediction : 50				Threshold : Top 126 Outliers Total number of prediction : 126			
	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers
1	11	25	2	14	23	25	4	76
2	12	25	2	12	22	25	3	77
3	10	25	2	13	22	25	4	77
4	10	25	2	15	23	25	4	76
5	11	25	2	14	23	25	3	77
6	9	25	2	16	22	25	4	77
7	9	25	2	16	22	25	4	77
8	10	25	2	15	22	25	5	76
9	10	25	2	15	21	25	3	79
10	11	25	2	14	23	25	3	77
Average	**10**	**25**	**2**	**15**	**22**	**25**	**4**	**77**
*SCO: Soft Contextual Outliers, *HCO: Hard Contextual Outliers. *Average is rounded up to nearest highest integer value for all true prediction and to the nearest lowest integer value of false predictions								

Performance of CoLA model in CiteSeer dataset

To predict the outliers manually injected in CiteSeer dataset, CoLA model is trained 10 times. The individual value of status of outliers detected by the model representing all the 10 runs is tabulated in table 10. The outlier detection performance of the model is performed for two threshold values, top 50 outliers and top 128 top outliers based on the outlier scores assigned during the training of the model. Taking the average value as the representative value of the model, for threshold value of top 50 outliers, *model_CoLA* predicted 37 true contextual outliers present in the dataset while its 13 predictions are false predictions. Likewise, in the list of 128 predictions given by the model during the threshold of top 128 outliers, 47 out of 50 true contextual outliers are predicted while remaining other 81 predictions made by the model are false predictions. The model is completely incapable of predicting structural outliers.

ROC curve is drawn by plotting TPR against FPR for contextual outliers prediction against each other for different threshold settings of outlier scores of the model. Figure 28 is the ROC curve representation of the model. From ROC curve, the AUC score is 0.80 as shown in figure 28.

Table 10 Status of outliers predicted by CoLA model in CiteSeer dataset with outliers injected. The model is run for 10 instances in the dataset. Average is the average value of each column for 10 experiments.

Experiment Number	Threshold : Top 50 Outliers Total number of prediction : 50				Threshold : Top 128 Outliers Total number of prediction : 128			
	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers	True SCO Outliers Predicted	True HCO Outliers Predicted	True Structural Outliers Predicted	False Predicted Outliers
1	12	25	2	13	22	25	2	81
2	12	25	2	13	22	25	2	81
3	13	25	2	12	22	25	3	80
4	13	25	2	12	21	25	2	83
5	12	25	2	13	21	25	2	83
6	14	25	2	11	21	25	4	80
7	11	25	2	14	22	25	1	81
8	12	25	2	13	21	25	2	82
9	13	25	2	12	21	25	2	82
10	11	25	2	14	23	25	2	80
Average	**12**	**25**	**2**	**13**	**22**	**25**	**2**	**81**

*SCO: Soft Contextual Outliers, *HCO: Hard Contextual Outliers.

*Average is rounded up to nearest highest integer value for all true prediction and to the nearest lowest integer value of false predictions

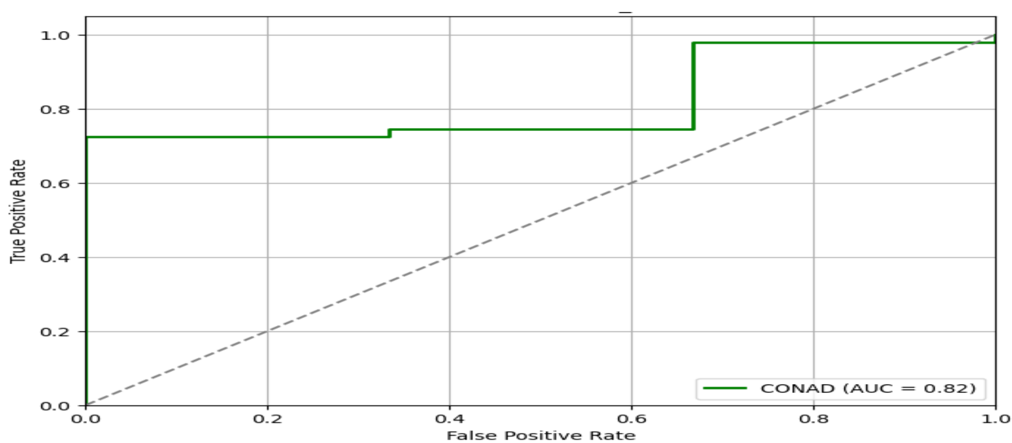


Figure 28: ROC Curve for *model_CoLA* for contextual outlier predictions in CiteSeer dataset containing injected outliers. True Positive Rate and False Positive Rate of the model are plotted against each other at various threshold of outlier scores assigned by the model. The diagonal dotted line represents the line of no discrimination.

5.5 Comparison of results

The performance of all the models (*model_Dominant*, *model_AnomalyDAE*, *model_GAAN*, and *model_CoLA*) is compared in this section. Table 11 provides the comparison of the performance of all the models in detecting the outliers for threshold of top 50 outliers in Cora and CiteSeer datasets. In figure 29, the line plots of the true predictions made by all the models for contextual outlier detection at threshold value of top 50 outliers are shown. Figure 29(a) is for Cora dataset and figure 29(b) is for CiteSeer dataset.

Table 11 True Positive and False Positive of models for contextual outlier detection in Cora and CiteSeer dataset with outliers injected for threshold value of top 50 outliers

Models	Cora Dataset		CiteSeer Dataset	
	True outliers predicted	False outliers predicted	True outliers predicted	False outliers predicted
model_Dominant	41	9	43	7
model_AnomalyDAE	43	7	43	7
model_GAAN	33	15	33	17
model_CoLA	35	15	37	13
*All the values are average value taken from 10 different runs rounded up in highest nearest integer.				

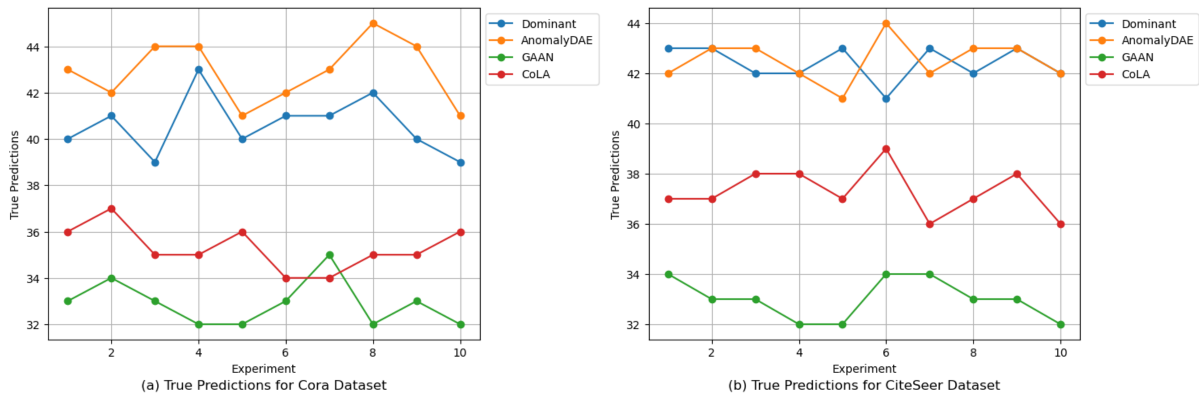


Figure 29: Line plots for true predictions of models, Dominant, AnomalyDAE, GAAN, and CoLA for contextual outlier detections in Cora and CiteSeer dataset carried out for 10 experiments keeping threshold value top 50 outliers

The AUC score of all the models for Cora and CiteSeer dataset for contextual outlier prediction is shown in table 12.

Table 12 AUC score of all the models in Cora and CiteSeer dataset for contextual outlier predictions

	Cora	CiteSeer
model_Dominant	0.80	0.83
model_AnomalyDAE	0.81	0.84
model_GAAN	0.74	0.78
model_CoLA	0.78	0.80

5.6 Discussion

Contextual outliers are the outliers that possess significant deviation from other inlier nodes in terms of node attributes. In soft contextual outliers injection, 100 node features were altered, and hard contextual outliers injection modified all the node features randomly. All models, DOMINANT, AnomalyDAE, GAAN and CoLA are able to detect hard contextual outliers while the models showed variation in soft contextual outliers detection for both Cora and CiteSeer dataset as shown in table 3, 4, 5, 6, 7, 8, 9 and 10. This high detection rate of hard contextual outliers can be attributed to the distinct and pronounced modifications made to every feature of the affected nodes, making them easier to differentiate from normal instances. Soft contextual outliers being the softer nature of contextual outliers, where only a subset of 100 features is modified, makes them more challenging to detect compared to hard contextual outliers. As far as structural outliers are concerned, these outliers involve anomalies introduced through changes in the graph structure, such as adding or removing edges. All models struggled to accurately detect structural outliers in both datasets. This difficulty arises from the complex interplay between nodes and edges, making it harder to isolate structural anomalies compared to node-level anomalies. Also, as the node features are not changed at all for all structural outliers that are injected in the datasets, the low detection rates for structural outliers suggest that the models primarily focus on node-level characteristics and struggle to effectively capture graph-level irregularities. However, as all the algorithms chosen in this work are node-level based outlier detection algorithms, the incapacity of the models to predict structural outliers does not carry any significance in models' prediction performance.

Additionally, adjusting the threshold for outlier detection influenced the number of outliers detected and the model's performance. Lower thresholds, such as the top 50 outliers, led to higher accuracy in identifying hard contextual outliers due to their more pronounced deviations. However, soft contextual outliers posed a greater challenge, resulting in varying detection rates across models.

To account for the model's performance in all thresholds, ROC curve is plotted that illustrates the trade-off between True Positive Rate (TPR) and False Positive Rate (FPR) for different threshold settings for the contextual outliers prediction. The AUC score quantifies the performance of the model across various thresholds. A higher AUC score indicates better

overall performance in distinguishing between normal and outlier instances. From table 12, AnomalyDAE has the highest AUC score in Cora dataset (0.81) followed by DOMINANT (0.8), CoLA (0.78) and GAAN (0.74). Likewise, in CiteSeer dataset, the highest AUC score order is AnomalyDAE (0.84), DOMINANT (0.83), CoLA (0.80) and GAAN (0.78). As AnomalyDAE uses dual autoencoder structure, structural autoencoder and attribute autoencoder, with each encoder in GAT framework, the node embeddings contains more structural and attribute information leading to higher discrimination ability to inliers and node-level outliers. Likewise, DOMINANT, a GCN based GAE algorithm, uses one attribute encoder based on GCN framework and two decoders, structure reconstruction decoder and attribute reconstruction decoder. This particular architecture is also equally powerful in identifying the node-level anomalies in the graph suggested by the AUC score. The generator and discriminator approach used in GAAN performed slightly lower (AUC score of 0.74 in Cora and 0.78 in CiteSeer) in node-level outlier detection as compared to AnomalyDAE and DOMINANT. This is because of the fact that the performance of model depends upon the appropriate selection of hyper parameters such as noise dimensions and number of neighbors in sampling. Fine tune of hyperparameters is therefore essential to optimize GAAN performance. Lastly, CoLA, a GCN based contrastive self-supervised method, is also equally sufficient to predict node-level outliers with AUC score of 0.78 in Cora and 0.80 CiteSeer.

In summary, while the models exhibited strong performance in identifying hard contextual outliers, they faced challenges in detecting soft contextual and structural outliers. These findings underscore the importance of considering the nature of anomalies and the complexity of graph structures when developing outlier detection models. Fine tuning of hyperparameters is also another important factor to get the optimum result for the models in outlier detection. The variations in AUC scores among the models can be attributed to differences in architectural design, learning processes, feature representations, and robustness to noise and variations in data distribution. Each model has its strengths and weaknesses, which should be considered when selecting the most suitable model for a particular outlier detection task.

6 Conclusion and Future work

This thesis work has meticulously explored the realm of GNNs and their potential applications in unraveling outlier detection within graph data. Initially, an in-depth study of graph basics is conducted to understand the fundamental aspects associated with graph theory. This study progresses as an extensive state-of-the-art examination into various GNN architectures, including GCN, GAT, GAE, GraphSAGE, GIN, VGAE, and others, to establish a solid theoretical groundwork. Additionally, the research delves into a state-of-the-art study of GNN-based outlier detection algorithms, preceded by a comprehensive analysis of the definition of outliers and their types in graph-based data. Furthermore, the primary focus is on node-level based outliers in static attributed graphs, leaving other outlier types such as edge-level outliers, sub-graph level outliers, and outliers in dynamic graphs for future exploration. The algorithms implemented for node-level based outlier detection are DOMINANT, AnomalyDAE, CoLA, and GAAN, each leveraging distinct GNN architectures and methodologies for anomaly identification. DOMINANT represents the GCN-based GAE method, AnomalyDAE represents the GAT-based GAE method, CoLA represents the GCN-based contrastive self-supervised learning method, and GAAN represents the GAN-based method.

This work has utilized the Cora dataset and CiteSeer dataset, popular benchmark citation networks, as the datasets to carry out the implementation of these GNN-based outlier detection algorithms. Since there were no outliers in both datasets, outliers are manually injected into the datasets. This is accomplished by randomly modifying node features and network structure by randomly selecting nodes to designate them as outliers in the graph representing the datasets. 50 node-level (contextual) outliers and 80 edge-level (structural) outliers are injected into the datasets (Cora and CiteSeer), serving as the ground truth for model evaluation. Four different models are designed, each corresponding to the DOMINANT, AnomalyDAE, CoLA, and GAAN algorithms. The AUC score for each model from the ROC curve is calculated to evaluate the performance of the models in node-level outlier detection. For the ROC curve, the true positive rate and false positive rate of the model in node-level outlier prediction in the datasets at different threshold settings are plotted against each other, and the AUC score is calculated. The threshold settings are chosen automatically. Out of the four models, AnomalyDAE had the highest AUC scores of 0.81 and 0.83 for the Cora and CiteSeer datasets, respectively, followed by DOMINANT, with AUC scores of 0.80 and 0.83 for the Cora and CiteSeer datasets, respectively. Likewise, CoLA had the third-best AUC scores of 0.78 for Cora and 0.80 for CiteSeer datasets, while GAAN had the lowest AUC scores out of all four models, with scores of 0.74 and 0.78 for the Cora and CiteSeer datasets, respectively. These AUC scores have essentially illuminated the reliability of all the models.

Conversely, during the injection of 50 node-level (contextual) outliers in the dataset, 25 of them were injected as soft contextual outliers, altering only the 100 features of the selected node, while 25 of them were injected as hard contextual outliers, altering each node feature of randomly selected nodes. The detection of soft contextual outliers presented a challenge across models, with variations observed in their AUC scores, underscoring the intricacies of anomaly detection in graph data, while all models detected all hard contextual outliers with ease.

Despite the inherent challenges posed by soft node-level outliers, which all models have grappled with, this study underscores the effectiveness of all the models implemented in pinpointing outliers with notable precision. These findings, coupled with insights into the limitations and avenues for enhancement, particularly in the detection of soft contextual outliers, provide a solid foundation for future advancements in GNN-based outlier detection methodologies. By bridging the chasm between theoretical underpinnings and practical

implementation, this research propels the development of more robust and adaptable algorithms, poised to tackle the complexities of complex graph data structures with quantifiable evidence of their performance.

Future Work

In paving the way for future research endeavors, particularly within the realm of static attributed graphs and node-level outlier detection, several promising avenues beckon exploration. Firstly, delving deeper into the fusion of GNN architectures with advanced techniques such as graph attention mechanisms and graph convolutional networks holds immense potential. The exploration of novel architectures that adeptly capture intricate graph structures and attribute information while mitigating the challenges posed by noise and structural outliers is paramount. Additionally, extending the scope of analysis to dynamic graph settings could offer valuable insights, enabling the development of outlier detection algorithms capable of adapting to evolving graph structures over time. Furthermore, integrating domain-specific knowledge and domain-specific features into outlier detection frameworks could enhance the robustness and interpretability of anomaly detection models. Lastly, the exploration of ensemble learning approaches, combining multiple outlier detection algorithms, could bolster performance and generalization capabilities, offering a promising avenue for future research endeavors in this domain.

References

[1] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. Wiltschko, "A Gentle Introduction to Graph Neural Networks," *Distill*, vol. 6, no. 8, p. 10.23915/distill.00033, Aug. 2021, doi: 10.23915/distill.00033.

[2] M. Newman, *Networks*. Oxford University Press, 2018.

[3] J. Zhou *et al.*, "Graph Neural Networks: A Review of Methods and Applications." arXiv, Oct. 06, 2021. Accessed: Apr. 27, 2024. [Online]. Available: http://arxiv.org/abs/1812.08434

[4] "PyTorch documentation — PyTorch 2.3 documentation." Accessed: May 13, 2024. [Online]. Available: https://pytorch.org/docs/stable/index.html

[5] "PyGOD 1.1.0 documentation." Accessed: May 13, 2024. [Online]. Available: https://docs.pygod.org/en/latest/index.html

[6] L. Wu, P. Cui, J. Pei, and L. Zhao, *Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer Nature, 2022.

[7] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks." arXiv, Dec. 02, 2015. Accessed: May 14, 2024. [Online]. Available: http://arxiv.org/abs/1511.08458

[8] S. Grossberg, "Recurrent neural networks," *Scholarpedia*, vol. 8, no. 2, p. 1888, 2013, doi: 10.4249/scholarpedia.1888.

[9] *Understanding Graph Neural Networks | Part 2/3 - GNNs and it's Variants*, (Sep. 20, 2020). Accessed: May 13, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=ABCGCf8cJOE

[10] R. van den Berg, T. N. Kipf, and M. Welling, "Graph Convolutional Matrix Completion." arXiv, Oct. 25, 2017. Accessed: Apr. 05, 2024. [Online]. Available: http://arxiv.org/abs/1706.02263

[11] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein Interface Prediction using Graph Convolutional Networks," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2017. Accessed: May 14, 2024. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/f507783927f2ec2737ba40afbd17efb5-Abstract.html

[12] S. X. Rao *et al.*, "xFraud: Explainable Fraud Transaction Detection," *Proc. VLDB Endow.*, vol. 15, no. 3, pp. 427–436, Nov. 2021, doi: 10.14778/3494124.3494128.

[13] Z. Cui, K. Henrickson, R. Ke, Z. Pu, and Y. Wang, "Traffic Graph Convolutional Recurrent Neural Network: A Deep Learning Framework for Network-Scale Traffic Learning and Forecasting." arXiv, Nov. 04, 2019. Accessed: May 14, 2024. [Online]. Available: http://arxiv.org/abs/1802.07007

[14] W. W. Lo, S. Layeghy, M. Sarhan, M. Gallagher, and M. Portmann, "E-GraphSAGE: A Graph Neural Network based Intrusion Detection System for IoT," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2022, pp. 1–9. doi: 10.1109/NOMS54207.2022.9789878.

[15] S. Zhang, H. Tong, J. Xu, and R. Maciejewski, "Graph convolutional networks: a comprehensive review," *Comput. Soc. Netw.*, vol. 6, no. 1, p. 11, Dec. 2019, doi: 10.1186/s40649-019-0069-y.

[16] M. Gori, G. Monfardini, and F. Scarselli, *A new model for Learning in Graph Domains*, vol. 2. 2005, p. 734 vol. 2. doi: 10.1109/IJCNN.2005.1555942.

[17] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Message Passing Neural Networks," in *Machine Learning Meets Quantum Physics*, K. T. Schütt, S. Chmiela, O. A. von Lilienfeld, A. Tkatchenko, K. Tsuda, and K.-R. Müller, Eds., Cham: Springer International Publishing, 2020, pp. 199–214. doi: 10.1007/978-3-030-40245-7_10.

[18] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks." arXiv, Feb. 22, 2017. Accessed: May 13, 2024. [Online]. Available: http://arxiv.org/abs/1609.02907

[19] A. Daigavane, B. Ravindran, and G. Aggarwal, "Understanding Convolutions on Graphs," *Distill*, vol. 6, no. 8, p. 10.23915/distill.00032, Aug. 2021, doi: 10.23915/distill.00032.

[20] "How powerful are Graph Convolutional Networks?" Accessed: May 13, 2024. [Online]. Available: http://tkipf.github.io/graph-convolutional-networks/

[21] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral Networks and Locally Connected Networks on Graphs." arXiv, May 21, 2014. Accessed: May 13, 2024. [Online]. Available: http://arxiv.org/abs/1312.6203

[22] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How Powerful are Graph Neural Networks?" arXiv, Feb. 22, 2019. Accessed: May 13, 2024. [Online]. Available: http://arxiv.org/abs/1810.00826

[23] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks." arXiv, Feb. 04, 2018. Accessed: May 13, 2024. [Online]. Available: http://arxiv.org/abs/1710.10903

[24] H. İ. Hatun, "Graph Neural Networks (GNNs)," Medium. Accessed: May 13, 2024. [Online]. Available: https://halil7hatun.medium.com/graph-neural-networks-gnns-1f463df4bb77

[25] *Understanding Graph Attention Networks*, (Apr. 16, 2021). Accessed: May 13, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=A-yKQamf2Fc

[26] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021, doi: 10.1109/TNNLS.2020.2978386.

[27] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2017. Accessed: May 13, 2024. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

[28] A. Majumdar, "Graph structured autoencoder," *Neural Netw.*, vol. 106, pp. 271–280, Oct. 2018, doi: 10.1016/j.neunet.2018.07.016.

[29] T. N. Kipf and M. Welling, "Variational Graph Auto-Encoders." arXiv, Nov. 21, 2016. Accessed: Apr. 08, 2024. [Online]. Available: http://arxiv.org/abs/1611.07308

[30] S. Pan, R. Hu, G. Long, J. Jiang, L. Yao, and C. Zhang, "Adversarially Regularized Graph Autoencoder for Graph Embedding." arXiv, Jan. 07, 2019. Accessed: Apr. 28, 2024. [Online]. Available: http://arxiv.org/abs/1802.04407

[31] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, “Temporal Graph Networks for Deep Learning on Dynamic Graphs.” arXiv, Oct. 09, 2020. Accessed: Apr. 04, 2024. [Online]. Available: http://arxiv.org/abs/2006.10637

[32] L. Cai *et al.*, “Structural Temporal Graph Neural Networks for Anomaly Detection in Dynamic Graphs.” arXiv, May 25, 2020. Accessed: Apr. 04, 2024. [Online]. Available: http://arxiv.org/abs/2005.07427

[33] F. E. Grubbs, “Procedures for Detecting Outlying Observations in Samples,” *Technometrics*, vol. 11, no. 1, pp. 1–21, Feb. 1969, doi: 10.1080/00401706.1969.10490657.

[34] H. Kim, B. S. Lee, W.-Y. Shin, and S. Lim, “Graph Anomaly Detection With Graph Neural Networks: Current Status and Challenges,” *IEEE Access*, vol. 10, pp. 111820–111829, 2022, doi: 10.1109/ACCESS.2022.3211306.

[35] X. Ma *et al.*, “A Comprehensive Survey on Graph Anomaly Detection with Deep Learning,” *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 12, pp. 12012–12038, Dec. 2023, doi: 10.1109/TKDE.2021.3118815.

[36] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM Comput. Surv.*, vol. 41, no. 3, pp. 1–58, Jul. 2009, doi: 10.1145/1541880.1541882.

[37] H. Kim, B. S. Lee, W.-Y. Shin, and S. Lim, “Graph Anomaly Detection With Graph Neural Networks: Current Status and Challenges,” *IEEE Access*, vol. 10, pp. 111820–111829, 2022, doi: 10.1109/ACCESS.2022.3211306.

[38] L. Akoglu, M. McGlohon, and C. Faloutsos, “oddball: Spotting Anomalies in Weighted Graphs,” in *Advances in Knowledge Discovery and Data Mining*, vol. 6119, M. J. Zaki, J. X. Yu, B. Ravindran, and V. Pudi, Eds., in Lecture Notes in Computer Science, vol. 6119. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 410–421. doi: 10.1007/978-3-642-13672-6_40.

[39] A. Grover and J. Leskovec, “node2vec: Scalable Feature Learning for Networks,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco California USA: ACM, Aug. 2016, pp. 855–864. doi: 10.1145/2939672.2939754.

[40] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, “LINE: Large-scale Information Network Embedding,” in *Proceedings of the 24th International Conference on World Wide Web*, Florence Italy: International World Wide Web Conferences Steering Committee, May 2015, pp. 1067–1077. doi: 10.1145/2736277.2741093.

[41] D. Kwon, H. Kim, J. Kim, S. C. Suh, I. Kim, and K. J. Kim, “A survey of deep learning-based network anomaly detection,” *Clust. Comput.*, vol. 22, no. S1, pp. 949–961, Jan. 2019, doi: 10.1007/s10586-017-1117-8.

[42] A. Vaswani *et al.*, “Attention Is All You Need.” arXiv, Aug. 01, 2023. Accessed: Apr. 08, 2024. [Online]. Available: http://arxiv.org/abs/1706.03762

[43] A. Kumagai, T. Iwata, and Y. Fujiwara, “Semi-supervised Anomaly Detection on Attributed Graphs.” arXiv, Feb. 27, 2020. Accessed: May 13, 2024. [Online]. Available: http://arxiv.org/abs/2002.12011

[44] Y. Pei, T. Huang, W. Van Ipenburg, and M. Pechenizkiy, “ResGCN: attention-based deep residual modeling for anomaly detection on attributed networks,” *Mach. Learn.*, vol. 111, no. 2, pp. 519–541, Feb. 2022, doi: 10.1007/s10994-021-06044-0.

[45] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition.” arXiv, Dec. 10, 2015. Accessed: May 13, 2024. [Online]. Available: http://arxiv.org/abs/1512.03385

[46] Y. Liu, Z. Li, S. Pan, C. Gong, C. Zhou, and G. Karypis, “Anomaly Detection on Attributed Networks via Contrastive Self-Supervised Learning,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 6, pp. 2378–2392, Jun. 2022, doi: 10.1109/TNNLS.2021.3068344.

[47] “CoLA - PyGOD 1.1.0 documentation.” Accessed: May 13, 2024. [Online]. Available: https://docs.pygod.org/en/latest/generated/pygod.detector.CoLA.html

[48] Z. Peng, M. Luo, J. Li, L. Xue, and Q. Zheng, “A Deep Multi-View Framework for Anomaly Detection on Attributed Networks,” *IEEE Trans. Knowl. Data Eng.*, pp. 1–1, 2020, doi: 10.1109/TKDE.2020.3015098.

[49] H. Fan, F. Zhang, and Z. Li, “AnomalyDAE: Dual autoencoder for anomaly detection on attributed networks.” arXiv, Feb. 12, 2020. Accessed: May 06, 2024. [Online]. Available: http://arxiv.org/abs/2002.03665

[50] Z. Chen, B. Liu, M. Wang, P. Dai, J. Lv, and L. Bo, “Generative Adversarial Attributed Network Anomaly Detection,” in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, Virtual Event Ireland: ACM, Oct. 2020, pp. 1989–1992. doi: 10.1145/3340531.3412070.

[51] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, “Generative Adversarial Networks: An Overview,” *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 53–65, Jan. 2018, doi: 10.1109/MSP.2017.2765202.

[52] “DOMINANT - PyGOD 1.1.0 documentation.” Accessed: May 13, 2024. [Online]. Available: https://docs.pygod.org/en/latest/generated/pygod.detector.DOMINANT.html#pygod.detector.DOMINANT

[53] “pygod.detector - PyGOD 1.1.0 documentation.” Accessed: May 13, 2024. [Online]. Available: https://docs.pygod.org/en/latest/pygod.detector.html

[54] “AnomalyDAE - PyGOD 1.1.0 documentation.” Accessed: May 13, 2024. [Online]. Available: https://docs.pygod.org/en/latest/generated/pygod.detector.AnomalyDAE.html

[55] “GAAN - PyGOD 1.1.0 documentation.” Accessed: May 13, 2024. [Online]. Available: https://docs.pygod.org/en/latest/generated/pygod.detector.GAAN.html

[56] *Understanding Graph Neural Networks | Part 1/3 - Introduction*, (Sep. 20, 2020). Accessed: May 13, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=fOctJB4kVlM

[57] A. Vaswani *et al.*, “Attention is All you Need,” in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2017. Accessed: May 14, 2024. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[58] K. Ding, J. Li, R. Bhanushali, and H. Liu, ‘‘Deep anomaly detection on attributed networks,’’ in Proc. SIAM Int. Conf. Data Mining (SDM), May 2019, pp. 594–602.

Appendices

Appendix A Task description

Faculty of Technology, Natural Sciences and Maritime Sciences, Campus Porsgrunn

FMH606 Master's Thesis

Title: Graph Neural Networks for outlier detection

USN supervisor: Leila Ben Saad

External partner: N/A

Task background:

Nowadays, data generated by networks are increasingly common and can be encountered in many applications. Graph-structured data are increasingly used to model complex systems ranging from social networks, financial nets, traffic networks, smart grid networks and sensor networks. Driven by the success of deep learning methods, recent years have seen a big interest in developing neural networks for data supported on graphs. Graph Neural Networks (GNNs) [1-3] extend the deep learning paradigm to the network setting and allow to learn hidden information from data defined on graphs. Meanwhile, detecting outliers or anomalies on graph has become a vital research problem for many applications in security-related domains, e.g., discovering suspicious financial transactions, detecting anomalous behaviors in smart grids, and unveiling malicious users in social networks. In this context, graph neural networks can be adopted to efficiently learn and detect outliers from graph [4].

Task description:

The aim of this master thesis is to implement and study graph neural network algorithms for outlier detection. The following tasks will be conducted:

1- Inspect the state-of-the-art of GNNs and understand the different GNN architectures.
2- Review the state-of-the-art of GNN algorithms for outlier detection.
3- Implement and study GNN algorithms for outlier detection using PyTorch and the Python library for graph outlier detection PyGOD [5-6].
4- Evaluate the performance of GNN based algorithms for outlier detection in datasets related a specific application.
5- Propose improvements of GNN based algorithms for outlier detection.

Requirements:

Advanced coding skills in Python, knowledge on graph theory, linear algebra, machine learning toolbox PyTorch, and Python library PyGOD are needed.

References:

[1] Sanchez-Lengeling, Benjamin; Reif, Emily; Pearce, Adam; Wiltschko, Alex", A Gentle Introduction to Graph Neural Networks". Distill. 6 (9): e33, 2021
doi:10.23915/distill.00033. ISSN 2476-0757.

[2] Tutorial 7: Graph Neural Networks,
https://colab.research.google.com/github/phlippe/uvadlc_notebooks/blob/master/docs/tutorial_notebooks/tutorial7/GNN_overview.ipynb

[3] Learning Methods on Graphs, https://pytorch-geometric.readthedocs.io/en/latest/notes/introduction.html?fbclid=IwAR3re90d9WQRJCzhpyl6xky0HKc99mTW9qkzCfSczwRph9i8nJ5HJMiSdoU#data-handling-of-graphs

[4] H. Kim, B. S. Lee, W. -Y. Shin and S. Lim, "Graph Anomaly Detection With Graph Neural Networks: Current Status and Challenges," in *IEEE Access*, vol. 10, pp. 111820-111829, 2022

[5] PyGOD, Full API Reference, https://docs.pygod.org/en/latest/

[6] A Python Library for Graph Outlier Detection (Anomaly Detection), PyGOD, https://github.com/pygodteam/pygod?fbclid=IwAR2S0TGd7w5mO_kq6MLengPuSGBCHoQWTZKDH0X4OoYbZZ-mjeTi_Pj_p0Q#wang2021one

Student category: IIA students

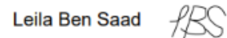
Is the task suitable for online students (not present at the campus)?: Yes

Practical arrangements: N/A

Supervision:
As a rule, the student is entitled to 15-20 hours of supervision. This includes necessary time for the supervisor to prepare for supervision meetings (reading material to be discussed, etc.).

Signatures:

Supervisor (date and signature): 01/02/2024



Student (date and signature):
Digitally Signed By
Sarthak Lamsal

SARTHAK LAMSAL
01/02/2024

Appendix B Code for Importing datasets and graph visualization for datasets

```
#importing libraries
import torch
import torch_geometric
from torch_geometric.datasets import Planetoid
from torch_geometric.transforms import NormalizeFeatures
import torch.nn.functional as F
import torch_geometric.utils as tg_utils
from matplotlib import pyplot as plt
import networkx as nx
import numpy as np
import random
from matplotlib import pyplot as plt
from pygod.detector import DOMINANT, AnomalyDAE, CoLA, GAAN



#Cora Dataset
data_cora = Planetoid(root='GNN', name='Cora', transform=NormalizeFeatures())[0]
#CiteSeer Dataset

data_citeseer = Planetoid(root='GNN', name='CiteSeer', transform=NormalizeFeatures())[0]
# Print basic information about the cora dataset
print(f'Number of nodes: {data_cora.num_nodes}')
print(f'Number of edges: {data_cora.num_edges}')  # Edges are undirected, so each edge is
counted twice
print(f'Number of features per node: {data_cora.num_node_features}')
print(f'Number of classes (labels): {data_cora.num_classes}')

# Print basic information about the citeseer dataset
print(f'Number of nodes: {data_ citeseer.num_nodes}')
```

```
print(f'Number of edges: {data_ citeseer.num_edges}')  # Edges are undirected, so each edge is counted twice
print(f'Number of features per node: {data_ citeseer.num_node_features}')
print(f'Number of classes (labels): {data_ citeseer.num_classes}')



# Convert the graph data to a NetworkX graph for visualization of cora dataset
graph = torch_geometric.utils.to_networkx(data_cora, to_undirected=True)
# Draw the graph
data = data _cora
# Define a list of colors (one for each class)
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2']
# Visualize the graph using NetworkX and matplotlib
plt.figure(figsize=(12, 12))
# pos is the layout (positioning) of the nodes
pos = nx.spring_layout(graph, seed=42)
# Draw nodes with different colors based on their class labels
for label in range(7):
    # Get nodes with the current class label
    nodes = [i for i in range(data.num_nodes) if data.y[i] == label]
    nx.draw_networkx_nodes(
        graph,
        pos,
        nodelist=nodes,
        node_size=20,
        node_color=colors[label],
        label=str(label)
    )
# Draw edges with low opacity
nx.draw_networkx_edges(graph, pos, alpha=0.2)
# Add a legend for the class labels
plt.legend(scatterpoints=1, markerscale=2, loc='upper right')
# Remove axis labels
```

```
plt.axis('off')
# Add a title that shows the number of nodes and edges
plt.title(f'Cora Dataset Graph\nNodes: {data.num_nodes}, Edges: {data.num_edges}')
# Show the graph
plt.show()


# Convert the graph data to a NetworkX graph for visualization of citeseer dataset
graph = torch_geometric.utils.to_networkx(data_citeseer, to_undirected=True)
data = citeseer_data
# Define a list of colors (one for each class)
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b'] #, '#e377c2']
# Visualize the graph using NetworkX and matplotlib
plt.figure(figsize=(12, 12))
# Initialize an empty list to store valid nodes
valid_nodes = []
# Iterate over each class label
for label in range(6):
    # Get nodes with the current class label
    nodes = [i for i in range(data.num_nodes) if data.y[i] == label]
    valid_nodes += nodes
# Create a new list to store nodes with valid positions
valid_nodes_with_positions = []
# Iterate over each node and check if it has a position
for node in valid_nodes:
    try:
        # Get the position of the node
        position = pos[node]
        valid_nodes_with_positions.append(node)
    except KeyError:
        # If the node has no position, skip it
        pass
# pos is the layout (positioning) of the nodes
pos = nx.spring_layout(graph, seed=42)
```

```
# Draw nodes with different colors based on their class labels
for label in range(6):
    # Get nodes with the current class label
    #nodes = [i for i in range(data.num_nodes) if data.y[i] == label]
    nodes = [node for node in valid_nodes_with_positions if data.y[node] == label]
    nx.draw_networkx_nodes(
        graph,
        pos,
        nodelist=nodes,
        node_size=20,
        node_color=colors[label],
        label=str(label)
    )
# Draw edges with low opacity
nx.draw_networkx_edges(graph, pos, alpha=0.2)
# Add a legend for the class labels
plt.legend(scatterpoints=1, markerscale=2, loc='upper right')
# Remove axis labels
plt.axis('off')
# Add a title that shows the number of nodes and edges
plt.title(f'CiteSeer Dataset Graph\nNodes: {data.num_nodes}, Edges: {data.num_edges}')
# Show the graph
plt.show()
```

Appendix C Outlier injection in datasets

```
def inject_outliers(graph, node_outliers, edge_outliers):
    #Define number of soft and hard contextual outliers
    num_soft_outliers = 25
    num_hard_outliers = node_outliers - num_soft_outliers
    # Set to track contextual outliers (node-level)
    contextual_outliers = set()
    soft_contextual_outliers = set()
    hard_contextual_outliers = set()

    #define feature range
    feature_range=(0, 0.5)

    # Inject soft contextual outliers
    for _ in range(num_soft_outliers):
        # Randomly select a node to modify
        node = random.choice(range(graph.num_nodes))

        # Randomly select 50% of the features to flip
        num_features_to_modify = 100
        features_to_modify = random.sample(range(graph.num_node_features), num_features_to_modify)

        # Modify the selected features
        for feature in features_to_modify:
            graph.x[node][feature] = random.uniform(feature_range[0], feature_range[1])

        # Add the node to the set of soft contextual outliers
        soft_contextual_outliers.add(node)
        # Add the node to the set of contextual outliers
        contextual_outliers.add(node)

    # Inject hard contextual outliers
```

```
    for _ in range(num_hard_outliers):
        # Randomly select a node to modify
        node = random.choice(range(graph.num_nodes))

        # Modify all features of the selected node with random values from range (0, 0.5)
        graph.x[node] = torch.tensor([random.uniform(feature_range[0], feature_range[1]) for _
in range(graph.num_node_features)])

        # Add the node to the set of hard contextual outliers
        hard_contextual_outliers.add(node)
        contextual_outliers.add(node)

    # Set to track structural outliers (edge-level)
    structural_outliers = set()

    # Convert the graph to a NetworkX graph
    nx_graph = tg_utils.to_networkx(graph, to_undirected=True)

    # Inject outliers by modifying edges
    outlier_edges = []
    for _ in range(edge_outliers):
        # Choose two random nodes to form an edge
        node1, node2 = random.sample(range(graph.num_nodes), 2)

        # Check if an edge already exists between the two nodes
        exists = nx_graph.has_edge(node1, node2)

        if exists:

            # If the edge exists, remove it to create an outlier
            graph.edge_index, _ = tg_utils.remove_edge(graph.edge_index, node1, node2)
            outlier_edges.append((node1, node2, 'removed'))

        else:
```

```
            # If the edge does not exist, add it to create an outlier
            # Add a new edge by concatenating to the existing edge_index
            new_edge = torch.tensor([[node1], [node2]])
            graph.edge_index = torch.cat([graph.edge_index, new_edge], dim=1)
            outlier_edges.append((node1, node2, 'added'))

        # Add both nodes involved in the edge modification to the set of structural outliers
        structural_outliers.add(node1)
        structural_outliers.add(node2)

    # Combine contextual and structural outliers to create total outliers
    total_outliers = contextual_outliers.union(structural_outliers)

    return graph, soft_contextual_outliers, hard_contextual_outliers, contextual_outliers, structural_outliers, total_outliers

# Inject outliers into the dataset
num_outliers = 50
edge_outliers = 40

data, sco, hco, co, so, outlier_indices = inject_outliers(data, num_outliers, edge_outliers)

# Print the indices of the injected outlier nodes
print(f"Injected contextual outlier nodes: {sorted(list(co))}")
print(f"Injected structural outlier nodes: {sorted(list(so))}")
print(f"Injected total outlier nodes: {sorted(list(outlier_indices))}")

print(f"Total number of injected outliers : {len(outlier_indices)}")
print(f"Total number of contextual outliers : {len(co)}")
print(f"Total number of structural outliers : {len(so)}")

# Convert lists to sets
set_co = set(co)
set_so = set(so)
```

```
# Find the common elements using intersection
common_elements = set_co & set_so

# Convert the set of common elements back to a list
common_elements_list = list(common_elements)

# Print the common elements
print("Total Common contextual and structural outliers:", len(common_elements_list))
print("List of Common contextual and structural outliers:", common_elements_list)

def visualize_graph(graph, contextual_outliers, structural_outliers):

    # Convert the PyTorch Geometric graph to a NetworkX graph for visualization
    nx_graph = tg_utils.to_networkx(graph, to_undirected=True)

    # Calculate inliers (nodes not in contextual or structural outliers)
    inliers = set(nx_graph.nodes()) - (contextual_outliers | structural_outliers)

    # Calculate nodes that are both contextual and structural outliers
    both_outliers = contextual_outliers & structural_outliers

    # Define colors for each type of node
    color_mapping = {
        'inliers': 'green',
        'contextual_outliers': 'red',
        'structural_outliers': 'blue',
        'both_outliers': 'yellow'  # Both contextual and structural outliers
    }

    # Define node colors based on the category
    node_colors = []
    for node in nx_graph.nodes():
        if node in both_outliers:
```

```
        node_colors.append(color_mapping['both_outliers'])
    elif node in contextual_outliers:
        node_colors.append(color_mapping['contextual_outliers'])
    elif node in structural_outliers:
        node_colors.append(color_mapping['structural_outliers'])
    else:
        node_colors.append(color_mapping['inliers'])

# Visualize the graph using NetworkX and matplotlib
plt.figure(figsize=(10, 10))

# Define a layout for the graph visualization
pos = nx.spring_layout(nx_graph, seed=42)

# Draw the graph
    nx.draw(nx_graph, pos, node_size=30, node_color=node_colors, with_labels=False,
alpha=0.7)

# Create a legend
labels = {
    color_mapping['inliers']: 'Inliers',
    color_mapping['contextual_outliers']: 'Contextual Outliers',
    color_mapping['structural_outliers']: 'Structural Outliers',
    color_mapping['both_outliers']: 'Both Contextual & Structural Outliers'
}
handles = [plt.Line2D([0], [0], marker='o', color='w', label=label, markersize=10,
markerfacecolor=color) for color, label in labels.items()]
plt.legend(handles=handles, loc='upper right', title='Node Types')

# Remove axis labels and title
plt.axis('off')

# Show the graph
plt.title('CiteSeer dataset with outliers injected')
```

```
    plt.show()


# Visualize the graph with inliers, contextual outliers, structural outliers, and both outliers
visualize_graph(data_cora, co, so)
visualize_graph(data_citeseer, co, so)
```

Appendix D Code of DOMINANT Model for outlier detection

```
#model definition
model = DOMINANT(lr=0.04,hid_dim=128,dropout=0.2,num_layers=4,theta =1.0,
batch_size=data.num_node_features, backbone =GCN, num_layers =4, contamination=0.1,
weight = 0.5, act = ReLU)

#model train
model.fit(data)

# Get the decision scores and identify outlier nodes
outlier_scores = model_.decision_score_

# Sort the outlier scores in descending order
sorted_indices = torch.argsort(outlier_scores, descending=True)
# Select the top 50 outliers based on their scores
pred_outlier_indices = sorted_indices[:50].tolist()
print('Number of outlier predicted', len(pred_outlier_indices))
# Get the outlier scores of the top 50 outliers
pred_outlier_scores = outlier_scores[pred_outlier_indices].tolist()

# Print the indices and scores of the top 50 outliers
print("Top 50 outlier indices:", sorted(pred_outlier_indices))
print("Top 50 outlier scores:", pred_outlier_scores)

# Filter out the labels corresponding to the outlier indices
pred_outlier_labels = [labels[i] for i in pred_outlier_indices]
print("Labels of top 50 predicted outliers:", pred_outlier_labels)

# Convert lists to sets
real_outliers_set = set(outlier_indices)
detected_outliers_set = set(pred_outlier_indices)

# Find the intersection of the two sets (common outliers)
common_outliers = real_outliers_set & detected_outliers_set
```

```
# Print the common outliers
print(f"Common outliers between real and detected outliers: {sorted(common_outliers)}")
print('total nummber of common outliers :',len(common_outliers))

# Find the intersection of the two sets (contextual outliers)
common_sco = sco & detected_outliers_set

# Print the common outliers
print(f"soft contextual outliers detected: {sorted(common_sco)}")
print('total nummber of soft contextual outliers :',len(common_sco))

# Find the intersection of the two sets (contextual outliers)
common_hco = hco & detected_outliers_set

# Print the common outliers
print(f"hard contextual outliers detected: {sorted(common_hco)}")
print('total nummber of hard contextual outliers :',len(common_hco))

# Convert PyTorch Geometric data to a NetworkX graph
nx_graph = tg_utils.to_networkx(data, to_undirected=True)

# Set up the plot
plt.figure(figsize=(12,12))

# Define positions for the nodes using a layout
pos = nx.spring_layout(nx_graph)

# Define colors for different types of nodes
# Normal nodes
normal_color = 'green'

# Detected outliers (nodes detected as outliers by the model)
detected_outlier_color = 'red'
```

```
# False positives (nodes detected as outliers by the model but not real outliers)
false_positive_color = 'yellow'

# Real outliers (nodes injected as outliers)
undetected_outlier_color = 'blue'

# Draw normal nodes
normal_nodes = [node for node in nx_graph.nodes if node not in co and node not in so]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=normal_nodes, node_color=normal_color, node_size=30, alpha=0.7)

# Draw true positive (true outliers predicted both contextual and structural outliers)
real_outliers_predicted = sorted(list(common_hco)) + sorted(list(common_sco))
#real_outliers_predicted = [node for node in nx_graph.nodes if node in common_hco or node in common_sco]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=real_outliers_predicted, node_color=detected_outlier_color, node_size=30, alpha=0.7)

# Draw contextual outliers not detected
undetected_contextual_outliers = [node for node in nx_graph.nodes if node in co and node not in real_outliers_predicted]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=pred_outlier_indices, node_color=undetected_outlier_color, node_size=30, alpha=0.7)

# Calculate false positives (nodes detected as outliers but not real outliers)
false_positives = [node for node in nx_graph.nodes if node in pred_outlier_indices and node not in real_outliers_predicted]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=false_positives, node_color=false_positive_color, node_size=30, alpha=0.7)

# Draw edges
nx.draw_networkx_edges(nx_graph, pos)

# Create a custom legend
legend_labels = {
```

```
    f'Inliers ({len(normal_nodes)})': normal_color,
    #f'Outliers not Detected ({len(undetected_contextual_outliers)})': detected_outlier_color,
    f'True Predicted Outliers  ({len(real_outliers_predicted)})': undetected_outlier_color,
    f'False Predicted Outliers ({len(false_positives)})': false_positive_color
}

handles = [plt.Line2D([0], [0], marker='o', color='w', label=label, markersize=10, markerfacecolor=color)
           for label, color in legend_labels.items()]
plt.legend(handles=handles, loc='upper right', title='Node Types')

# Remove axis labels and title
plt.axis('off')

plt.show()

from sklearn.metrics import roc_curve, auc

pred_oulier_label_np = np.array(pred_outlier_labels)  # True labels (0: normal, 1: outlier)
dominant_scores = np.array(pred_outlier_scores)

# Calculate ROC curve and AUC for each model
dominant_fpr, dominant_tpr, _ = roc_curve(pred_oulier_label_np, dominant_scores)
dominant_auc = auc(dominant_fpr, dominant_tpr)

# Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(dominant_fpr, dominant_tpr, color='green', lw=2, label=f'DOMINANT (AUC = {dominant_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

```
plt.title('ROC Curve for model_DOMINANT')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
```

Appendix E Code of AnomalyDAE Model for outlier detection

```
#model defintion
#model train
model = AnomalyDAE(lr=0.01,hid_dim=128,dropout=0.2, batch_size =
data.num_node_features,  backbone =GAT, num_layers =4, contamination=0.1, weight = 0.5,
act = ReLU)
model.fit(data)
# Get the decision scores and identify outlier nodes
outlier_scores = model_.decision_score_
# Sort the outlier scores in descending order
sorted_indices = torch.argsort(outlier_scores, descending=True)

# Select the top 50 outliers based on their scores
pred_outlier_indices = sorted_indices[:50].tolist()

print('Number of outlier predicted', len(pred_outlier_indices))

# Get the outlier scores of the top 50 outliers
pred_outlier_scores = outlier_scores[pred_outlier_indices].tolist()

# Print the indices and scores of the top 50 outliers
print("Top 50 outlier indices:", sorted(pred_outlier_indices))
print("Top 50 outlier scores:", pred_outlier_scores)

# Filter out the labels corresponding to the outlier indices
pred_outlier_labels = [labels[i] for i in pred_outlier_indices]
print("Labels of top 50 predicted outliers:", pred_outlier_labels)

# Convert lists to sets
real_outliers_set = set(outlier_indices)
detected_outliers_set = set(pred_outlier_indices)

# Find the intersection of the two sets (common outliers)
```

```
common_outliers = real_outliers_set & detected_outliers_set

# Print the common outliers
print(f"Common outliers between real and detected outliers: {sorted(common_outliers)}")
print('total nummber of common outliers :',len(common_outliers))

# Find the intersection of the two sets (contextual outliers)
common_sco = sco & detected_outliers_set

# Print the common outliers
print(f"soft contextual outliers detected: {sorted(common_sco)}")
print('total number of soft contextual outliers :',len(common_sco))

# Find the intersection of the two sets (contextual outliers)
common_hco = hco & detected_outliers_set

# Print the common outliers
print(f"hard contextual outliers detected: {sorted(common_hco)}")
print('total nummber of hard contextual outliers :',len(common_hco))

# Convert PyTorch Geometric data to a NetworkX graph
nx_graph = tg_utils.to_networkx(data, to_undirected=True)

# Set up the plot
plt.figure(figsize=(12,12))

# Define positions for the nodes using a layout
pos = nx.spring_layout(nx_graph)

# Define colors for different types of nodes
# Normal nodes
normal_color = 'green'

# Detected outliers (nodes detected as outliers by the model)
```

```
detected_outlier_color = 'red'

# False positives (nodes detected as outliers by the model but not real outliers)
false_positive_color = 'yellow'

# Real outliers (nodes injected as outliers)
undetected_outlier_color = 'blue'

# Draw normal nodes
normal_nodes = [node for node in nx_graph.nodes if node not in co and node not in so]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=normal_nodes, node_color=normal_color,
node_size=30, alpha=0.7)

# Draw true positive (true outliers predicted both contextual and structural outliers)
real_outliers_predicted = sorted(list(common_hco)) + sorted(list(common_sco))
#real_outliers_predicted = [node for node in nx_graph.nodes if node in common_hco or node
in common_sco]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=real_outliers_predicted,
node_color=detected_outlier_color, node_size=30, alpha=0.7)

# Draw contextual outliers not detected
undetected_contextual_outliers = [node for node in nx_graph.nodes if node in co and node not
in real_outliers_predicted]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=pred_outlier_indices,
node_color=undetected_outlier_color, node_size=30, alpha=0.7)

# Calculate false positives (nodes detected as outliers but not real outliers)
false_positives = [node for node in nx_graph.nodes if node in pred_outlier_indices and node
not in real_outliers_predicted]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=false_positives,
node_color=false_positive_color, node_size=30, alpha=0.7)

# Draw edges
nx.draw_networkx_edges(nx_graph, pos)

# Create a custom legend
```

```
legend_labels = {
    f'Inliers ({len(normal_nodes)})': normal_color,
    #f'Outliers not Detected ({len(undetected_contextual_outliers)})': detected_outlier_color,
    f'True Predicted Outliers  ({len(real_outliers_predicted)})': undetected_outlier_color,
    f'False Predicted Outliers ({len(false_positives)})': false_positive_color
}

handles = [plt.Line2D([0], [0], marker='o', color='w', label=label, markersize=10, markerfacecolor=color)
           for label, color in legend_labels.items()]
plt.legend(handles=handles, loc='upper right', title='Node Types')

# Remove axis labels and title
plt.axis('off')
plt.show()

from sklearn.metrics import roc_curve, auc

pred_oulier_label_np = np.array(pred_outlier_labels)  # True labels (0: normal, 1: outlier)
anomalydae_scores = np.array(pred_outlier_scores)

# Calculate ROC curve and AUC for each model
anomalydae_fpr, anomalydae_tpr, _ = roc_curve(pred_oulier_label_np, anomalydae_scores)
anomalydae_auc = auc(anomalydae_fpr, anomalydae_tpr)

# Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(anomaly_dae_fpr, anomaly_dae_tpr, color='red', lw=2, label=f'AnomalyDAE (AUC = {anomaly_dae_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

```
plt.title('ROC Curve for model_AnomalyDAE')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
```

Appendix F Code of CoLA Model for outlier detection

```
#model defintion
#model train
model = CoLA(lr=0.04,hid_dim=128,dropout=0.2, batch_size=data.num_node_features, backbone =GCN, num_layers =4, contamination=0.1, weight = 0.5, act = ReLU)
model.fit(data)
# Get the decision scores and identify outlier nodes
outlier_scores = model_.decision_score_
# Sort the outlier scores in descending order
sorted_indices = torch.argsort(outlier_scores, descending=True)

# Select the top 50 outliers based on their scores
pred_outlier_indices = sorted_indices[:50].tolist()

print('Number of outlier predicted', len(pred_outlier_indices))

# Get the outlier scores of the top 50 outliers
pred_outlier_scores = outlier_scores[pred_outlier_indices].tolist()

# Print the indices and scores of the top 50 outliers
print("Top 50 outlier indices:", sorted(pred_outlier_indices))
print("Top 50 outlier scores:", pred_outlier_scores)

# Filter out the labels corresponding to the outlier indices
pred_outlier_labels = [labels[i] for i in pred_outlier_indices]
print("Labels of top 50 predicted outliers:", pred_outlier_labels)

# Convert lists to sets
real_outliers_set = set(outlier_indices)
detected_outliers_set = set(pred_outlier_indices)

# Find the intersection of the two sets (common outliers)
common_outliers = real_outliers_set & detected_outliers_set
```

```
# Print the common outliers
print(f"Common outliers between real and detected outliers: {sorted(common_outliers)}")
print('total nummber of common outliers :',len(common_outliers))

# Find the intersection of the two sets (contextual outliers)
common_sco = sco & detected_outliers_set

# Print the common outliers
print(f"soft contextual outliers detected: {sorted(common_sco)}")
print('total nummber of soft contextual outliers :',len(common_sco))

# Find the intersection of the two sets (contextual outliers)
common_hco = hco & detected_outliers_set

# Print the common outliers
print(f"hard contextual outliers detected: {sorted(common_hco)}")
print('total nummber of hard contextual outliers :',len(common_hco))

# Convert PyTorch Geometric data to a NetworkX graph
nx_graph = tg_utils.to_networkx(data, to_undirected=True)

# Set up the plot
plt.figure(figsize=(12,12))

# Define positions for the nodes using a layout
pos = nx.spring_layout(nx_graph)

# Define colors for different types of nodes
# Normal nodes
normal_color = 'green'

# Detected outliers (nodes detected as outliers by the model)
detected_outlier_color = 'red'
```

```
# False positives (nodes detected as outliers by the model but not real outliers)
false_positive_color = 'yellow'

# Real outliers (nodes injected as outliers)
undetected_outlier_color = 'blue'

# Draw normal nodes
normal_nodes = [node for node in nx_graph.nodes if node not in co and node not in so]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=normal_nodes, node_color=normal_color, node_size=30, alpha=0.7)

# Draw true positive (true outliers predicted both contextual and structural outliers)
real_outliers_predicted = sorted(list(common_hco)) + sorted(list(common_sco))
#real_outliers_predicted = [node for node in nx_graph.nodes if node in common_hco or node in common_sco]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=real_outliers_predicted, node_color=detected_outlier_color, node_size=30, alpha=0.7)

# Draw contextual outliers not detected
undetected_contextual_outliers = [node for node in nx_graph.nodes if node in co and node not in real_outliers_predicted]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=pred_outlier_indices, node_color=undetected_outlier_color, node_size=30, alpha=0.7)

# Calculate false positives (nodes detected as outliers but not real outliers)
false_positives = [node for node in nx_graph.nodes if node in pred_outlier_indices and node not in real_outliers_predicted]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=false_positives, node_color=false_positive_color, node_size=30, alpha=0.7)

# Draw edges
nx.draw_networkx_edges(nx_graph, pos)

# Create a custom legend
legend_labels = {
```

```
    f'Inliers ({len(normal_nodes)})': normal_color,
    #f'Outliers not Detected ({len(undetected_contextual_outliers)})': detected_outlier_color,
    f'True Predicted Outliers  ({len(real_outliers_predicted)})': undetected_outlier_color,
    f'False Predicted Outliers ({len(false_positives)})': false_positive_color
}

handles = [plt.Line2D([0], [0], marker='o', color='w', label=label, markersize=10,
markerfacecolor=color)
        for label, color in legend_labels.items()]
plt.legend(handles=handles, loc='upper right', title='Node Types')

# Remove axis labels and title
plt.axis('off')

plt.show()

from sklearn.metrics import roc_curve, auc

pred_oulier_label_np = np.array(pred_outlier_labels)  # True labels (0: normal, 1: outlier)
cola_scores = np.array(pred_outlier_scores)

# Calculate ROC curve and AUC for each model
cola_fpr, cola_tpr, _ = roc_curve(pred_oulier_label_np, cola_scores)
cola_auc = auc(cola_fpr, cola_tpr)

# Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(cola_fpr, cola_tpr, color='green', lw=2, label=f'CoLA (AUC = {cola_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for model_CoLA')
```

```
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
```

Appendix G Code of GAAN Model for outlier detection

```
#model defintion
#model train
model_GAAN = GAAN(lr=0.04,hid_dim=128,dropout=0.2, batch_size=1024, backbone
=GIN, num_layers =4, contamination=0.1, weight = 0.5, act = ReLU)
model.fit(data)
# Get the decision scores and identify outlier nodes
outlier_scores = model_.decision_score_
# Sort the outlier scores in descending order
sorted_indices = torch.argsort(outlier_scores, descending=True)

# Select the top 50 outliers based on their scores
pred_outlier_indices = sorted_indices[:50].tolist()

print('Number of outlier predicted', len(pred_outlier_indices))

# Get the outlier scores of the top 50 outliers
pred_outlier_scores = outlier_scores[pred_outlier_indices].tolist()

# Print the indices and scores of the top 50 outliers
print("Top 50 outlier indices:", sorted(pred_outlier_indices))
print("Top 50 outlier scores:", pred_outlier_scores)
# Filter out the labels corresponding to the outlier indices
pred_outlier_labels = [labels[i] for i in pred_outlier_indices]
print("Labels of top 50 predicted outliers:", pred_outlier_labels)

# Convert lists to sets
real_outliers_set = set(outlier_indices)
detected_outliers_set = set(pred_outlier_indices)

# Find the intersection of the two sets (common outliers)
common_outliers = real_outliers_set & detected_outliers_set
```

```
# Print the common outliers
print(f"Common outliers between real and detected outliers: {sorted(common_outliers)}")
print('total nummber of common outliers :',len(common_outliers))

# Find the intersection of the two sets (contextual outliers)
common_sco = sco & detected_outliers_set

# Print the common outliers
print(f"soft contextual outliers detected: {sorted(common_sco)}")
print('total nummber of soft contextual outliers :',len(common_sco))

# Find the intersection of the two sets (contextual outliers)
common_hco = hco & detected_outliers_set

# Print the common outliers
print(f"hard contextual outliers detected: {sorted(common_hco)}")
print('total nummber of hard contextual outliers :',len(common_hco))

# Convert PyTorch Geometric data to a NetworkX graph
nx_graph = tg_utils.to_networkx(data, to_undirected=True)

# Set up the plot
plt.figure(figsize=(12,12))

# Define positions for the nodes using a layout
pos = nx.spring_layout(nx_graph)

# Define colors for different types of nodes
# Normal nodes
normal_color = 'green'

# Detected outliers (nodes detected as outliers by the model)
detected_outlier_color = 'red'
```

```
# False positives (nodes detected as outliers by the model but not real outliers)
false_positive_color = 'yellow'

# Real outliers (nodes injected as outliers)
undetected_outlier_color = 'blue'

# Draw normal nodes
normal_nodes = [node for node in nx_graph.nodes if node not in co and node not in so]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=normal_nodes, node_color=normal_color,
node_size=30, alpha=0.7)

# Draw true positive (true outliers predicted both contextual and structural outliers)
real_outliers_predicted = sorted(list(common_hco)) + sorted(list(common_sco))
#real_outliers_predicted = [node for node in nx_graph.nodes if node in common_hco or node
in common_sco]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=real_outliers_predicted,
node_color=detected_outlier_color, node_size=30, alpha=0.7)

# Draw contextual outliers not detected
undetected_contextual_outliers = [node for node in nx_graph.nodes if node in co and node not
in real_outliers_predicted]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=pred_outlier_indices,
node_color=undetected_outlier_color, node_size=30, alpha=0.7)

# Calculate false positives (nodes detected as outliers but not real outliers)
false_positives = [node for node in nx_graph.nodes if node in pred_outlier_indices and node
not in real_outliers_predicted]
nx.draw_networkx_nodes(nx_graph, pos, nodelist=false_positives,
node_color=false_positive_color, node_size=30, alpha=0.7)

# Draw edges
nx.draw_networkx_edges(nx_graph, pos)

# Create a custom legend
legend_labels = {
    f'Inliers ({len(normal_nodes)})': normal_color,
```

```
    #f'Outliers not Detected ({len(undetected_contextual_outliers)})': detected_outlier_color,
    f'True Predicted Outliers  ({len(real_outliers_predicted)})': undetected_outlier_color,
    f'False Predicted Outliers ({len(false_positives)})': false_positive_color
}

handles = [plt.Line2D([0], [0], marker='o', color='w', label=label, markersize=10,
markerfacecolor=color)
           for label, color in legend_labels.items()]
plt.legend(handles=handles, loc='upper right', title='Node Types')

# Remove axis labels and title
plt.axis('off')

plt.show()

from sklearn.metrics import roc_curve, auc

pred_oulier_label_np = np.array(pred_outlier_labels)  # True labels (0: normal, 1: outlier)
gaan_scores = np.array(pred_outlier_scores)

# Calculate ROC curve and AUC for each model
gaan_fpr, gaan_tpr, _ = roc_curve(pred_oulier_label_np, gaan_scores)
gaan_auc = auc(gaan_fpr, gaan_tpr)

# Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(gaan_fpr, gaan_tpr, color='purple', lw=2, label=f'GAAN (AUC = {gaan_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for model_GAAN')
plt.legend(loc="lower right")
```

```
plt.grid(True)
plt.show()
```