Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Università degli Studi di Padova

Department of Information Engineering

Master Degree in Computer engineering

Task-Oriented Embedding for Node Heaviness Prediction in Graphs

Supervisor:
Prof. Fabio Vandin

Master Candidate:
Matteo Salvalaio
2088249

Academic Year 2023/2024
Date 21/10/2024

Abstract

In recent years, graph analysis has gained significant traction due to the widespread presence of networks across various domains, including biology, linguistics, and social sciences. This thesis explores the intricate relationship between node embedding techniques and the accuracy of motif estimation in complex networks, highlighting the critical role that embedding methods play in understanding network structures. We introduce the concept of Node Heaviness, which quantifies a node's involvement in motifs, and propose a novel node embedding technique specifically designed for predicting node heaviness.

Through comprehensive experiments, we evaluate several embedding methods, including matrix factorization approaches, random walks, and deep learning techniques, with the goal of enhancing the accuracy of node heaviness predictions by capturing structural characteristics of nodes. Our analysis reveals varying effectiveness based on dataset characteristics and the specific nature of the graphs. While deep learning methods generally exhibit superior performance, they often struggle with dense graphs, where the complexity of the network structure can hinder their effectiveness. In contrast, our custom node embedding technique tailored for this task demonstrates adequate performance, showcasing significant potential for improvement.

By analyzing the strengths and weaknesses of different embedding strategies, this thesis contributes valuable insights to the field of network analysis. The findings underscore the importance of selecting appropriate embedding techniques for specific graph characteristics.

Contents

List of Abbreviations

Symbol	Description
G	Graph
V	Set of nodes in a graph
E	Collection of edges in a graph
$\mathcal{M}$	Motif
$N_{\mathcal{M}}(u)$	Node Heaviness of node u
ℓ	Loss function
$\mathcal{L}$	Empirical Loss
Z	Embedding matrix
A	Adjacency matrix
D	Degree matrix
T	Transition probability matrix
L	Lenght of a walk
M_g	Global similarity matrix
M_l	Local similarity matrix
I	Identity matrix
P	Permutation matrix
K	Number of iterations
W	Weights matrix
X	Multiset
n	Number of negative samples per node
d	Dimensions of the embeddings
μ	Learning rate

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, graph analysis has garnered significant interest due to the widespread presence of networks in real-world situations. We use graphs to represent information across various fields, including biology, linguistics and social sciences. [1, 2, 4, 28, 33] Modeling interplay between data on graphs has made it possible to systematically understand different network systems, such as protein-protein interaction, friendship networks, and word co-occurrence instances. These networks help with tasks like finding community groups and predicting links, which assist in understanding how rumors or diseases spread, finding new drug applications, and identifying links between genes and diseases.

The study of networks has evolved into Network Science, with applications extending to brain imaging, drug discovery, social media analysis, finance, and scientific collaborations. Unlike regular grid-like data such as images, audio, and text, which mainly originate from Euclidean spaces, network data originate from irregular, non-Euclidean domains. Therefore graphs serve as simple yet powerful and versatile models for representing and analyzing complex informations. As nonlinear structures, they provide a universal language for describing and modeling convoluted systems and making network data ever-present across diverse application fields. Consequently networks are fundamentally combinatorial structures made of interconnected nodes but without an inherent spatial context or geometric information like coordinates. Using graphs to model complex systems allows for capturing valuable high-order geometric patterns, which significantly enhances the performance of various network analysis tasks.

The widespread use of network data in many fields highlights the importance of graph analysis. Graph modeling provides a structured way to gain deep insights into complex systems, making networks essential tools for both scientific research and practical uses. Graph analytics, or network analysis, has emerged as a vibrant and influential field. Developing efficient analytics tools for graph analysis is crucial for better understanding complex networks. However, traditional methods such as path, connectivity, community, and centrality analysis primarily rely on handcrafted graph topological features extracted

from adjacency matrices. These methods can become computationally expensive and memory-intensive when applied to large-scale networks in industrial systems due to the high-dimensional and heterogeneous nature of these networks.
Graph analysis tasks generally fall into four main categories:

- **Node classification**, which involves assigning labels to nodes based on the labels of other nodes and the network's structure.
- **Link prediction**, which aims to predict missing or future links within the network.
- **Clustering**, which focuses on grouping similar nodes together.
- **Visualization**, which helps reveal the network's structure.

1.1 Graph Embeddings

Over the past few years, various approaches have been developed for the task described above. For node classification, the methods are typically divided into two groups: those that use random walks to spread labels and those that extract features from nodes to apply classifiers. Link prediction include methods like maximum likelihood models, similarity-based approaches, and probabilistic models. Clustering techniques, instead, consist of attribute-based models and those that optimize inter-cluster and intra-cluster distances. Models that were designed to solve graph-based problems usually operate on the original graph adjacency matrix or on a derived vector space.

Recently, methods that represent networks in vector spaces while preserving their properties have become increasingly popular [13, 14, 19]. These embeddings serve as a base for models, allowing parameters to be learned from training data and reducing the need for complex classification models to be applied directly to the graph. Network embedding, or representation learning, involves mapping these networks to a geometric space, such as a Euclidean space, to create an embedding space as shown in Figure 1.1. This process assigns geometric coordinates to nodes while still preserving key properties of the network. We begin by introducing several preliminary concepts, recalling the definition of Wang et al. [8].

Definition 1.1.1 (Graph)**.** *A graph $G = (V, E)$ is a structure where $V = \{v_1, \ldots, v_n\}$ represents a set of vertices (or nodes), and $E = \{e_{i,j}\}_{i,j=1}^{n}$ represents a collection of edges that link pairs of vertices.*

Definition 1.1.2 (Graph Embedding)**.** *Given a graph $G = (V, E)$, a graph embedding is a mapping $f : v_i \mapsto z_i \in \mathbb{R}^d$ for all $i \in \{1, \ldots, n\}$, such that $d \ll |V|$ and the function f preserves some proximity measure defined on graph G.*

The objective of network embedding is to generate a low-dimensional vector representation of a high-dimensional network, where the relationships between nodes are reflected in their distances within the lower-dimensional space. This vector representation help facilitating various tasks, including but not limited to: visualization, link prediction, network inference, community detection, and node classification. Additionally, embedding can reveal insights into the structure of the represented network data, which can help enhance performances by leveraging the embedding space's richer geometry. For many applications, especially those involving machine learning, converting the network into a vector space is a foundamental task.

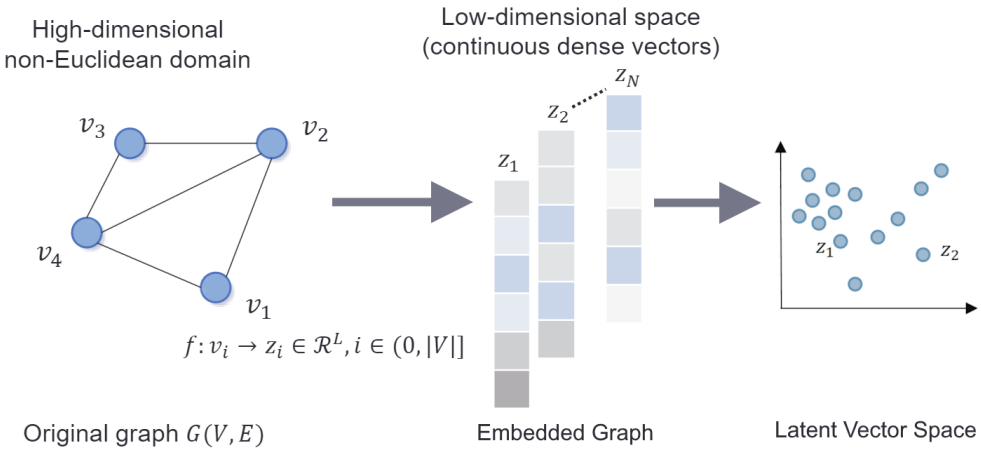


Figure 1.1: Schematic of node embedding in an undirected and unweighted graph. (Figure taken from [1])

Recent advancements in the graph embedding world, particularly on node embedding techniques, have demonstrated an impressive ability to convert high-dimensional, sparse graphs into low-dimensional, dense, and continuous vector spaces, while preserving the structural properties. These informative and nonlinear embeddings are useful for various graph analytic tasks. By encoding nodes into a latent vector space, node similarity in the original complex network can be evaluated through different similarity measures like dot product and cosine distance in the geometric embedded space. This transformation helps support faster and more accurate graph analytics compared to working directly in the high-dimensional graph domain.

Network embedding, however, still presents numerous challenges, maily due to the wide array of available techniques and the rapid evolution of the field. For decades, dimensionality reduction methods that were based on factorization matrices have served as foundational approaches for encoding topological network information. However, in the recent surge of new embedding methods, it has made increasingly difficult for professionals to stay up to date.
The abundance of diverse techniques creates a significant challenge in selecting the most

appropriate method for a specific application. There is also a notable lack of standardized tests for systematic comparison, and existing methods are often evaluated on a limited range of tasks and small real-world datasets. This makes it challenging to assess and compare the relative performance of different methods. Creating vector representations for each node in a graph is inherently complex and introduces several research-driven obstacles, further complicating the task of choosing and implementing the most effective embedding approach. Among them we have:

- **Choice of Property**: A good vector embedding should focus on maintaining the graph structure and node connections. However, determining which specific characteristics to preserve is challenging due to the numerous available distance metrics and properties. The embedding space could focus on retaining intra-community similarity, structural role similarity, or node label similarity, with the effectiveness of the embedding often depending on the application. Therefore, choosing which graph property to preserve is a fundamental question in designing network embeddings.

- **Scalability**: Embedding methods must be efficient for large-scale networks with millions of nodes and edges, processing them within a reasonable time while addressing challenges like data sparsity and low parallelizability. Scalability may be particularly challenging when the goal is to preserve global network properties that often require computationally heavy embedding methods.

- **Dimensionality of the Embedding**: Determining the optimal dimensionality requires balancing the preservation of information from the original network, which often favors higher dimensions, and the need to reduce complexity or minimize noise, which generally supports lower dimensions. While higher dimensions can enhance reconstruction precision, they also increase time and space complexity. Conversely, lower dimensions might improve performance in specific tasks, such as link prediction, by focusing on local node connections. Thus, the choice of dimensionality often depends on the application and the specific goals of the embedding.

- **Adaptability**: We need embedding methods to be versatile and capable of being applied to different types of data and tasks without needing to redo the full learning process.

- **Topology Awareness**: The embedding should ensure that the distances between nodes in the latent space accurately represent the original network's connectivity and/or homophily, meaning that similar nodes in the network should remain close to each other in the embedding space.

1.2 Purpose of the thesis

Determining the amount of motifs in a graph play a critical role in understanding the underlying structure and function of complex networks. They are used in a variety of applications, such as identifying unusual patterns in social networks and pinpointing critical functional components in biological systems. However, the direct computation of these motifs is often computationally expensive because of the vast number of possible subgraphs in large networks. As a result, accurately estimating motif recurrences has emerged as a valuable tradeoff between precision and computational complexity. In recent years, approximation techniques have gained traction, offering feasible solutions for various tasks while maintaining a balance between efficiency and accuracy.
In addition to the choice of the machine learning approach and algorithm used to predict motif counts, the quality of input data also plays a crucial role. The accuracy of these models can be significantly impacted by the quality of the input features, which raises the question of whether the effort spent on improving input data is worth the potential gains in performance.

This thesis aims to explore how input data quality impacts the accuracy of motif estimation and whether the time spent enhancing input data is warranted by the improvements in results. We seek to identify the most effective node embedding method and to evaluate whether specialized approaches for this task outperform more generic methods. To this end, we introduce a novel node embedding technique specifically designed for motif estimation and compare its performance with well-established embedding methods in the field.
We present the concept of Node Heaviness, defined as follows.

Definition 1.2.1 (Node Heaviness)**.** *Given a target motif $\mathcal{M}$ and a node in a graph $u \in V$, we define the heaviness $N_{\mathcal{M}}(\{u\})$ of the node u as the number of instances of $\mathcal{M}$ in which u is involved.*

In this thesis we will evaluate the accuracy of our methods by assessing their performance in predicting node heaviness, focusing on triangles as motifs. This approach allows us to quantify and analyze how well our node embedding techniques capture the involvement of nodes in triangles, providing insights into their effectiveness.

Chapter 2

Literature Review

Recently, graph embedding techniques have emerged as fundamental tools in representing complex graph-structured data in a lower-dimensional space, facilitating numerous tasks in machine learning and data analysis. This chapter provides a comprehensive taxonomy of graph embedding methods, aiming to categorize and elucidate the diverse approaches employed to transform graph data into meaningful vector representations. The literature reveals that graph embeddings can be broadly classified into several categories based on their underlying methodologies and applications.[6, 10]

These include classical techniques, such as matrix factorization and random walk-based methods, as well as more recent advances that leverage deep learning architectures. Classical methods, like `DeepWalk`[16] and `node2vec`[15], focus on capturing local and global structural information through random walks, while matrix factorization approaches such as `GraRep`[19] aim to decompose graph matrices into latent factors. On the other hand, emerging methods incorporate neural networks to capture more intricate patterns in graph data, exemplified by techniques like Graph Convolutional Networks (GCNs) and Graph Attention Networks (GATs) [31, 33]. By examining both established and cutting-edge techniques, this chapter seeks to offer a structured understanding of graph embedding approaches and their implications for various domains of research and application.

2.1 Taxonomy of Graph Embeddings Methods

A network embedding maps each node of a network to a latent space, typically a Euclidean vector space $\mathbb{R}^d$, where $d \ll n$ and n is the number of nodes. In this latent space, certain properties of the nodes, edges, or the entire network are preserved. This can be defined as a mapping function: $f : V \to \mathbb{R}^d$, where each node v_i is mapped to a vector z_i in the latent space. This vector z_i is expected to capture the topological properties of the original network while reducing its dimensionality.

The main goal of graph embedding is to encode nodes into a low-dimensional space so

that the similarity in this latent space approximates the similarity in the original high-dimensional graph, while preserving the graph's structural properties. Advanced graph node embedding techniques achieve this by solving an optimization problem through an unsupervised learning approach, independent of downstream prediction tasks.

The taxonomy proposed in this work is based on a mathematical perspective. It divides the methods into two main categories based on their depth: shallow embedding methods and deep learning methods. Below, we describe the characteristics of each category and provide a summary of representative approaches within each category.

Category	**Type**	**Method**	**Time Complexity**	**Property Preservation**
Shallow Embedding	Factorization	LE[14]	$O(\|E\|d^2)$	1^{st} order proximity
		GraRep[19]	$O(\|V\|^3)$	1^{st} - k^{th} order proximity
	Random Walk	node2vec[15]	$O(\|V\|d)$	1^{st} - k^{th} order proximity
	Optimization	HOPE[7]	$O(\|E\|d^2)$	1^{st} - k^{th} order proximity
Deep Learning	$--$	VGAE[18]	$O(\|V\|d + \|V\|^2)$ [34]	1^{st} order proximity
	$--$	GIN[13]	$O(\|V\|d^2 + \|E\|)$ [35]	1^{st} - k^{th} order proximity

Table 2.1: Detailed Comparison of Graph Embedding Techniques highlighting Methodological Differences, Computational Complexity, and Property Preservation

2.1.1 Property Preservation

Proximity measures are crucial for assessing how well graph embedding methods preserve the properties of the graph structure. In particular, there are three main types of proximity measures (see Table 2.1):

1. **First-Order Proximity**: The first-order proximity between two nodes, v_i and v_j, is determined by the weight of the edge s_{ij} connecting them. This weight serves as a direct measure of similarity based on existing connections. If $s_{ij} > 0$, there is positive first-order proximity between v_i and v_j; otherwise, the proximity is zero. This type of proximity reflects the local network structure and is useful for preserving direct connections between nodes. However, since many similar nodes may not be directly connected due to sparse real-world data, first-order proximity alone is insufficient for capturing all meaningful relationships.

2. **Second-Order Proximity**: Second-order proximity focuses on the similarity of the neighborhood structures of nodes. For a node v_i, its first-order proximity vector is denoted as $s_i = [s_{i_1}, \ldots, s_{i_n}]$ as show in Figure 2.1. The second-order proximity

between nodes v_i and v_j is determined by comparing these vectors s_i and s_j. This measure operates on the principle that nodes are similar if they have many common neighbors. It is particularly effective for identifying similarities between nodes even when they are not directly connected, thereby enhancing the capture of the global network structure and addressing issues of data sparsity.

3. **Higher-Order Proximity**: Higher-order proximity includes more complex measures such as the Katz Index, Rooted PageRank, and metrics based on common neighbors. While second-order proximity is often effective for most embedding methods, higher-order proximity can provide additional and more detailed insights into the global structure of the network.

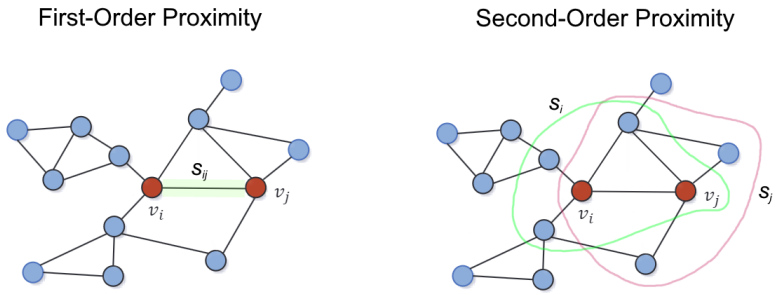


Figure 2.1: Illustration of Proximity Orders: the image illustrates the concepts of first-order proximity and second-order proximity in a graph. The diagram visually differentiates these proximities, with first-order proximity highlighted by direct edges and second-order proximity by overlapping neighborhoods.

2.1.2 Encoder-Decoder Framework

Recently, significant efforts have been made to create general frameworks that unify various embedding methods under a common mathematical formulation. Notably, Hamilton et al. [12] proposed an encoder-decoder framework that organizes embedding methods around four key components:

1. **Pairwise Similarity Function** $s_G : V \times V \to \mathbb{R}^+$, which measures the similarity between nodes in the graph.

2. **Encoder**, defined as ENC : $V \to \mathbb{R}^d$, that maps each node $v_i \in V$ to a low-dimensional vector, or embedding $z_i \in \mathbb{R}^d$.

3. **Decoder**, which is a function that takes these node embeddings and reconstructs user-specified graph statistics. A common approach is to use a pairwise decoder: DEC : $\mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$, which maps pairs of node embeddings to a real-valued similarity measure, quantifying the similarity between nodes in the original graph.

4. **Loss Function** $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, which evaluates the quality of the reconstruction.

The goal is to optimize both the encoder and decoder so that the decoded similarity between two embeddings z_i and z_j approximates a predefined graph-based similarity measure $s_G(v_i, v_j)$. This can be expressed as:

$$\text{DEC}(\text{ENC}(v_i), \text{ENC}(v_j)) = \text{DEC}(z_i, z_j) \approx s_G(v_i, v_j).$$

To achieve this, we minimize an empirical loss function over a set of training node pairs V:

$$\mathcal{L} = \sum_{(v_i, v_j) \in V} \ell(\text{DEC}(z_i, z_j), s_G(v_i, v_j)),$$

where $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ measures the discrepancy between the decoded similarity and the true similarity.

Once trained, the encoder can be used to generate embeddings for nodes, which serve as features for downstream machine learning tasks.

While many node embedding methods can be described using this framework, some approaches, particularly those involving higher-order network embeddings, extend beyond this pairwise similarity-based model. These methods, which often incorporate deep learning techniques, require additional considerations to be fully captured within this framework.

2.2 Shallow Network Embeddings

The majority of node embedding algorithms rely on what we term *shallow embedding*. In these approaches, the encoder function, which maps nodes to their corresponding vector embeddings, is simply an *embedding lookup*:

$$\text{ENC}(v_i) = Z \times v_i$$

Here, $Z \in \mathbb{R}^{d \times |V|}$ is a matrix containing the embedding vectors for all nodes, and v_i is the indicator vector associated with each node v_i (a vector of zeros except at position i, where the element is 1). The trainable parameters for shallow embedding methods are simply $\Theta_{\text{ENC}} = \{Z\}$, meaning that the embedding matrix Z is optimized directly (see Figure 2.2).

These shallow embedding approaches are largely inspired by classic matrix factorization techniques for dimensionality reduction and multi-dimensional scaling. Many of these methods were initially conceived as factorization algorithms, reinterpreted here within the encoder-decoder framework. The objective of the embedding process is to optimize the embedding matrix Z to achieve the best mapping between nodes and their corresponding embedding vectors.

The proposed taxonomy categorizes embedding methods into three broad classes:

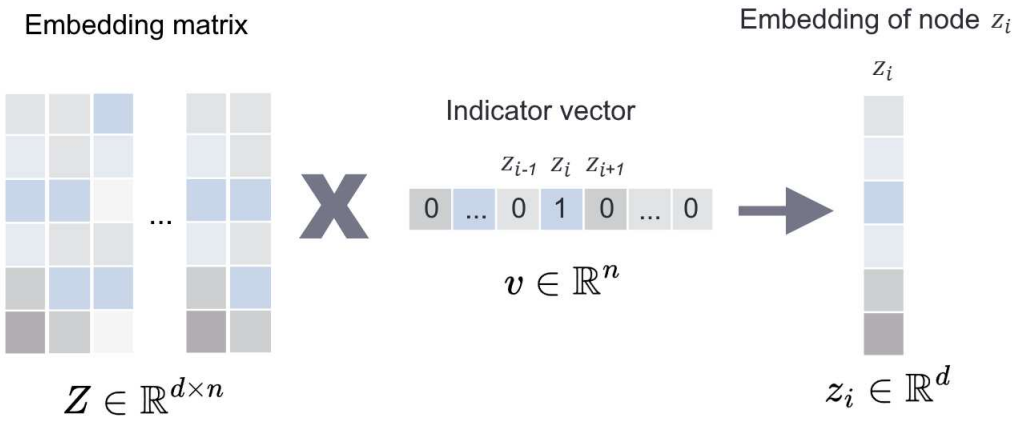


Figure 2.2: Diagram showing the process of generating a shallow embedding: the image depicts how the embedding matrix and indicator vector are used to compute the embedding of a node. The indicator vector selects the corresponding column from the embedding matrix to generate the node's embedding representation.

- **Factorization-Based Methods:** These methods involve matrix factorization techniques to decompose the adjacency matrix or similarity matrix of the graph.
- **Random Walk-Based Methods:** These methods use random walk processes to capture the graph's structural properties and node relationships.
- **Optimization-Based Methods:** These methods optimize an objective function directly over the embedding space, often involving gradient descent or similar techniques to minimize a loss function that captures the desired properties of the graph.

This framework helps to systematically categorize and understand various embedding techniques by their underlying mathematical processes and application methods.

2.2.1 Factorization-Based Methods

Early methods for learning node representations primarily relied on matrix-factorization approaches, inspired by classic dimensionality reduction techniques such as PCA and LDA, as well as non-linear methods like t-SNE and UMAP. Many embedding methodologies were developed using an inner-product decoder, where the strength of the relationship between two nodes is proportional to the dot product of their embeddings. These techniques utilize a mean-squared-error (MSE) loss function, differing mainly in how they define node similarity.

Matrix factorization-based approaches aim to learn embeddings that approximate a deterministic measure of node similarity by factorizing matrices representing node connections, such as the adjacency matrix or Laplacian matrix. The factorization technique

used depends on the matrix properties; for instance, positive semi-definite matrices like the Laplacian allow for eigenvalue decomposition, while unstructured matrices may require gradient descent or Singular Value Decomposition (SVD).

These methods have demonstrated effectiveness in graph reconstruction but tend to overfit to the adjacency matrix, limiting their utility in tasks such as link prediction. Additionally, their computational complexity, typically $O(|V|^2)$, poses challenges for scaling to large networks. Moreover, their reliance on adjacency matrix-based similarity measures, which primarily capture local connections, can further restrict their applicability in broader tasks.

Laplacian Eigenmaps

One of the earliest and most notable methods is the Laplacian Eigenmaps [14] (`LE`) technique. It is a technique aimed at embedding a network such that nodes close in the original graph remain close in a low-dimensional embedding space. Within the encoder-decoder framework, `LE` is a shallow embedding approach where the decoder measures the squared Euclidean distance between node embeddings. The loss function is weighted by the similarity of nodes within the graph, focusing on preserving local structures.

This is achieved by preserving a similarity measure defined by the weights between nodes, encoded in a weight matrix W, where W_{ij} represents the weight between nodes i and j. The optimization objective is:

$$\mathcal{L} = \sum_{v_i, v_j \in V} \mathrm{DEC}(z_i, z_j) \cdot s_G(v_i, v_j)$$

where $\mathrm{DEC}(z_i, z_j) = \|z_i - z_j\|_2^2$ and $s_G(v_i, v_j) = W_{ij}$. If the graph is unweighted this value is equal to the adjacency matrix A_{ij}.

It's important to note that `LE` uses a quadratic decoder function, which can penalize small distances between embedded nodes, potentially disrupting the preservation of local topology. Despite this, the algorithm seeks to keep the embedding of two nodes close when their corresponding weight W_{ij} is high. Laplacian Eigenmaps are particularly useful in cases where high-dimensional data lies on a low-dimensional manifold. The algorithm is simple, involving local computations and solving a sparse eigenvalue problem.

GraRep

`GraRep` [19] is an advanced graph embedding method that extends the skip-gram model to capture higher-order node similarity by considering k-step neighbors, where $1 \leq k \leq K$ and K represents the highest order. The method is designed to handle both direct node connections and more distant relationships by progressively increasing the value of k. For

each k, `GraRep` aims to minimize the following loss function:

$$\mathcal{L} = \sum_{v_i, v_j \in V} T_{ij}^k \log(\sigma(x_i^T x_j)) + \lambda \mathbb{E}_{v_j \sim p_k(V)}[\log(\sigma(-x_i^T x_j))]$$

Definition 2.2.1 (Transition Probability Matrix)**.** *Given a graph $G = (V, E)$ with nodes V, edges E, an adjacency matrix A representing the connections between nodes and the degree matrix D representing the number of neighbour of each node, the transition probability matrix T is defined as follows:*

$$T = A \cdot D^{-1}$$

Here, x_i and x_j are the vector representations of nodes v_i and v_j, respectively, $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function, and $p_k(V)$ is the noise distribution over the nodes, used for negative sampling. The term λ controls the number of negative samples.

`GraRep` transforms this optimization problem into a matrix factorization problem. Specifically, the k-step loss function is computed using the matrix X_k which represents the log-transformed k-step transition probabilities, adjusted by a threshold β. Negative values are set to zero, ensuring that only significant similarities are retained.

Next, the low-dimensional embeddings for each k-step are obtained by applying Singular Value Decomposition (SVD) to X_k, producing a matrix C_k. The final node representation is formed by concatenating the embeddings across all k-steps:

$$C = [C_1, C_2, \ldots, C_K]$$

This concatenated representation integrates information from multiple scales, capturing both local and global structure within the graph. While `GraRep` is effective in preserving high-order proximities, its scalability is limited due to the potential size of T^k, which can have $O(|V|^2)$ non-zero entries.

`GraRep` is especially useful for tasks that require capturing complex relationships in graphs, but it faces challenges in scaling to large networks due to the computational demands of handling high-order transition matrices.

2.2.2 Random Walks-Based Methods

Graph embedding methods based on random walks are highly effective for solving a wide range of graph-related tasks. However, the growing volume of research in this area has made it increasingly challenging to compare different approaches and pinpoint areas for further improvement in the field.

Random walks have been employed to approximate various properties in graphs, such as node centrality and similarity. They are particularly useful when only a partial view of the graph is available or when the graph is too large to measure completely. Embedding

techniques that use random walks to derive node representations have been proposed, with `DeepWalk` [16] and `node2vec` [15] being notable examples.

The core idea of random walk embedding is to encode the scores from the random walk into an embedding space. Random walks serve as stochastic similarity measures for various problems, including content recommendation, community detection, and image segmentation. Specifically, node similarity in complex graphs can be defined as the probability of reaching a node v_j from a source node v_i during a random walk of length l (see Figure 2.3).

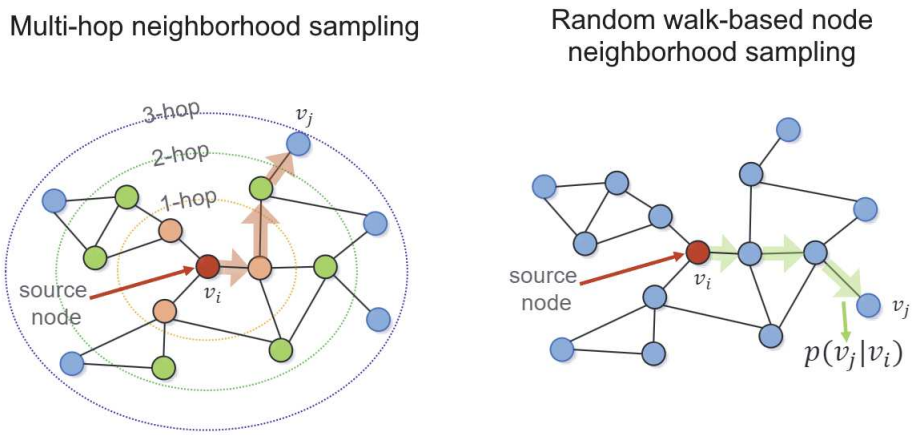


Figure 2.3: Node neighborhood sampling techniques [1]:
(A) Multi-hop neighborhood sampling: In this method nodes of various colors represent those sampled at different distances or "hops" from v_i. The node v_j is shown as one of the neighbors located three hops away from v_i, with the arrows indicating the shortest path from v_i to v_j.
(B) Random walk-based node neighborhood sampling: In this technique, the neighborhood of each node is sampled based on transition probabilities computed through a random walk process.

The nodes in the random walk are generated based on a distribution, where r_i represents the i-th node in the walk starting from $r_0 = u$, π_{vx} denotes the unnormalized transition probability between nodes v and x and Z is a normalizing constant.

We can define a random walk as a Markov chain over the set of nodes V. The transition probability of moving to node v depends solely on the previous node u and is determined by the adjacency matrix A. For a standard random walk, the transition probability is proportional to the edge weight A_{uv}:

$$p(x_{t+1} = v \mid x_t = u) = \frac{A_{uv}}{\deg(u)}$$

where $x_t \in V$ is the position of the walker at time t.

Transition probabilities between all pairs of nodes are represented by the transition probability matrix T.

Standard random walks naturally capture node neighborhoods in undirected connected graphs. One can also design biased random walks to explore different neighborhood notions.

Random-walk-based similarity functions rely on the co-visiting probabilities of a walker. An important property of random walks, often overlooked in the embedding literature, is their capacity to capture similarity at different structural scales (e.g., local vs. global).

node2vec

`DeepWalk` and `node2vec` are graph embedding techniques that, similar to matrix factorization approaches, rely on shallow embeddings and use a decoder based on the inner product. However, instead of decoding a deterministic node similarity measure, they optimize embeddings to encode the statistics of random walks on the graph. The core idea is to learn embeddings such that:

$$\mathrm{DEC}(z_i, z_j) = \frac{e^{z_i^\top z_j}}{\sum_{v_k \in V} e^{z_i^\top z_k}} \approx p_{G,T}(v_j \mid v_i)$$

where $p_{G,T}(v_j \mid v_i)$ is the probability of visiting node v_j during a random walk of length L starting from node v_i. Typically, L is chosen from the range $L \in \{2, \ldots, 10\}$. This probability $p_{G,L}(v_j \mid v_i)$ is stochastic and asymmetric, unlike the similarity measures in traditional approaches.

These methods aim to minimize the following cross-entropy loss:

$$\mathcal{L} = \sum_{(v_i, v_j) \in V} -\log(\mathrm{DEC}(z_i, z_j))$$

where the training set V is generated by sampling random walks starting from each node, meaning N pairs (v_i, v_j) are sampled from the distribution $(v_i, v_j) \sim p_{G,L}(v_j \mid v_i)$. However, directly evaluating this loss is computationally expensive, particularly $O(|V|^2)$, because the denominator in the normalization factor $\sum_{v_k \in V} e^{z_i^\top z_k}$ has a time complexity of $O(|V|)$.

To address this, `node2vec` use different optimizations and approximations to compute this loss. While `DeepWalk` employs a hierarchical softmax technique, leveraging a binary-tree structure to speed up the computation, `node2vec` approximates the loss using negative sampling, where instead of normalizing over the entire set of vertices, it uses a random set of "negative samples."

Beyond these algorithmic differences, the key distinction between `node2vec` and `DeepWalk` is the flexibility of the random walks. While `DeepWalk` uses simple unbiased random walks, `node2vec` introduces two hyperparameters, p and q, that bias the random walk. The hy-

perparameter p controls the likelihood of the walk immediately revisiting a node, while q influences the likelihood of revisiting a node's one-hop neighborhood. By adjusting these parameters, `node2vec` can smoothly transition between walks that are more similar to breadth-first search (BFS) or depth-first search (DFS). This flexibility allows `node2vec` to learn embeddings that either emphasize community structures (BFS-like behavior) or local structural roles (DFS-like behavior).

`node2vec` modifies the random walk process to account for different types of network neighborhoods. The probability of transitioning from node v to node u after having just visited node t is given by:

$$P(c_i = u \mid c_{i-1} = v) = \frac{\pi_{vu}}{Z}$$

if $(v, u) \in E$, and 0 otherwise. Here, π_{vu} is the unnormalized transition probability between nodes v and u, and Z is the normalizing constant. The variable π is defined as follows:

$$\pi_{vu} = \begin{cases} \frac{1}{p}\omega_{vu} & \text{if } d_{tu} = 0 \\ \omega_{vu} & \text{if } d_{tu} = 1 \\ \frac{1}{q}\omega_{vu} & \text{if } d_{tu} = 2 \end{cases}$$

where ω_{vu} is the weight of the edge between nodes v and u, and d_{tu} is the shortest path distance between node u and node t (the node visited before v). The parameters p and q control the random walk's behavior: p influences the likelihood of revisiting the previous node (controlling the walk's tendency towards depth-first search), and q affects the likelihood of exploring nearby nodes (steering the walk towards breadth-first search).

2.2.3 Optimization-Based Methods

Matrix factorization and random walk methods for graph embedding utilize distinct mathematical approaches. Matrix factorization, being a well-established mathematical technique, differs fundamentally from random walk-based methods, which, while diverse, are unified by the principle of exploring graph structure through stochastic processes.

Beyond these two categories, there are other embedding techniques that do not fit neatly into either group but aim to achieve similar goals through optimization. These methods, often termed hybrid approaches, rely on a shared step of optimization, typically carried out via gradient descent. The core of these optimization-based methods is the loss function, which encapsulates the desired properties to be preserved in the embeddings. This loss function integrates node similarities from the original space with regularization terms to maintain specific network features.

In essence, while matrix factorization and random walks offer different mathematical

frameworks, many modern embedding methods leverage optimization techniques to refine node representations. By focusing on the optimization of a well-defined loss function, these approaches align closely in their goal of preserving relevant network characteristics, even as they employ varied mathematical processes.

HOPE

High-Order Proximity preserved Embedding [7] (`HOPE`) is an algorithm designed to efficiently capture high-order similarities in large-scale networks, particularly for directed graphs where asymmetric transitivity is crucial. Asymmetric transitivity refers to the idea that the relationship from node v_i to node v_j may differ from the relationship from v_j to v_i. `HOPE` addresses this by approximating high-order proximity through a loss function that minimizes the difference between the observed similarity and the predicted similarity based on the embedding, specifically using the L_2-norm to measure the reconstruction error:

$$\mathcal{L} = \sum_{v_i, v_j \in V} \|\text{Dec}(z_i, z_j) - s_G(v_i, v_j)\|_2^2,$$

where $\text{Dec}(z_i, z_j) = z_i^T z_j$ and $s_G(v_i, v_j)$ denotes any similarity measure between v_i and v_j.

The authors of `HOPE` propose a general factorization approach for various similarity measures, expressing the similarity matrix S as a product of two matrices, M_g^{-1} and M_l. In this formulation, M_g^{-1} represents the inverse of the global similarity matrix, capturing broad structural relationships across the network, while M_l is the local similarity matrix, focusing on more localized connections within the network. Both matrices are typically polynomial functions of the adjacency matrix or its variants, which allows for efficient computation using Generalized Singular Value Decomposition (SVD).

Proximity Measurement	$\mathbf{M_g}$	$\mathbf{M_l}$
Katz	$I - \beta \cdot A$	$\beta \cdot A$
Personalized PageRank	$I - \alpha T$	$(1 - \alpha) \cdot I$
Common Neighbors	I	A^2
Adamic-Adar	I	$A \cdot D \cdot A$

Table 2.2: Proximity Measurements and Corresponding M_g and M_l Matrices

`HOPE` can be used with several common similarity measures, including the Katz Index, Rooted PageRank, Common Neighbors, and Adamic-Adar score. These measures can all be expressed in the general formulation $S = M_g^{-1} M_l$ which are shown in Table 2.2. For example:

- **Katz Index** considers all paths between two nodes, with a decay parameter β to control the weight of longer paths.

- **Rooted PageRank** measures the probability that a random walk from node v_i ends at node v_j in a steady state, with a parameter α controlling the random walk behavior.
- **Common Neighbors** counts the number of shared neighbors between two nodes
- **Adamic-Adar** modifies the common neighbors approach by weighting neighbors inversely by their degree

`HOPE`'s formulation unifies these diverse similarity measures under a single framework, enabling the algorithm to efficiently embed large-scale graphs while preserving both local and global asymmetric transitivity.

2.3 Deep Learning Embeddings

Representation learning has always been a key challenge in machine learning, with many studies focused on finding effective methods to learn meaningful features of data samples. Recently, deep neural networks have demonstrated their strong ability to create valuable embeddings across various types of data. [27, 33]

While deep learning has achieved great success in various fields, there hasn't been as much focus on using these techniques for network data, especially in learning network representations. However, the growing interest in deep learning has led to an increase in methods that apply deep neural networks to graph data. Deep autoencoders, which are known for their ability to model complex, non-linear structures, have been used for tasks like dimensionality reduction and network embedding in approaches such as SDNE (Structural Deep Network Embedding) and DNGR (Deep Neural Networks for Learning Graph Representations). These methods take advantage of the deep autoencoders' ability to model non-linearities, allowing them to create embeddings that capture the intricate structure of graphs.

The success of deep learning in data analysis, including network embedding, can be attributed to its ability to capture complex features and non-linear relationships among input variables.
Graph Neural Networks (GNNs) extend the concept of Convolutional Neural Networks to graph data, enabling the encoding of high-dimensional information from a node's neighborhood into a dense vector embedding. GNN methods typically consist of two components: an encoder, which maps a node v_i to a low-dimensional embedding vector z_i based on its local neighborhood and attributes, and a decoder, which extracts user-specified predictions from the embedding vector. This approach is well-suited for end-to-end learning and provides state-of-the-art performance. GNN-based methods include:

- **Graph Convolutional Networks (GCNs)**: GCNs extend the principles of Convolutional Neural Networks (CNNs) to graph-structured data. They apply convo-

lutional operations directly to the nodes and edges of a graph, allowing them to capture local properties and relationships within the graph, making them effective for tasks like node classification and link prediction.

- **Graph Attention Networks (GATs)**: GATs enhance the capabilities of Graph Convolutional Networks by introducing an attention mechanism. This mechanism allows the model to weigh the importance of different neighboring nodes, rather than treating them all equally. By using masked self-attentional layers, GATs can focus more on relevant neighbors, improving the model's ability to capture complex relationships in the graph.

To define a Graph Neural Network (GNN) model, we first need to understand the graph data structure and its inherent limitations. The contextual information of a graph dataset depends on both the features of the nodes and the structural properties of the graph itself. Ignoring the relationships between nodes and treating them as independent entities can lead to a loss of structural information and result in a model that is less expressive and interpretable. Thus, it is crucial to incorporate the graph's structure into the model. We start by examining the concept of permutation invariance and its significance for GNN models. Next, we describe the message-passing mechanism used in GNNs, which integrates both node features and structural information through local feature extraction.

2.3.1 Permutation Invariance

A GNN model must be either permutation invariant or permutation equivariant to be effective. This means that the function f should not be influenced by the arbitrary ordering of nodes in the adjacency matrix.

Definition 2.3.1 (Permutation Invariance and Equivariance)**.** *A function f that processes an adjacency matrix A should satisfy one of the following properties:*

- $f(PAP^T) = f(A)$ *(Permutation Invariance)*
- $f(PAP^T) = Pf(A)$ *(Permutation Equivariance)*

where P is a permutation matrix.

In essence, it is not feasible to input the adjacency matrix of a graph into a neural network to produce a graph embedding, as this approach is dependent on the arbitrary ordering of nodes within the matrix.

2.3.2 Neural Message Passing Framework

The Neural Message Passing Framework is a general approach for processing graph-structured data using iterative computations. The core idea is that the embedding of

each node is updated in iterations by aggregating information from its neighbors. The computation proceeds in iterations, where the embedding of a node is updated based on the embeddings of its neighboring nodes. At iteration $k = 0$, each node v has an initial embedding $h_v^{(0)} = x_v$, where x_v represents the node's features.

- **AGGREGATE:** The $\texttt{AGGREGATE}^{(k)}$ function collects the embeddings of the neighbors of node u at iteration k and computes a message:

$$m_{N(u)}^{(k)} = \texttt{AGGREGATE}^{(k)}\left(\{h_v^{(k)}, \forall v \in N(u)\}\right),$$

  where $N(u)$ is the set of neighbors of node u.

- **UPDATE:** The $\texttt{UPDATE}^{(k)}$ function then updates the embedding of node u using its current embedding and the aggregated message:

$$h_u^{(k+1)} = \texttt{UPDATE}^{(k)}\left(h_u^{(k)}, m_{N(u)}^{(k)}\right).$$

After K iterations, the final embedding of node u is denoted as $z_u = h_u^{(K)}$. In its most basic form, the AGGREGATE function sums up the embeddings of neighboring nodes:

$$m_{N(u)}^{(k)} = \sum_{v \in N(u)} h_v^{(k)}.$$

The UPDATE function then computes the new embedding of node u as follows:

$$h_u^{(k+1)} = \sigma\left(W_{\text{self}}^{(k+1)} h_u^{(k)} + W_{\text{neigh}}^{(k+1)} m_{N(u)}^{(k)} + b^{(k+1)}\right),$$

where $W_{\text{self}}^{(k+1)}$ and $W_{\text{neigh}}^{(k+1)}$ are trainable weight matrices, $b^{(k+1)}$ is the bias term and $\sigma(\cdot)$ is an element-wise non-linear activation function.

Graph Convolutional Networks (GCNs)

Graph Convolutional Networks normalize the AGGREGATE function by the degrees of the nodes, taking into account self-loops:

$$h_u^{(k+1)} = \sigma\left(\sum_{v \in N(u) \cup \{u\}} \frac{1}{\sqrt{d_u d_v}} h_v^{(k)} W^{(k+1)} + b^{(k+1)}\right),$$

where d_u and d_v are the degrees of nodes u and v, respectively and the summation includes both the neighbors and the node itself (self-loop).
This formulation ensures that the message-passing process in GCNs is normalized by node degrees, making the model less sensitive to the degree distribution of the graph.

2.3.3 Limitations of Graph Neural Networks

Graph Neural Networks (GNNs) have revolutionized the way we approach learning from graph-structured data by leveraging node and edge information to produce meaningful embeddings. However GNNs still face several limitations, particularly when dealing with graph isomorphism and expressive power.

Definition 2.3.2 (Graph Isomorphism)**.** *Two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are isomorphic if and only if there exists a bijection $f : V_G \to V_H$ such that for every pair of vertices $u, v \in V_G$, the edge $(u, v) \in E_G$ if and only if $(f(u), f(v)) \in E_H$.*

In other words, G and H are isomorphic if there exists a one-to-one correspondence between their vertex sets that preserves adjacency.
The graph isomorphism problem seeks to determine whether two graphs are topologically identical. Despite significant research, no polynomial-time algorithm is known to solve this problem efficiently. The Weisfeiler-Lehman (WL) test, introduced by Weisfeiler and Lehman [21], offers a practical approach to this challenge by providing an effective and computationally efficient method for distinguishing a broad class of graphs.

The WL test works by iteratively refining node labels based on the labels of their neighboring nodes. The test involves the following steps:

1. **Aggregation**: Collect and aggregate the labels of a node and its neighbors.
2. **Hashing**: Generate new labels by hashing the aggregated information.

Two graphs are deemed non-isomorphic if their node labels differ at any iteration of the test. The Weisfeiler-Lehman test is based on the following theorem:

Theorem 1 (Weisfeiler-Lehman Test Theorem)**.** *Let G and H be two graphs. If the WL test terminates with the labels of corresponding nodes in G and H being different, then G and H are non-isomorphic. Conversely, if G and H are non-isomorphic, then there exists an iteration at which the labels of corresponding nodes in G and H are different.*

GNNs have demonstrated substantial utility in various applications by iteratively updating node embeddings to capture network structures and features of surrounding nodes. However, they face inherent limitations in their expressiveness and capability to distinguish graph structures. GNNs are inspired by the WL test's approach to graph labeling and aggregation. While GNNs effectively learn from node neighborhoods, their aggregation schemes face limitations:

- **Aggregation Scheme**: Many GNNs use aggregation functions that may not be injective. This means different node neighborhoods can be mapped to the same embedding, reducing the network's ability to capture distinct subtree structures.
- **Subtree Structures**: The expressiveness of a GNN is constrained by its ability to distinguish different subtree structures. If a GNN's aggregation scheme is not

injective, it might fail to differentiate between graphs that have distinct subtree structures but produce similar embeddings.

To analyze a GNN's representational power, we consider the concept of a multiset:

Definition 2.3.3 (Multiset)**.** *A multiset is a generalized concept of a set that allows multiple instances for its elements. More formally, a multiset is a 2-tuple $X = (S, m)$, where S is the underlying set of X formed from its distinct elements, and $m : S \to \mathbb{N}_{\geq 1}$ gives the multiplicity of the elements.*

The goal is to determine whether a GNN's aggregation scheme maps different multisets to distinct embeddings. A maximally powerful GNN would map different multisets to unique representations, reflecting the true diversity of graph structures.

The theoretical framework for analyzing GNNs involves:

- **Injectivity**: A GNNs aggregation function should be injective to ensure that distinct neighborhoods map to unique embeddings.
- **Expressive Power**: To achieve maximal expressiveness, a GNN must represent injective multiset functions effectively. Many popular GNNs exhibit non-injective aggregation schemes, limiting their ability to capture all graph properties.

Despite their advancements, GNNs are inherently limited by their aggregation mechanisms and their ability to distinguish complex graph structures. Understanding these limitations through the WL test and related theoretical frameworks helps in designing more powerful GNNs that better capture and differentiate the nuances of graph data.

2.3.4 Advanced Graph Neural Network Models

This section delves into two advanced graph neural network frameworks: Variational Graph Auto-Encoders (`VGAE`) and Graph Isomorphism Networks (`GIN`). `VGAE` builds on variational autoencoders to capture latent structures in graph data, enhancing link prediction through probabilistic modeling. `GIN`, on the other hand, extends the Weisfeiler-Lehman test to achieve maximal discriminative power, leveraging deep multisets for effective node representation.

VGAE

Variational Graph Auto-Encoders (`VGAE`) is a framework for unsupervised learning on graph-structured data that builds on the variational autoencoder (`VAE`) approach. `VGAE` leverages latent variables to learn interpretable representations for undirected graphs. Specifically, the model uses a graph convolutional network (GCN) as the encoder and a simple inner product as the decoder, enabling it to perform well on tasks like link prediction in citation networks. Unlike many existing unsupervised learning models for graph data and link prediction, `VGAE` can naturally integrate node features, leading to

improved predictive performance across various benchmark datasets.
The inference model is parameterized by a two-layer GCN, which estimates the mean and variance of the latent variables based on the node features and the graph structure. The GCN is defined as

$$h_u^{(k+1)} = \text{ReLU}\left(\tilde{A} \sum_{v \in N(u) \cup \{u\}} \frac{1}{\sqrt{d_u d_v}} h_v^{(k)} W^{(0)}\right) W^{(1)}$$

where:

- $h_v^{(k)}$ is the feature representation of node v at the k-th layer.
- $W^{(0)}$ and $W^{(1)}$ are the learnable weight matrices.
- ReLU is the activation function.
- $\tilde{A} = D^{-1/2} A D^{-1/2}$ is the normalized adjacency matrix.

The generative model predicts the graph structure by computing the inner product between pairs of latent variables. Learning is achieved by optimizing the variational lower bound, involving the reconstruction of the adjacency matrix and the Kullback-Leibler divergence between the approximate and prior distributions of the latent variables.

Kipf et al. demonstrated that using variational autoencoders for graph embedding can significantly enhance performance compared to non-probabilistic autoencoders, as `VGAE` effectively captures higher-order dependencies between nodes in the graph.

GIN

The Graph Isomorphism Network (`GIN`) extends the Weisfeiler-Lehman (WL) test, achieving maximal discriminative power among Graph Neural Networks (GNNs).

To model injective multiset functions for neighbor aggregation, it introduce the concept of "deep multisets", which involves parameterizing universal multiset functions using neural networks. The parameter ϵ can be learnable or fixed. `GIN` then updates node representations as follows:

$$h_v^{(k)} = \text{MLP}^{(k)}\left(\left(1 + \epsilon^{(k)}\right) \cdot h_v^{(k-1)} + \sum_{u \in N(v)} h_u^{(k-1)}\right)$$

While other powerful GNNs may exist, `GIN` is a maximally powerful and simple example. The node embeddings learned by `GIN` can be applied to tasks like node classification and link prediction. As the number of iterations increases, node representations, which correspond to subtree structures, become more global. Sufficient iterations are key to achieving strong discriminative power, though features from earlier iterations may sometimes generalize better. To capture all structural information, we utilize data from all depths/iterations of the model. This is similar to *Jumping Knowledge Networks*, where graph representations are concatenated across all iterations/layers of `GIN`.

Chapter 3

Short Walks Node Embedder (SNOW)

In response to the increasing evidence that custom embeddings can outperform standard methods for tasks such as link prediction and node classification, we developed a custom random walk embedder called *Short Walks NOde embedder* (`SNOW`) specifically designed to predict node heaviness by exploiting information from short random walks.

This method aims to derive node embeddings that capture the likelihood of a given node being part of a triangle. For an edge $\{u, v\}$, we focus on reflecting the frequency with which random walks starting from node u reach node v and vice versa, excluding the direct edge $\{u, v\}$ itself. In the case of a heavy edge, where the edge participates in multiple triangles, random walks from u are more likely to reach v through a shared neighbor, and vice versa. This behavior is effectively captured by the random walks. By incorporating this information into the embeddings of the nodes connected by the edge, the resulting node embeddings provide a detailed representation of the local triangle structure.

3.1 Implementation Details

We implemented this approach using the steps described in the following sections.

3.1.1 Embedding matrix initialization

If our dataset contains at least d node features (where d represents the desired dimensionality of the final embedding), we apply Principal Component Analysis (PCA) on them to obtain a starting vector for each node. The resulting vectors will have a dimensionality equal to the embedding dimension, d.
In cases where the number of features is insufficient (i.e., num_features $< d$) or when no features are present, we initialize each node with a random vector in $\mathbb{R}^d$. These vectors

are drawn from a uniform distribution with values in the range $[0, 1)$, which is the default range used by the Python random function for generating random floats.

3.1.2 Negative Sampling

To improve the performance of SNOW, we apply negative sampling to the nodes in order to refine the embeddings. We perform negative sampling before applying the positive ones, as this helps to "adjust" the initial embedding vectors, which might otherwise have vectors too skewed or exaggerated. By first smoothing out these irregularities, the embeddings are better prepared for the positive sampling phase. Empirically, this approach results in a slight improvement in overall accuracy.

For each node u, as shown in Algorithm 1, we select a fixed number (if possible) of 2-paths $(u - v - z)$ that do not form triangles. To achieve this, we select z from the set $N_{u \oplus v} \leftarrow (N_u \oplus N_v)$, which consists of nodes that are neighbors of either u or v, but not both. Here, $N_u \oplus N_v$ represents the symmetric difference between the neighbors of u and v, ensuring that the selected node z does not connect to both u and v. Then, in Algorithm 3, we update the embedding of the nodes using a fixed learning rate η_n:

$$z_u = z_u - \sigma(z_w \cdot z_u)\eta_n$$

$$z_v = z_v - \sigma(z_v \cdot z_w)\eta_n$$

This process helps in reducing the similarity between embeddings of nodes that are not forming a triangle. Empirical results show that incorporating negative sampling into the SNOW algorithm can improve the accuracy, making the embeddings more accurate and effective for predicting node heaviness.

3.1.3 Random Walks

For each edge (u, v), as shown in Algorithm 2 and 3, we will perform multiple random walks starting from both u and v. These random walks are of length 3 (because a cycle of length 3 is necessary to form a triangle) allowing for the exploration of the local neighborhood structure while maintaining focus on proximity relationships. On each walk, the path is traced to check if it finishes at the other end of the edge: v for walks starting from u, and u for walks starting from v.

Heaviness Probability P_{uv}

The probability that a random walk starting from node u reaches node v (and vice versa) is computed by performing the walk multiple times and counting the number of successful paths that connect u to v through a neighbor node (see Figure 3.1). These probabilities, $p_{u,v}$ and $p_{v,u}$, are normalized by the number of walks w and are then used to update the

Algorithm 1 GenerateNegativeSamples(G, n)

Input: Graph $G = (V, E)$, number n of negative samples for each node
Output: R set of negative samples
$R \leftarrow \{\}$
for $u \in V$ **do**
  $R_u \leftarrow \{\}$
  $N_u \leftarrow$ list of neighbors of u
  if $N_u \neq \emptyset$ **then**
    while $|R_u| < n$ **do**
      $v \leftarrow$ uniformly randomly selected node from N_u
      $N_v \leftarrow$ list of neighbors of v
      $N_{u \oplus v} \leftarrow (N_u \oplus N_v)$
      if $N_{u \oplus v} \neq \emptyset$ **then**
        $w \leftarrow$ uniformly randomly selected node from $N_{u \oplus v}$
        $R_u \leftarrow R_u \bigcup \{(u, v, w)\}$
      end if
    end while
    $R \leftarrow R \bigcup R_u$
  end if
end for
return R

Algorithm 2 PerformWalk(G, u, v, l)

Input: Graph G, nodes u, v, length of walk l
Output: 1 if the path from u goes through v, 0 otherwise.
$M \leftarrow \{u\}$
$N_u \leftarrow$ list of neighbors of u
while $|M| < l$ **do**
  if $N_u \neq \emptyset$ **then**
    $w \leftarrow$ uniformly randomly selected node from N_u
    $N_u \leftarrow$ list of neighbors of w
    $M \leftarrow M \cup \{w\}$
  else
    return 0
  end if
end while
if $M[-1] = v$ **then**
  return 1
end if
return 0

nodes embedding accordingly.

$$P_{uv} = \frac{p_{v,u} + p_{u,v}}{w}$$

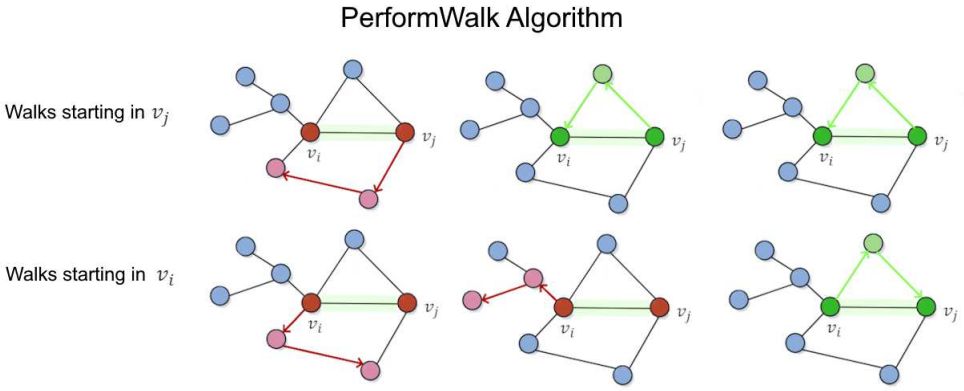


Figure 3.1: Visualization of the PerformWalk algorithm: in this figure, we illustrate the PerformWalk algorithm. Green edges denote successful runs, while red edges indicate failed attempts. The first segment represents the walks from vertex v_i to vertex v_j, and the second segment captures the reverse walks from v_j to v_i.

Node Embedding Vector Update

We multiply the dot product between the embeddings of nodes u and v by the heaviness probability P_{uv} and a fixed learning rate η_p to update the embedding vector of node u. The same update process is applied to the embedding vector of node v:

$$z_u = z_u + \eta_p P_{uv} \sigma(z_u \cdot z_v)$$

$$z_v = z_v + \eta_p P_{uv} \sigma(z_u \cdot z_v)$$

3.1.4 Loss Function

The objective function of the SNOW algorithm combines both negative sampling loss and positive sampling loss to optimize node embeddings. Below is the formulation of the objective function:

The negative sampling loss aims to minimize the similarity between nodes that don't form triangles in the graph. For each triplet of negative samples (u, v, w), the negative sampling loss is defined as:

$$\mathcal{L}_{\text{neg}} = \sum_{(u,v,w) \in R} \frac{\sigma(z_w \cdot z_u) + \sigma(z_v \cdot z_w)}{2}$$

Here, R is the set of negative samples, and $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the sigmoid function.

Algorithm 3 SNOW$(G, n, d, w, l, \eta_p, \eta_n)$

Input: Graph $G = (V, E)$,
number of negative samples for each node n,
embedding dimension d,
number of walks w,
lenght of walks l,
lerning rate for positive and negative samples η_p, η_n,
Output: Z list of node embeddings
$Z \leftarrow$ InitializeNodeEmbeddings(G, d) such that $Z[i] \in \mathbb{R}^d$ for each $i \in V$
$R \leftarrow$ GenerateNegativeSamples(G, n)

% Negative Samples

for $(u, v, w) \in R$ **do**
    $z_u \leftarrow Z[u]$
    $z_v \leftarrow Z[v]$
    $z_w \leftarrow Z[w]$
    $Z[u] \leftarrow z_u - \sigma(z_w \cdot z_u)\eta_n$
    $Z[v] \leftarrow z_v - \sigma(z_v \cdot z_w)\eta_n$
end for

% Positive Samples

for $\{u, v\} \in E, u \neq v$ **do**
    $E' \leftarrow E \setminus \{\{u, v\}\}$
    $p_{u,v} \leftarrow 0$
    $p_{v,u} \leftarrow 0$
    for w times **do**
        $p_{u,v} \leftarrow p_{u,v} +$ PerformWalk$(G' = (V, E'), u, v, l)$
        $p_{v,u} \leftarrow p_{v,u} +$ PerformWalk$(G' = (V, E'), v, u, l)$
    end for
    $P_{uv} \leftarrow p_{v,u}/w + p_{u,v}/w$
    $z_u \leftarrow Z[u]$
    $z_v \leftarrow Z[v]$
    $Z[u] \leftarrow z_u + \eta_p P_{uv} \sigma(z_u \cdot z_v)$
    $Z[u] \leftarrow z_u + \eta_p P_{uv} \sigma(z_v \cdot z_u)$
end for
return Z

On the other hand, the positive sampling loss encourages the embeddings of highly triangle-connected nodes to be similar. For each edge $\{u, v\} \in E$, the positive sampling loss is defined as:

$$\mathcal{L}_{\text{pos}} = -\sum_{\{u,v\}\in E} \frac{p_{v,u} + p_{u,v}}{w} \sigma(z_u \cdot z_v)$$

where $p_{u,v}$ and $p_{v,u}$ represent the successful random walks between nodes u and v, and w is the number of walks.

The final loss function, which the `SNOW` algorithm aims to minimize, thus combines both the negative and positive sampling losses:

$$\mathcal{L} = \mathcal{L}_{\text{neg}} + \mathcal{L}_{\text{pos}}$$

Substituting the expressions for $\mathcal{L}_{\text{neg}}$ and $\mathcal{L}_{\text{pos}}$, we get the following complete loss function:

$$\mathcal{L} = \sum_{(u,v,w)\in R} \frac{\sigma(z_w \cdot z_u) + \sigma(z_v \cdot z_w)}{2} - \sum_{\{u,v\}\in E} \frac{p_{v,u} + p_{u,v}}{w} \sigma(z_u \cdot z_v)$$

Therefore, the objective of the `SNOW` algorithm is to minimize the combined loss $\mathcal{L}$, optimizing the embeddings Z for the graph.

3.1.5 Complexity

We now analyze the temporal complexity of the `SNOW` algorithm.

- The initialization process using Principal Component Analysis (PCA) requires $O(d^2|V| + d^3)$, where d represents the dimensionality of the embeddings and $|V|$ denotes the number of nodes in the graph.
- Generating negative samples incurs a complexity of $O(|V|n)$, which arises from iterating over each node and sampling it n times.
- Applying these negative samples also requires $O(|V|n)$, reflecting the total number of negative samples to be processed.
- Performing the random walks incurs a complexity of $O(l)$, where l denotes the length of each walk.
- Finally, the application of positive samples necessitates $O(2|E|wl)$, where $|E|$ is the number of edges, w is the number of walks per edge, and l is the length of the walks. This complexity arises from iterating over each edge, the number of walks per edge, and applying the PerformWalk function twice.

At the end of the analysis, the final complexity of the `SNOW` algorithm is given by $O(d^2|V| + d^3 + 2|V|n + 2|E|wl)$. Since the length l of the walks is very short, this factor can often be

ignored. Additionally, because the number w of walks, the embedding dimension d, and the number n of negative samples per node are generally small compared to other terms, the overall time complexity is essentially $O(|V| + |E|)$.

Chapter 4

Experimental Methodology

The prediction of node heaviness is a crucial problem in network analysis, with significant implications for the study of social networks, biological networks, and other complex systems. In this context, we define the heaviness of a node as the number of triangles adjacent to it, as triangles represent the simplest and most fundamental motif in graph theory. Understanding heavy nodes within a graph provides valuable insights into the structure and connectivity of the network. Therefore, accurately predicting node heaviness requires embedding methods capable of capturing both local and global graph structures.

Together with `SNOW` we evaluate several state-of-the-art advanced graph embedding methods to determine their suitability for predicting node heaviness in unweighted, undirected graphs. The methods considered include the Graph Isomorphism Network (`GIN`), Variational Graph Auto-Encoder (`VGAE`), High-Order Proximity preserved Embedding (`HOPE`), `node2vec`, Laplacian Eigenmaps (`LE`), and `GraRep`. Each method is here discussed in terms of its ability to capture the graph properties essential for accurately predicting triangle counts, and by extension, node heaviness.

4.1 State of the art Embedding Methods and Their Applicability

For the task of predicting triangle counts in unweighted, undirected graphs, deep learning-based methods like `GIN` prove to be particularly effective due to their capability to capture complex graph structures. `GIN`, in particular, stands out for its expressive power and versatility, making it highly suitable for this task. `node2vec` remains a strong candidate in scenarios where neighborhood information is crucial, although it may require tuning for optimal performance in triangle prediction while `GraRep` emerges as an attractive option due to its multi-scale representation capabilities, making it well-suited for capturing the necessary structural information for triangle count prediction.

Laplacian Eigenmaps

`LE` is an effective dimensionality reduction technique that utilizes the Laplacian eigenvectors of the graph to map nodes into a lower-dimensional space. By prioritizing the preservation of local neighborhood information through minimizing the Laplacian quadratic form, this approach excels at capturing the graph's fundamental structure. Its focus on local relationships allows it to deliver a meaningful representation of the graph's inherent geometry. `LE` offers a strong basis for uncovering and analyzing the underlying patterns in large, unweighted, undirected graphs.

GraRep

`GraRep` is a method that extends matrix factorization techniques to capture global graph structures at multiple scales. By using higher-order proximity information, `GraRep` generates embeddings that encode various levels of graph structure. It's ability to capture higher-order proximities makes it well-suited for identifying structural motifs across different scales of the graph.

node2vec

`node2vec` is a random walk-based embedding method that generates node representations by simulating random walks over the graph. By adjusting the bias parameters, `node2vec` can explore different neighborhood structures, capturing a balance between local and global graph properties. `node2vec` is relatively efficient and scalable, making it suitable for large graphs, although it may not capture triangle structures as comprehensively as deep learning methods.

High-Order Proximity preserved Embedding (HOPE)

High-Order Proximity preserved Embedding is an optimization-based method designed to preserve higher-order proximities between nodes in the embedding space. By focusing on preserving asymmetric transitivity, `HOPE` can capture more complex relationships between nodes that are not directly connected. Although triangles are inherently symmetric, the ability to capture high-order proximities and asymmetric relationships can help understand the broader structural context in which triangles form. `HOPE` captures the structural roles of nodes within the graph, indirectly supporting the identification of triangle-rich regions.

Variational Graph Auto-Encoder (VGAE)

The Variational Graph Auto-Encoder is a deep learning-based approach that excels at learning graph embeddings through a probabilistic framework. `VGAE` leverages both the graph's structure and node features to learn embeddings that reflect the overall topology of the graph. The embeddings learned by `VGAE` are highly expressive, enabling the model

to capture complex structural information critical for accurate triangle count prediction and its probabilistic nature allows it to adapt effectively to various graph types without reliance on additional node labels.

Graph Isomorphism Network (GIN)

The Graph Isomorphism Network is another deep learning-based method designed to enhance the expressive power of graph neural networks. `GIN`'s ability to capture intricate node relationships and substructures makes it especially suitable for tasks requiring an understanding of triangles in the graph. Also it does not require labeled nodes, making it ideal for scenarios involving unweighted, undirected graphs without additional node attributes.

4.2 Direct Triangle Counting Methods

In this section, we turn our focus to methods specifically designed for directly predicting or counting triangles in graph structures. These methods will be used as a basis for evaluating the efficiency of embedding-based approaches used for predicting node heaviness, in terms of computational time and accuracy.

To achieve a thorough evaluation, we will compare the performance of these direct triangle counting techniques with that of embedding methods we presented in the previous chapter. Our objective is to evaluate not only the accuracy of each approach but also their efficiency regarding time complexity and scalability.

By examining both direct counting methods and embeddings methods in our tests, we aim to uncover the strengths and weaknesses of each approach. The subsequent sections will explore specific methods for triangle counting and prediction, including a simple linear degree predictor algorithm and NetworkX's triangle counting function.

4.2.1 Simple Degree Predictor (SDP)

We propose an extremely simple linear model to predict node heaviness. The predictive model is expressed in the form of a linear equation:

$$\tilde{N}_{\mathcal{M}}(u) = \alpha \cdot \deg(u) + \beta$$

where $\deg(u)$ represents the degree of node u, and α and β are parameters to be determined. This model is motivated by the observation that nodes with higher degrees are generally more likely to be involved in a larger number of triangles, as a higher degree increases the probability of forming interconnected neighborhoods.
The degree of a node often serves as a strong indicator of its local clustering, implying that nodes with more connections tend to participate in more triangular structures.

To estimate α and β, we apply linear regression to 40% of the nodes. This training set allows us to find the optimal values for these parameters. Subsequently, we use the remaining 60% of the data, which serves as the test set, to evaluate the performance of our predictive model. By leveraging this approach, we aim to provide a simple yet effective method for estimating node heaviness, with α and β capturing the linear influence of the node's degree on its likelihood of participating in triangles.

4.2.2 NetworkX Triangle Counting Function

NetworkX provides a straightforward and efficient method for counting triangles in a graph through its 'triangles' function [36], as shown in Algoritmh 4. This function computes the number of triangles that each node participates in. Specifically, it calculates the number of triangles for each node u in the graph by checking the nodes adjacent to u and counting all sets of three interconnected nodes that include u.

Algorithm 4 nx.triangles(G)

Input: Graph $G = (V, E)$
Output: Number of triangles for each node or in the graph.
Initialize $R \leftarrow \{\}$
for each node $u \in V(G)$ **do**
  $R[u] \leftarrow \{v \mid v \in N(u) \text{ and } v \notin R \text{ and } v \neq u\}$
end for
Initialize $triangle\_counts \leftarrow$ Counter(dict.fromkeys($V(G), 0$))
for each $u \in V(G)$ **do**
  for each neighbor $v \in R[u]$ **do**
    $triangle\_counts[u] \leftarrow triangle\_counts[u] + |R[u] \cap R[v]|$
    $triangle\_counts[v] \leftarrow triangle\_counts[v] + |R[u] \cap R[v]|$
    $triangle\_counts.update(|R[u] \cap R[v]|)$
  end for
end for
return dict($triangle\_counts$)

4.3 Methodological Details

Our objective is to evaluate the performance of different graph embedding methods when used as input features for a neural network model designed to predict the number of triangles that a given node participates in within a graph G. In this section, we will provide a comprehensive overview of the experimental setup, detailing the process of assessing the effectiveness of various embedding techniques.

We use node embeddings as input features for our model. These embeddings, represented as a tensor z, are computed using the entire graph beforehand and are used as input to a feedforward neural network. The objective is to train the model in a fully-supervised setting using the Mean Absolute Error (MAE) loss function, with the target values being the actual number of triangles associated with each node in the graph. To balance the fact that in most datasets many nodes have very few triangles adjacent, we use a large train set of 40% of the nodes; for each of the nodes in the training set, in addition to its embedding we have its exact heaviness (number of triangles the node is involved in).

For each test configuration, we conduct 5 runs and aggregate the results by computing the mean.

4.3.1 Model Architecture

The model employed is a simple feedforward neural network composed of multiple linear layers interspersed with ReLU activation functions and dropout for regularization. The architecture is summarized as follows:

- **Input Layer:** The input consists of precomputed node embeddings.
- **Hidden Layers:** Five hidden layers with decreasing units (64, 32, 16, 8, and 4) to progressively capture complex interactions.
- **Output Layer:** A single unit that outputs a scalar value representing the predicted number of triangles.

We use dropout as a regularization technique to prevent overfitting. Dropout works by randomly "dropping out" a fraction of the neurons during each training iteration, which helps the model learn more robust features and prevents it from becoming overly reliant on specific neurons. This technique also enhances generalization, leading to better performance on unseen data. Additionally, dropout breaks co-adaptations, which occur when neurons adjust to compensate for each other's presence, resulting in a more independent and distributed learning process. The dropout layers are applied after the non-linear activation functions in the network, and the dropout rate is determined as a hyperparameter during the model selection procedure.

The network structure is defined as follows:

$$\begin{aligned}\mathbf{x} &\to \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \to \text{Dropout} \\ &\to \text{ReLU}(\mathbf{W}_2\mathbf{x} + \mathbf{b}_2) \to \text{Dropout} \\ &\to \text{ReLU}(\mathbf{W}_3\mathbf{x} + \mathbf{b}_3) \to \text{Dropout} \\ &\to \text{ReLU}(\mathbf{W}_4\mathbf{x} + \mathbf{b}_4) \to \text{Dropout} \\ &\to \text{ReLU}(\mathbf{W}_5\mathbf{x} + \mathbf{b}_5) \to \text{Output}\end{aligned}$$

where $\mathbf{W}_i$ and $\mathbf{b}_i$ represent the weights and biases of the i-th layer, respectively.

4.3.2 Loss Function and Optimization

The loss function used is Mean Absolute Error (MAE), defined as:

$$\text{MAE} = \frac{1}{|V|}\sum_{i=1}^{|V|}|y_i - \tilde{y}_i|$$

where y_i is the actual number of triangles, and $\tilde{y}_i$ is the predicted value for node i.

To optimize the model, we use the AdamW optimizer, which is a variant of the Adam optimizer with weight decay. This optimizer is chosen for its effectiveness in handling sparse gradients and noisy data. The learning rate is set to 0.002, which was found to be a good trade-off between convergence speed and stability.

4.3.3 Weight Decay

Weight decay is a regularization technique used to prevent overfitting by penalizing large weights. This is achieved by adding a penalty term to the loss function that discourages the network from assigning excessively large values to the model parameters. The weight decay term in the loss function is given by:

$$\text{Loss} = \text{MSE} + \lambda\sum_i \mathbf{W}_i$$

where λ is the weight decay coefficient, and $\mathbf{W}_i$ are the weights of the network.

In AdamW, weight decay is applied directly to the weights before the parameter update is performed, thus decoupling it from the gradient-based updates. The update rule for AdamW is:

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta\left(\frac{m_t}{\sqrt{v_t}+\epsilon} + \lambda\mathbf{W}_t\right)$$

where $\mathbf{W}_t$ are the weights at time step t, η is the learning rate, m_t is the first moment estimate, v_t is the second moment estimate, and ϵ is a small constant for numerical

stability. The weight decay term $\lambda \mathbf{W}_t$ is added directly to the gradient.

For this model, we use a weight decay value of $\lambda = 0.01$. This value was chosen based on empirical experiments to balance regularization and model performance effectively.

4.3.4 Weight Initialization

Proper initialization of weights is crucial for the efficient training of neural networks. In this model, we use Xavier initialization (also known as Glorot initialization), which sets the weights of the network based on the size of the previous layer. The method initializes the weights W with values drawn from a uniform distribution:

$$\mathbf{W} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

where n_{in} and n_{out} are the number of input and output units in the weight tensor, respectively. Xavier initialization ensures that the variance of the outputs remains consistent across layers, preventing issues such as vanishing or exploding gradients during the training process.

4.3.5 Hyperparameter Selection

Hyperparameters play a critical role in determining the performance of the model. Below (Table 4.1) is the table summarizing the hyperparameters used:

Hyperparameter	**Value**
Learning Rate	0.002
Batch Size	32
Number of Epochs	3000
Dropout Rate	0.03
Weight Initialization	Xavier
Weight Decay	0.01
Optimizer	AdamW
Loss Function	MAE

Table 4.1: Summary of selected Hyperparameters on the NN

These hyperparameters were selected after experimentation to ensure that the model achieves optimal performance on the given task. The small dropout rate and learning rate help in stabilizing the training process and preventing the model from overfitting to the training data.

4.4 Datasets

In this study, we employ six diverse datasets: CiteSeer, PPI, PubMed, WikiCS, Arxiv and Products which are commonly used in the field of graph-based machine learning. These datasets encompass a variety of domains, including academic citation networks, biological protein-protein interaction networks, and web-based content classification, offering a broad spectrum of challenges for testing the robustness and generalization capabilities of graph-based algorithms.

The following table (Table 4.2) summarizes the main characteristics of these datasets, highlighting the number of nodes, edges, and features associated with each dataset:

Dataset	**Nodes $\|V\|$**	**Edges $\|E\|$**	**Number of Features**
CiteSeer [22]	3,327	4,732	3,703
PPI [23]	56,944	818,716	50
PubMed [24]	19,717	44,338	500
WikiCS [25]	11,701	216,123	300
Arxiv [26]	169,343	1,166,243	128
Products [26]	2,449,029	61,859,140	100

Table 4.2: Main Characteristics of the datasets used

- The **CiteSeer** dataset is a citation network where each node represents a scientific document, and the edges represent citation links between these documents. It is often used to evaluate models for document classification and citation prediction.
- The **PPI (Protein-Protein Interaction)** dataset represents a biological network where nodes correspond to proteins, and edges denote interactions between them. This dataset is particularly useful for testing models on biological network tasks such as predicting protein functions.
- The **PubMed** is another citation network, specifically focusing on documents related to the topic of diabetes. Nodes in this dataset represent biomedical papers, while edges denote citation relationships between these papers. The dataset is enriched with features derived from the text of the documents.
- The **WikiCS** is derived from Wikipedia, where nodes represent articles, and edges signify the presence of hyperlinks between these articles. This dataset is often used for evaluating machine learning models on tasks like node classification and semi-supervised learning.

- The **Arxiv** dataset is a large-scale citation network drawn from the Arxiv repository. In this dataset, nodes correspond to scientific papers, and edges represent citations between them. Due to its large scale, Arxiv is an excellent dataset for testing the scalability of graph-based models.

- The **Products** dataset represents a large-scale product co-purchase network. In this dataset, nodes represent products, and edges indicate co-purchase relationships between these products. It is used for evaluating graph-based models on tasks such as node classification and link prediction in large e-commerce networks.

Chapter 5

Results

In this chapter, we examine the performance of various node embedding techniques used for predicting node heaviness across multiple datasets. By analyzing the results and visualizations, we aim to understand the effectiveness and robustness of each method, especially under varying levels of tolerance for prediction errors.

5.1 Accuracy of Predictions

To thoroughly assess the performance of our node heaviness prediction model across different datasets, we need to identify the optimal balance between prediction accuracy and allowable error tolerance. Since the number of triangles adjacent to each node in the dataset follows an simil-exponential distribution, we can pinpoint the critical position where the curve shows a sharp change in curvature (elbow). This inflection point, where the number of adjacent triangles sharply increases, will be used to set the tolerance level for each prediction.

We will vary the tolerance from 0% (exact match) up to 50% (half the value of the identified node). Additionally, before calculating the accuracy, we will round the predictions to the nearest integer since the target values are integers.

We choose this value because it represents the critical division between a small number of very heavy nodes and a large number of lighter nodes. For heavy nodes, which have high numerical values and are thus more susceptible to small errors, a higher tolerance is necessary to accommodate the considerable numerical differences.

For instance, in datasets like Arxiv, where heavy nodes have values in the tens of thousands, this is crucial. Conversely, lighter nodes, which have lower values, need a lower tolerance due to their small numerical range. A percentage-based tolerance relative to the node's value is not ideal because it results in excessively tight tolerances for light nodes, leading to misclassification of small variations as significant errors. This approach also disproportionately affects heavy nodes, potentially leading to less accurate predictions.

These evaluations will be based on the rounded results of the test set of the simple NN defined in Chapter 4.

5.1.1 Determining the elbow point

Given two arrays x and y, the objective is to identify the point on the curve that is farthest from the straight line connecting the first and last points of said curve. We define the starting point $p_1 = (x_0, y_0)$ and the ending point $p_2 = (x_n, y_n)$ and calculate the vector $v_l = (x_n - x_0, y_n - y_0)$ representing the line from the start point p_1 to the end point p_2.

Then we calculate the projection of v_i onto v_l and we can finally get the elbow point by calculating the point with the maximum perpendicular distance: $i_e = \arg\max(\|v_i - \text{Proj}(v_i, v_l)\|)$. We here show the elbow values of the examined datasets in Table 5.1.

	Citeseer	**PubMed**	**PPI**	**WikiCS**	**Arxiv**	**Products**
Elbow Value	5	10	158	5166	780	264
Top % of Nodes	96.00%	95.61%	90.15%	95.46%	99.45%	98.24%

Table 5.1: Elbow value and amount of node with lower value in the examined datasets

5.1.2 Accuracy Results

As the tolerance increases from 0% to 50%, the accuracy of all methods generally improves, which is expected because a higher tolerance allows for more lenient error margins. However, the rate of accuracy improvement varies across methods and datasets, indicating that some predictions are more accurate than others.

Firstly, it is important to note, as shown in Table 5.2, that a significant portion of the nodes exhibit a heaviness level equal to zero. This has a considerable impact on the levels of accuracy, as this majority of nodes become easy to predict, and their correct classification tends to partially obscure the potential challenges in predicting the heavy nodes, which are fewer and exhibit much more variable values.

Citeseer	**Pubmed**	**PPI**	**WikiCS**	**Arxiv**	**Products**
69.61%	75.56%	28.13%	12.50%	26.06%	23.11%

Table 5.2: Percentage of nodes with 0 heaviness in the datasets.

In any case, we observe that in dense datasets, where the number of triangles adjacent to the nodes is very high, the neural network struggles to provide precise values imme-

diately. This is likely due to the presence of extreme values, which make it difficult to fine-tune the weights more accurately. This phenomenon is also present in sparser graphs, but rounding the final output to the nearest integer helps mitigate this issue.

Figure 5.1 presents the test set accuracy of various models across different tolerance levels for the Citeseer dataset. We observe that GIN consistently achieves top-tier accuracy, significantly outperforming other embedding methods across the board. SNOW also demonstrates strong performance, particularly at lower tolerance levels, where it notably surpasses node2vec. In contrast, GraRep records the lowest accuracy, characterized by a substantial variance across tolerance levels.

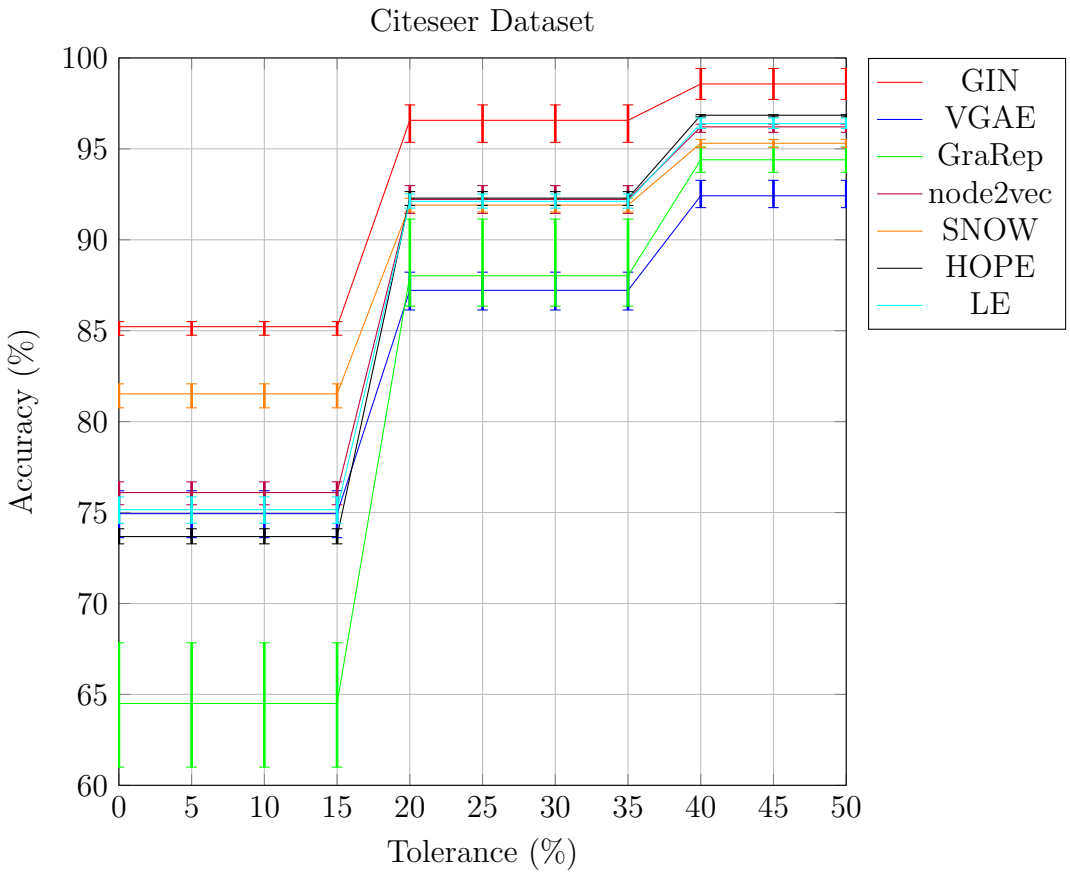


Figure 5.1: Accuracy of various models on the Citeseer dataset ($d = 64$).

The PubMed dataset results (Figure 5.2) show a slightly different trend. Here, GIN continues to lead in accuracy, though with a narrower margin than observed in Citeseer. SNOW initially underperforms in comparison to node2vec, yet it recovers and matches node2vec at higher tolerance levels. GraRep, which struggled in the previous dataset, improves on PubMed, delivering accuracy comparable to other embedding methods.

Turning to the PPI dataset (Figure 5.3), the high density of the graph results in very low accuracy at zero tolerance, but it improves markedly at low tolerance levels. HOPE, which achieved only moderate results on previous datasets, claims and maintains the highest accuracy here, closely followed by GIN. Meanwhile, SNOW lags significantly behind.

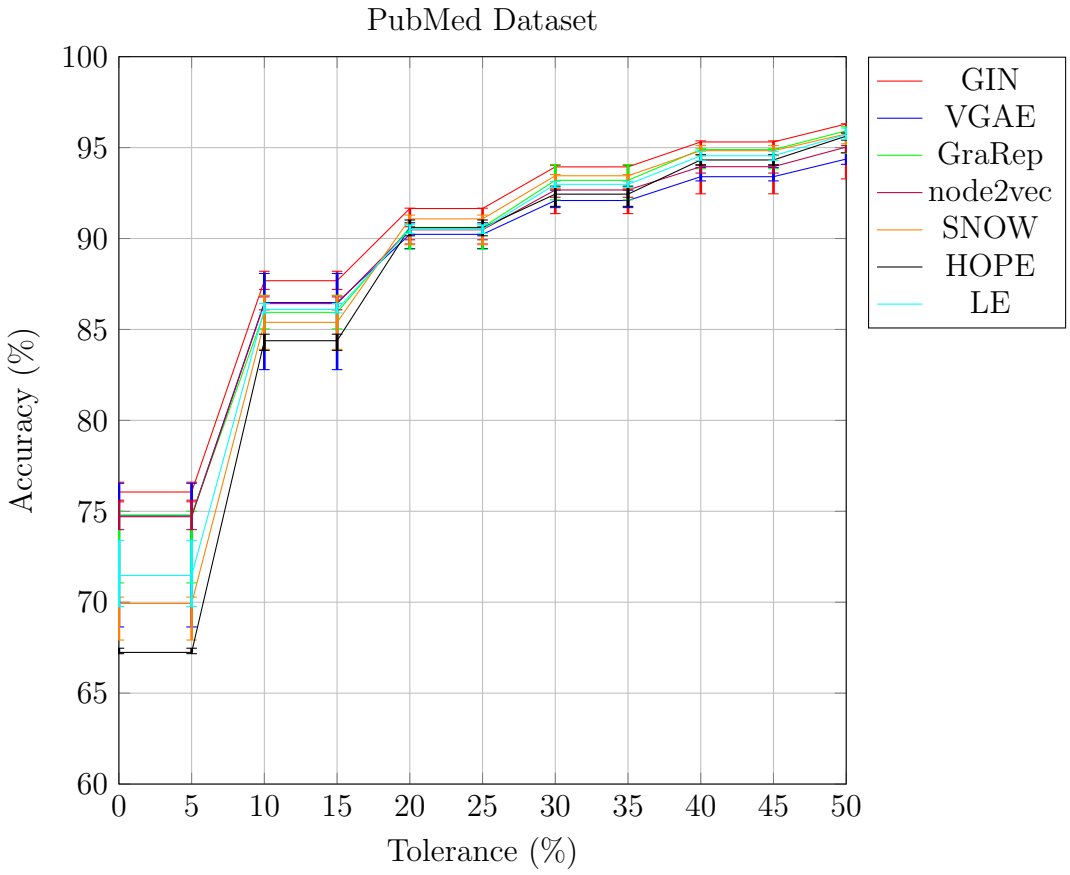


Figure 5.2: Accuracy of various models on the PubMed dataset ($d = 64$).

For the WikiCS dataset (Figure 5.4), which is structurally similar to PPI but with more features, LE emerges as the top performer at low tolerance levels, with GIN in close pursuit. VGAE, however, struggles considerably, particularly at low tolerance levels where its accuracy remains much lower than other methods.

The Arxiv dataset results (Figure 5.5) reflect its unique node characteristics as many nodes have a very low number of triangles adjacent. This trait makes most nodes relatively easy to classify, resulting in accuracy levels approaching 100% even at low tolerance. Here, node2vec excels, surpassing matrix factorization methods such as GraRep and LE.

Finally, Figure 5.6 presents the accuracy trends for the Products dataset. Due to the dataset's substantial size, only some of the models have been tested because of memory and runtime limitations. At low tolerance, LE and SNOW outperform GIN by a noticeable margin. However, GIN performs better at zero tolerance and achieves accuracy comparable to other models at higher tolerance levels.

At the level of individual results for each type of embedder, we notice that `GIN` consistently ranks among the top embedding methods (as shown in Table 5.3), although it slightly underperforms in the case of dense graphs, where it still achieves results comparable to those of other methods.

In general, the methods tend to behave differently depending on the type and structure of the graph they are applied to, making it difficult to provide a clear ranking of the

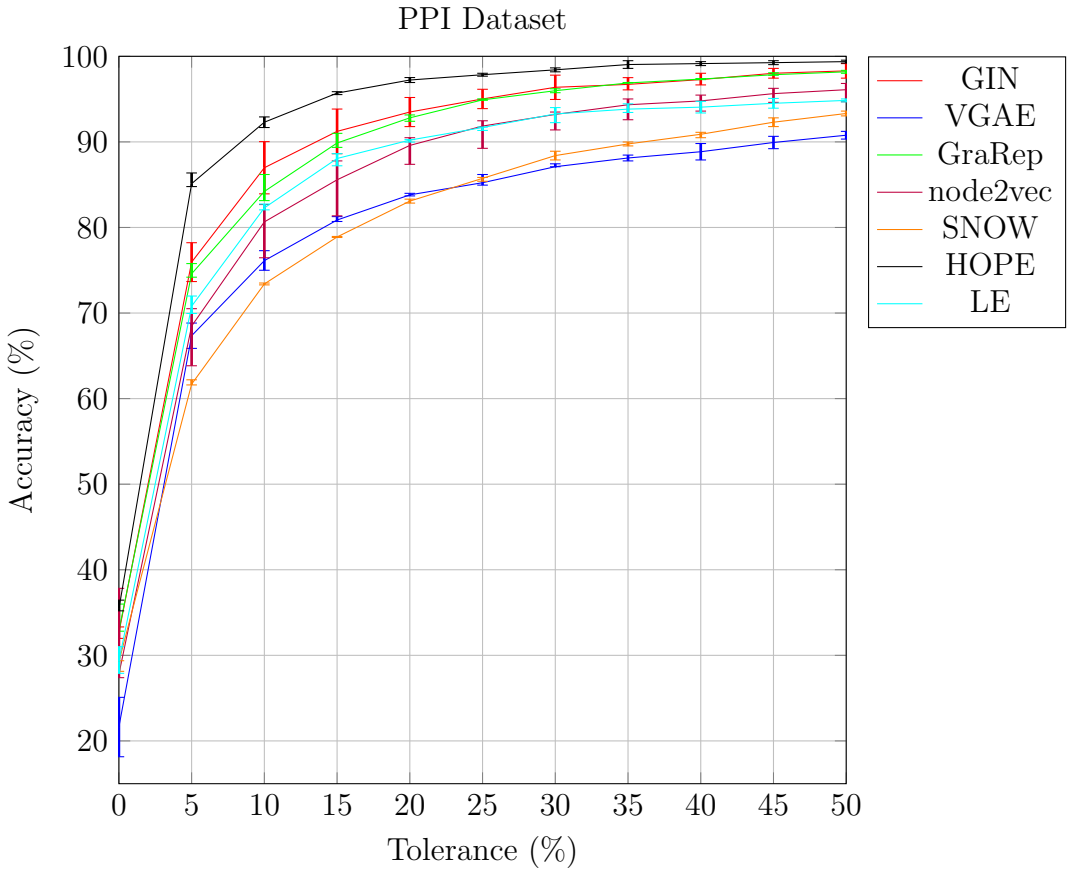


Figure 5.3: Accuracy of various models on the PPI dataset ($d = 64$).

best embedding methods in terms of accuracy.

`SNOW`, which performs well in sparse graphs, tends to suffer in dense graphs, as the presence of high-degree nodes increases the likelihood that all random walks will fail without finding triangles. This is the main problem within `SNOW`, which can be observed in the plots provided in the appendix. While `SNOW` correctly classifies a significant portion of the nodes, albeit with some noise, a considerable portion of the heavy nodes is misclassified as really light. This bias in the runs should be examined and addressed, as in the absence of this issue, `SNOW` could prove to be a viable alternative to random walk methods such as `node2vec`. Nevertheless, we can observe that `SNOW` is better at recognizing heavy nodes compared to `node2vec` directly from the embeddings, as can shown in Figure 5.7 from the UMAP visualizations of the `SNOW` and `node2vec` embeddings.

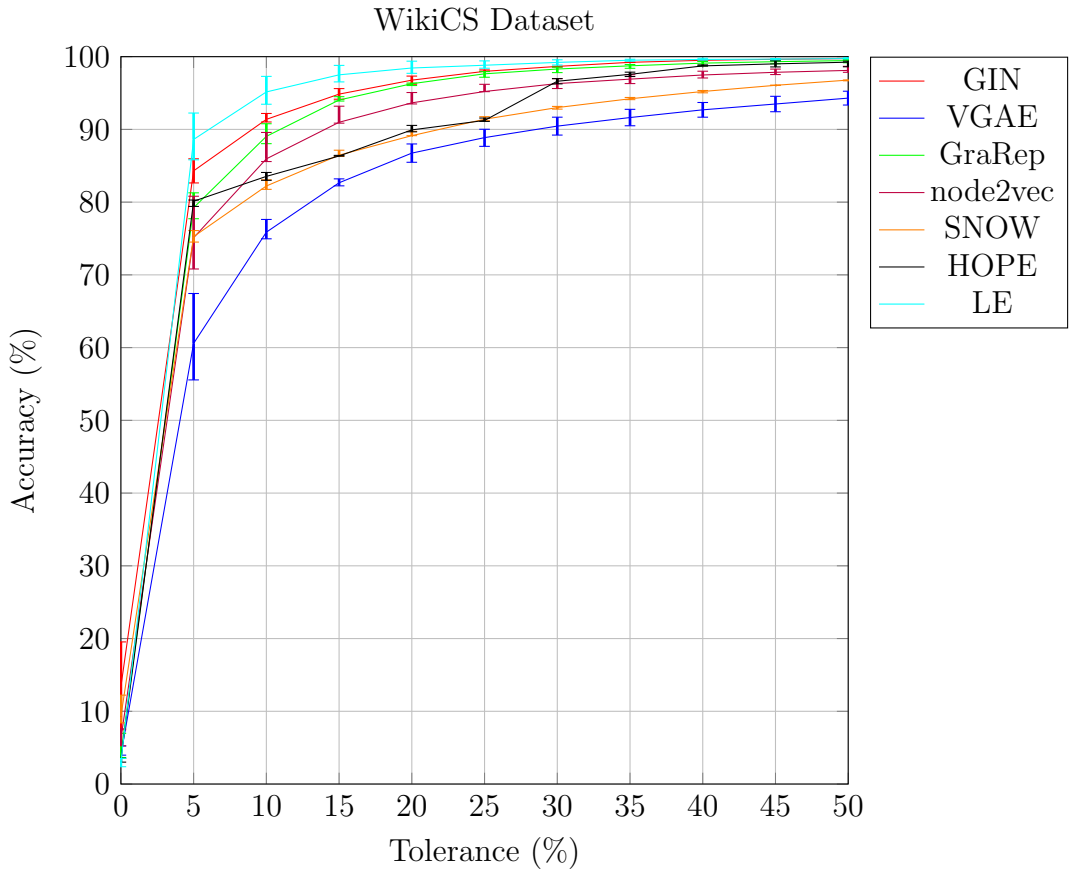


Figure 5.4: Accuracy of various models on the WikiCS dataset ($d = 64$).

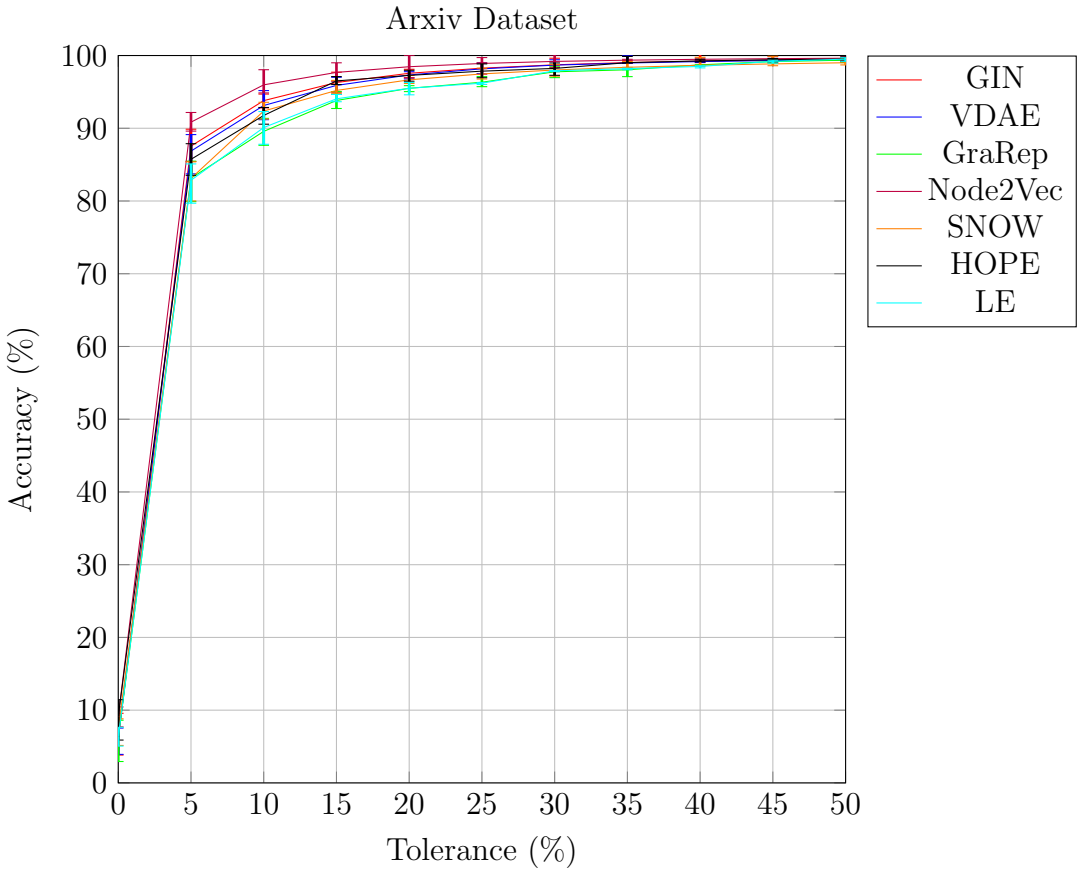


Figure 5.5: Accuracy of various models on the Arxiv dataset ($d = 64$).

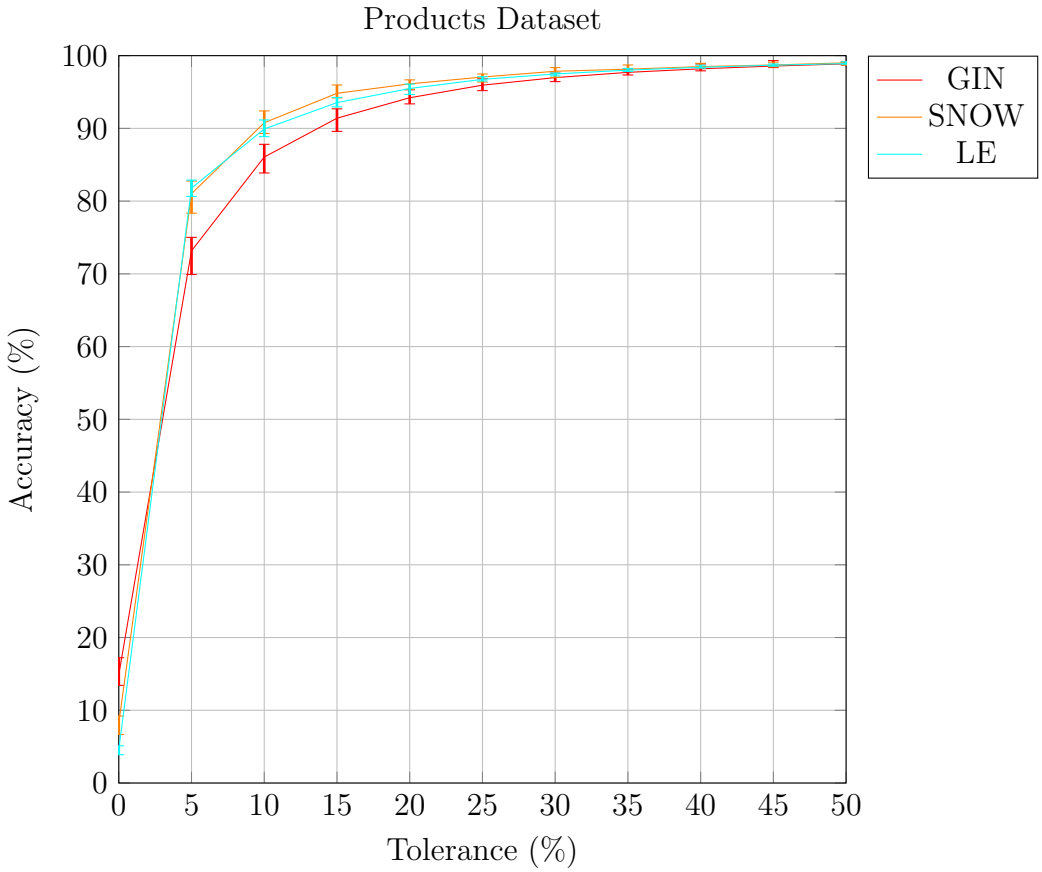


Figure 5.6: Accuracy of various models on the Products dataset ($d = 64$).

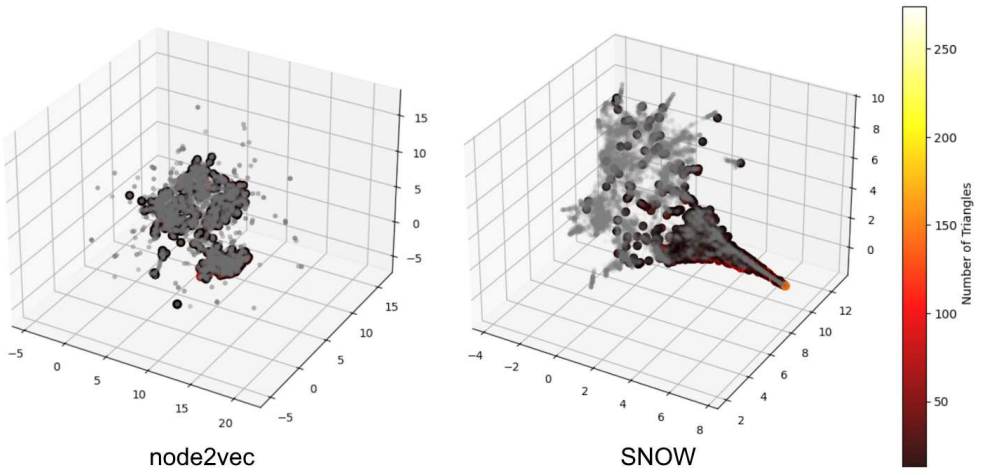


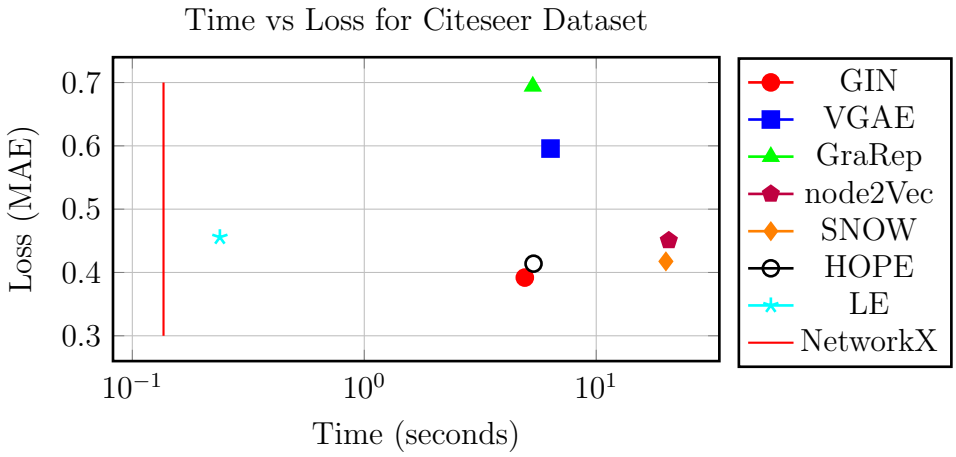
Figure 5.7: 3D UMAP plot of node2vec and SNOW on the PubMed dataset: In the SNOW plot, the heavy nodes are distinctly separated from the light ones, demonstrating the algorithm's enhanced ability to differentiate node types. (Nodes with 0 heaviness are shown in grey)

Dataset	GIN	VGAE	GraRep	node2Vec	SNOW	HOPE	LE	SDP
CiteSeer	0.3919	0.5957	0.6940	0.4506	0.4175	0.4139	0.4557	0.7782
PubMed	0.8053	1.0475	1.3015	1.0974	1.1109	1.0178	0.9437	2.2348
PPI	8.6022	28.1166	10.7680	21.2688	26.1585	7.0973	53.1471	37.5665
WikiCS	141.6618	207.6236	196.2895	308.7476	348.9465	205.7345	148.6957	469.2210
Arxiv	26.5038	28.3918	34.9250	25.2351	32.8094	29.7230	20.1128	35.0209
Products	13.8710	–	–	–	32.4755	29.7230	19.2938	–

Table 5.3: Average MAE for different embedding methods across various datasets. The best result in each row is highlighted in green. All accuracy results are within a 8% variance margin.

5.1.3 Time efficiency

We now analyze the accuracy of various embedding methods in comparison to the time required to compute them. It is immediately evident that the direct calculation of triangles using the "triangles" function from NetworkX is the most efficient both in terms of time and accuracy, as it is a precise calculation rather than an estimate. However, several observations are worth analyzing in detail (As shown in Figure 5.8).

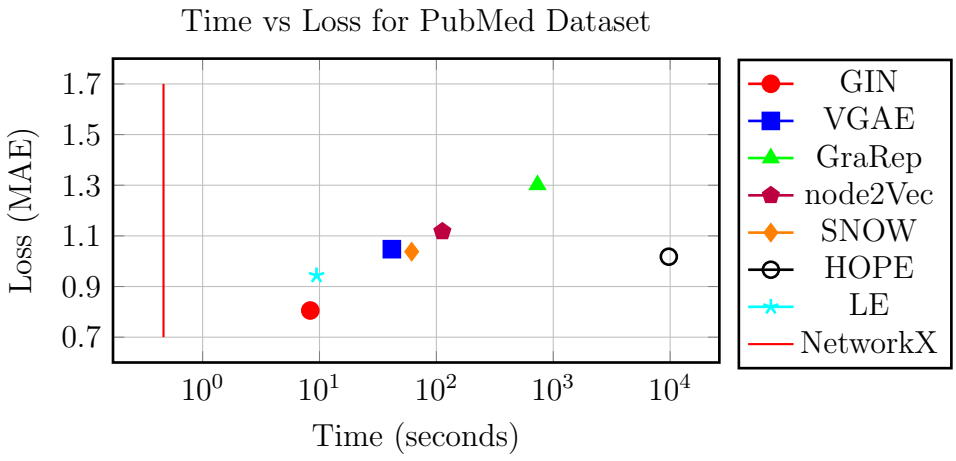


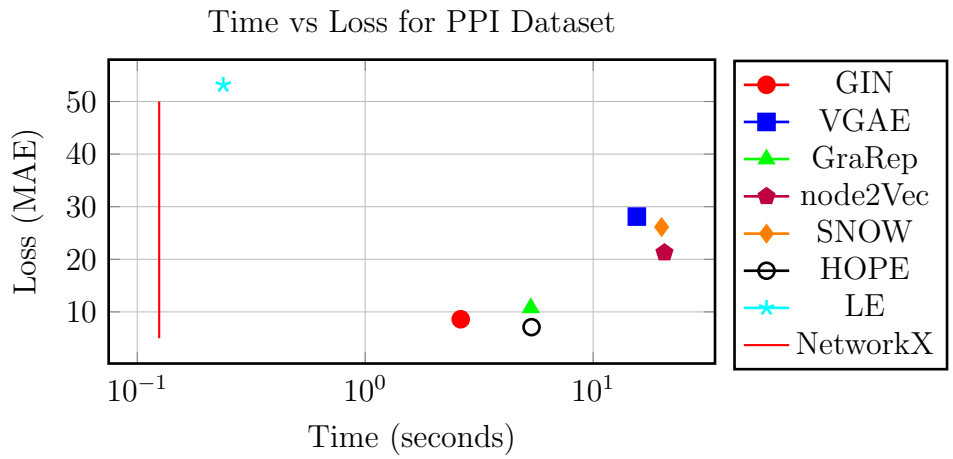
Among the deep learning methods, we note that the `GIN` algorithm, which has a considerable training time in datasets with few nodes (e.g., Citeseer and PPI), tends to scale more efficiently than other methods as the number of nodes increases. This results in `GIN`'s computation time approaching that of NetworkX for larger datasets.

In contrast, `LE` (Laplacian Eigenmaps) starts off very quickly on smaller datasets but soon encounters significant slowdowns due to its cubic time complexity.

`SNOW` and `node2vec` have very similar computation times, and in general, random walk-based methods tend not to be among the fastest or the most accurate.

Finally, while `HOPE` demonstrates a good level of accuracy on smaller datasets, it quickly becomes impractical in terms of time when applied to larger graphs.





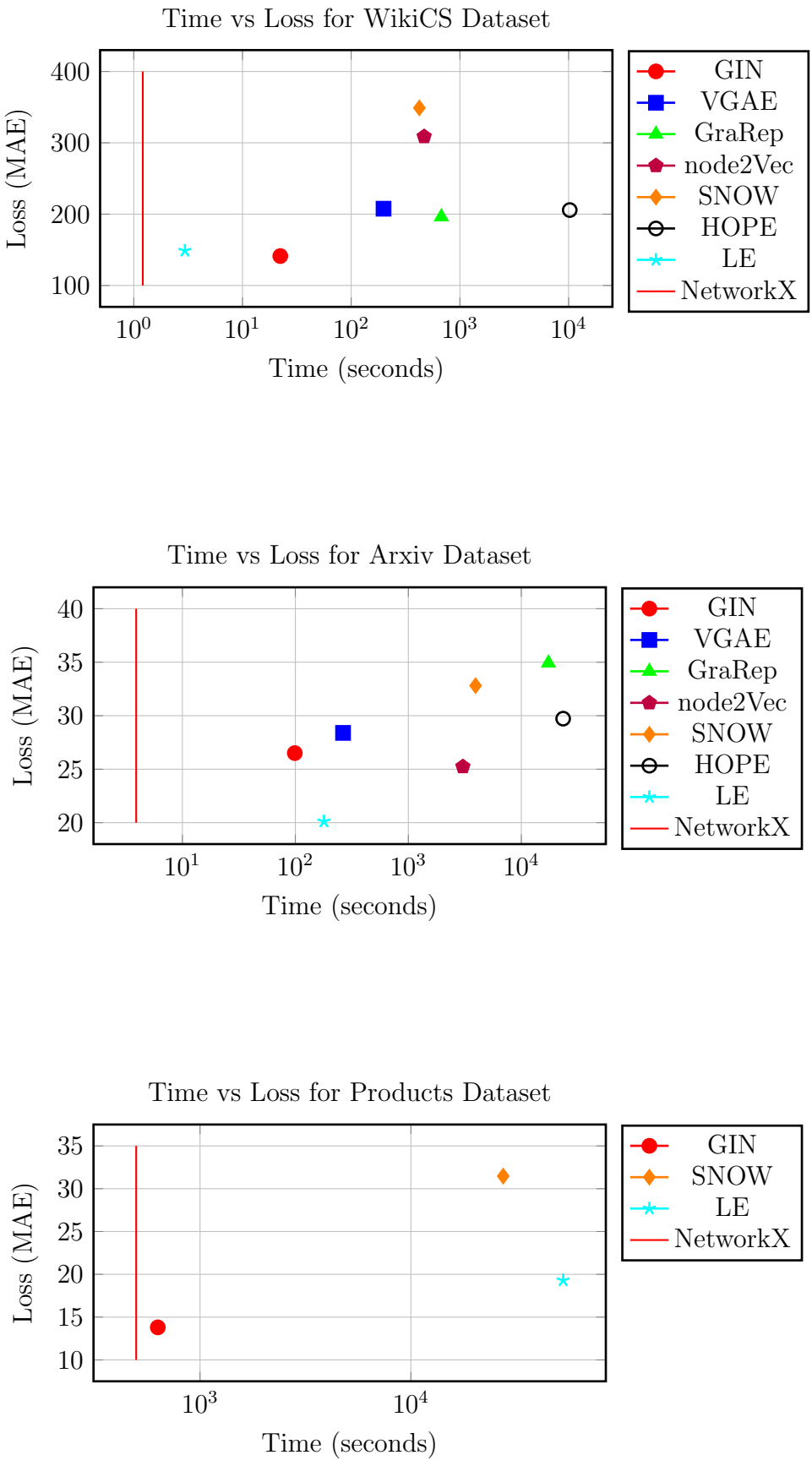


Figure 5.8: Performance of Embedding Methods: Time vs Loss (MAE)

5.1.4 Simple Degree Predictor (SDP)

We now analyze the performance of the *simple degree predictor*. The best values for α and β for each dataset are shown in Table 5.4.

We recall the formula of the predictor:

$$\tilde{N}_{\mathcal{M}(u)} = \alpha \cdot deg(u) + \beta$$

As shown in Table 5.3 the performance of SDP is inferior to all the embedding methods analyzed. However, the difference is not extreme, and as shown in Figure 5.9, the predicted values tend to correctly follow the curve of the true values. Nonetheless, the results on the WikiCS dataset reveal a challenge: when the distribution of the heaviness does not follow a linear function with the node degrees, it becomes difficult to make accurate predictions. In the case of WikiCS dataset, the high-degree nodes are significantly under-predicted.

Dataset	α	β
Citeseer	0.8467	-1.2646
PubMed	0.8020	-1.7008
PPI	5.7636	-42.8670
WikiCS	25.2797	-107.1657
ArXiv	6.1901	-45.0748

Table 5.4: Best α and β values for different datasets on the simple degree predictor.

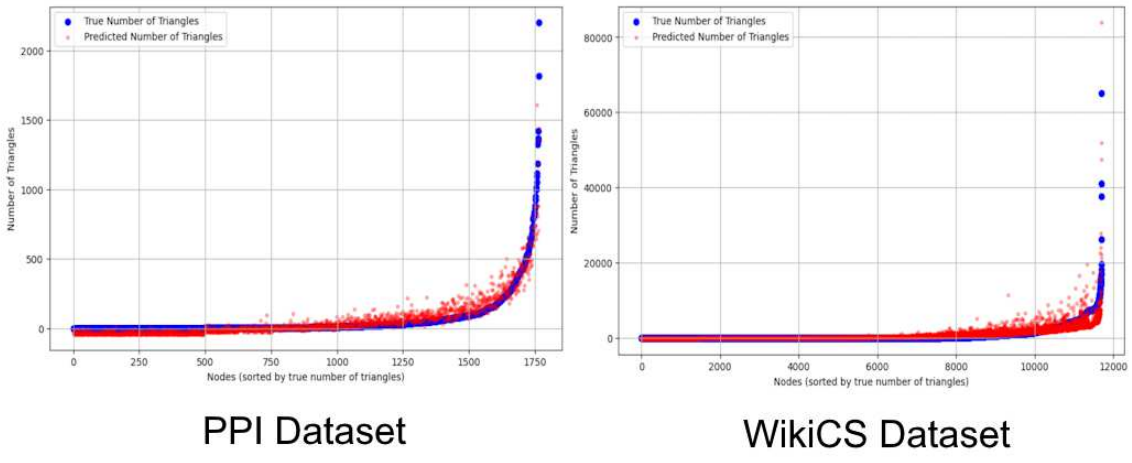


Figure 5.9: Prediction plots of the simple degree predictor on PPI and WikiCS datasets (The red dots are the predictions while the blue ones are the true values of the heaviness)

This highlights that SDP is highly susceptible to the type of graph and its distribution, although it is extremely efficient in terms of computation time by being able to find the

optimal α and β in minimal amount of time, even on large datasets like Products and Arxiv.

5.2 Accuracy of Predictions on Partial Graph Visibility

Unlike other types of embeddings such as random walk and matrix factorization, which require the entire graph as input, GNN-based embeddings can learn with just a subgraph containing some nodes and edges.
Therefore in this section, we analyze the accuracy performance of `GIN` and `VGAE` when graph visibility is reduced. Specifically, we randomly select 40% of the edges from the datasets along with their corresponding nodes. We then train `GIN` and `VGAE` on this subgraph and embed the remaining 60% of the edges and their corresponding nodes.

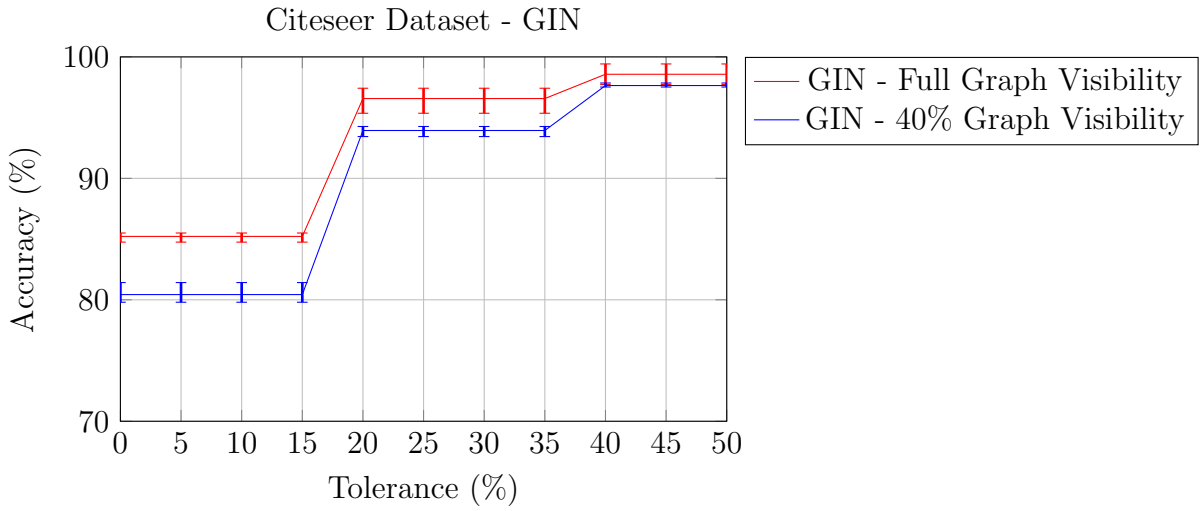


Figure 5.10: Accuracy of GIN on the Citeseer dataset with full and partial graph visibility.

We observe that the accuracy is generally lower when graph visibility is reduced, particularly, as we can see in Figure 5.10, in the case of `GIN`, whereas `VGAE` is less affected by this reduction (as shown in Figure 5.11). This effect is more pronounced in datasets with a smaller number of nodes and a higher edge density, such as the PPI dataset. However, despite the performance drop, the variance in the results does not increase significantly and remains relatively contained. Furthermore, even with reduced performance, `GIN` still achieves relatively high levels of accuracy.

Training time is highly dependent on the number of nodes selected from the dataset. In dense graphs, the training time with partially visible graphs is almost the same as that for fully visible graphs. However, for sparser datasets like PubMed and Arxiv, the training time is significantly reduced with partial graph visibility.

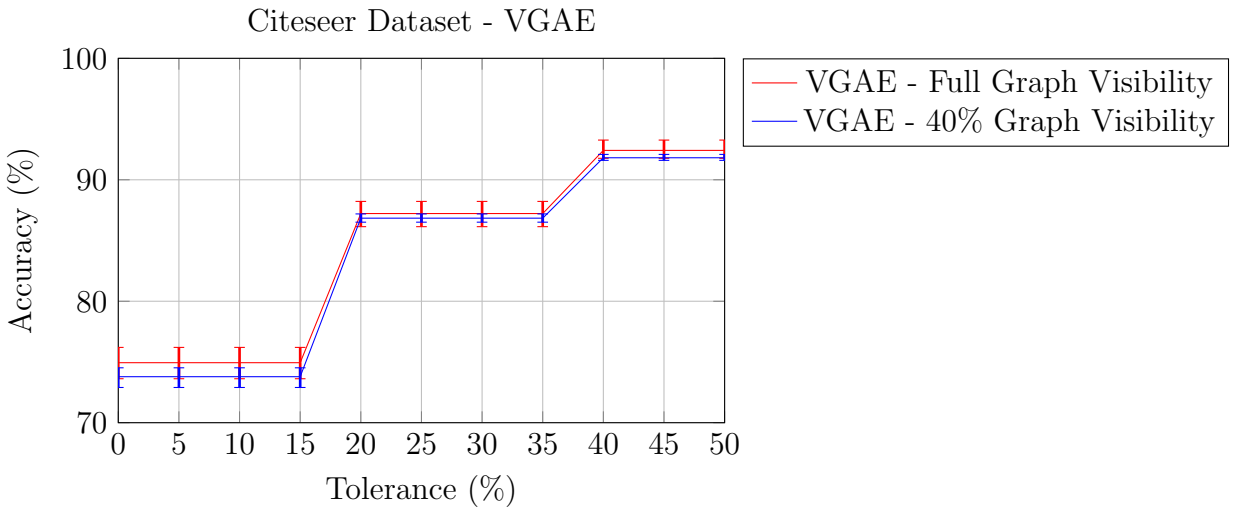


Figure 5.11: Accuracy of VGAE on the Citeseer dataset with full and partial graph visibility.

5.3 Heavy Node Classification

We now conduct a binary classification analysis on the datasets to evaluate the ability of the predictor to identify heavy nodes, defined as the top X% of nodes with the highest number of adjacent triangles.
We tested thresholds of 1%, 5%, 10%, and 25% to explore the performance of different embedding methods. Our approach involved comparing the top X% of nodes identified through our predictions with those identified through ground truth values.

Dataset	GIN	VGAE	Grarep	n2vec	SNOW	HOPE	LE	SDP
Citeseer	0.86	0.96	0.76	0.81	0.74	0.91	0.77	0.79
Pubmed	0.90	0.85	0.77	0.79	0.78	0.77	0.79	0.58
PPI	1.00	0.83	0.94	0.78	0.84	1.00	0.31	0.88
WikiCS	0.92	0.53	0.77	0.70	0.76	0.68	0.86	0.54
Arxiv	0.77	0.84	0.64	0.81	0.67	0.70	0.84	0.64
Products	0.86	–	–	–	0.69	–	0.80	–

GIN	VGAE	Grarep	n2vec	SNOW	HOPE	LE	SDP
0.86	0.67	0.69	0.72	0.69	0.86	0.75	0.31
0.83	0.60	0.66	0.64	0.66	0.65	0.70	0.19
0.89	0.83	0.94	0.78	0.89	0.78	0.83	0.39
0.75	0.48	0.71	0.58	0.79	0.68	0.91	0.25
0.63	0.54	0.55	0.64	0.40	0.58	0.66	0.88
0.73	–	–	–	0.32	–	0.65	–

Table 5.5: Precision and Recall values at 1% heavy nodes. All results are within a 6% variance margin.

Dataset	GIN	VGAE	Grarep	n2vec	SNOW	HOPE	LE	SDP
Citeseer	0.87	0.73	0.84	0.80	0.72	0.93	0.83	0.64
Pubmed	0.83	0.80	0.72	0.76	0.76	0.75	0.80	0.50
PPI	0.96	0.65	0.96	0.82	0.84	1.00	0.64	0.80
WikiCS	0.91	0.70	0.88	0.85	0.83	0.88	0.90	0.75
Arxiv	0.76	0.69	0.59	0.74	0.52	0.78	0.90	0.55
Products	0.77	–	–	–	0.56	–	0.80	–

GIN	VGAE	Grarep	n2vec	SNOW	HOPE	LE	SDP
0.68	0.50	0.48	0.62	0.57	0.70	0.73	0.58
0.77	0.40	0.53	0.65	0.57	0.69	0.72	0.80
0.96	0.80	0.96	0.82	0.85	0.93	0.85	0.88
0.95	0.59	0.89	0.77	0.81	0.75	0.91	0.28
0.46	0.48	0.38	0.70	0.31	0.43	0.57	0.96
0.64	–	–	–	0.34	–	0.69	–

Table 5.6: Precision and Recall values at 5% heavy nodes. All results are within a 6% variance margin.

Dataset	GIN	VGAE	Grarep	n2vec	SNOW	HOPE	LE	SDP	GIN	VGAE	Grarep	n2vec	SNOW	HOPE	LE	SDP
Citeseer	0.90	0.70	0.90	0.82	0.73	0.93	0.84	0.66	0.67	0.48	0.32	0.65	0.54	0.64	0.70	0.65
Pubmed	0.84	0.78	0.64	0.79	0.69	0.72	0.78	0.52	0.72	0.24	0.59	0.60	0.59	0.68	0.74	0.92
PPI	0.95	0.61	0.92	0.85	0.84	0.97	0.77	0.67	0.95	0.79	0.90	0.82	0.85	0.96	0.93	0.98
WikiCS	0.92	0.71	0.90	0.89	0.87	0.85	0.93	0.84	0.95	0.61	0.86	0.79	0.82	0.88	0.91	0.71
Arxiv	0.72	0.67	0.58	0.76	0.50	0.68	0.93	0.55	0.44	0.41	0.38	0.73	0.39	0.58	0.55	0.99
Products	0.75	–	–	–	0.54	–	0.81	–	0.66	–	–	–	0.39	–	0.76	–

Table 5.7: Precision and Recall values at 10% heavy nodes. All results are within a 6% variance margin.

Dataset	GIN	VGAE	Grarep	n2vec	SNOW	HOPE	LE	SDP	GIN	VGAE	Grarep	n2vec	SNOW	HOPE	LE	SDP
Citeseer	0.90	0.70	0.68	0.90	0.82	0.92	0.90	0.69	0.56	0.44	0.58	0.50	0.45	0.43	0.59	0.78
Pubmed	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
PPI	0.93	0.66	0.91	0.87	0.82	0.95	0.82	0.70	0.92	0.77	0.90	0.81	0.86	0.94	0.90	0.99
WikiCS	0.92	0.67	0.90	0.86	0.88	0.93	0.95	0.71	0.95	0.59	0.86	0.80	0.92	0.93	0.94	1.00
Arxiv	0.62	0.46	0.52	0.73	0.50	0.76	0.95	0.73	0.53	0.41	0.59	0.81	0.69	0.79	0.60	0.98
Products	0.75	–	–	–	0.54	–	0.88	–	0.79	–	–	–	0.62	–	0.84	–

Table 5.8: Precision and Recall values at 25% heavy nodes. All results are within a 6% variance margin. (At the 25% threshold for the PubMed dataset, we observed that the nodes have no adjacent triangles)

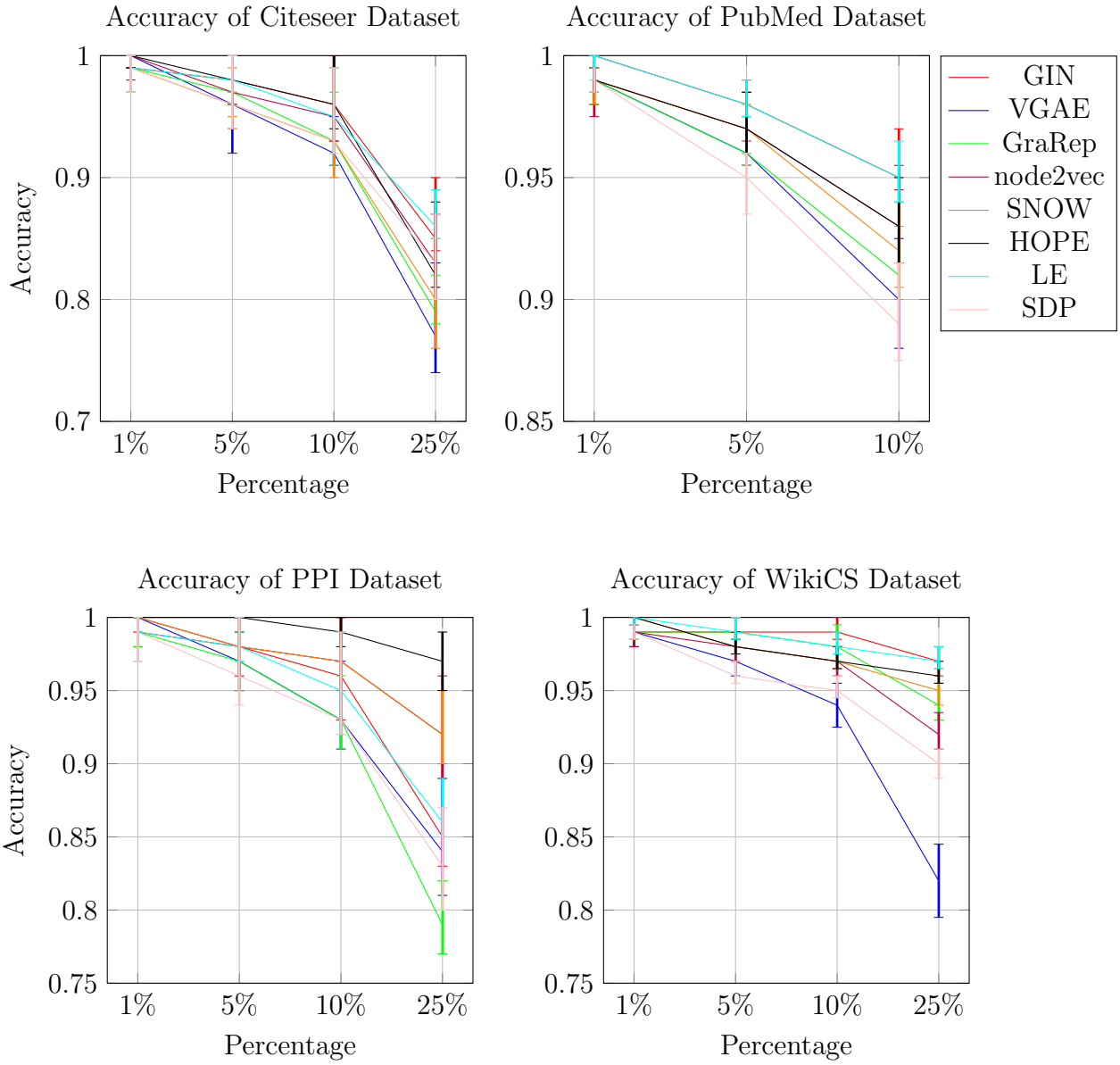
In Table 5.5 we present the test set precisions and recalls of various models across different datasets at 1% threshold. We observe that Graph Isomorphism Networks (`GIN`) maintain the highest levels of precision and recall across nearly all datasets. `Grarep` performs weakly in all cases except for PPI, while `SNOW`, `node2vec`, and `HOPE` yield mediocre results. In Table 5.6, at 5% threshold, `GIN` continues to achieve good results; however, `HOPE` and `LE` exhibit better precision values, highlighting an improvement over deep learning methods. At 10% and 25% threshold, as shown in Tables 5.7 and 5.8, the trend remains consistent, with `HOPE` and `LE` recording the best results, while `GIN`, despite providing good outcomes, struggles to compete effectively.

The results reveal that deep learning methods, such as Graph Isomorphism Networks (`GIN`), consistently perform best at lower thresholds, demonstrating superior accuracy and precision. However, `GIN` suffers heavily from low accuracy as the number of nodes surpasses the elbow point of the graph. While `GIN` achieves high precision and recall at the 1% and 5% thresholds, performance drops significantly when attempting to classify a larger portion of heavy nodes, as shown in Figure 5.12 which plot the accuracy of the test set of the various embedding models across different datasets. This decline highlights the method's limitations in scaling effectively to higher thresholds.

Matrix factorization methods like Laplacian Eigenmaps (`LE`) and `HOPE`, on the other hand, surpass deep learning methods as the threshold increases, particularly excelling in classifying larger node subsets. This suggests their effectiveness in handling higher

percentages of heavy nodes.

SDP performs relatively well in terms of recall at the 10% and 25% heavy node thresholds, but its generally low accuracy and precision present a different reality. These results indicate that while SDP can recall heavy nodes at higher thresholds, its overall performance is hindered by imprecision, which limits its reliability in more comprehensive classification tasks. In contrast, random walk-based methods perform poorly across all thresholds, indicating their unsuitability for this classification task.



5.4 Impact of Embedding Dimensions on Model Performance

In this analysis, we investigated the impact of varying embedding dimensions on model performance. Our experiments aimed to determine whether increasing the size of the

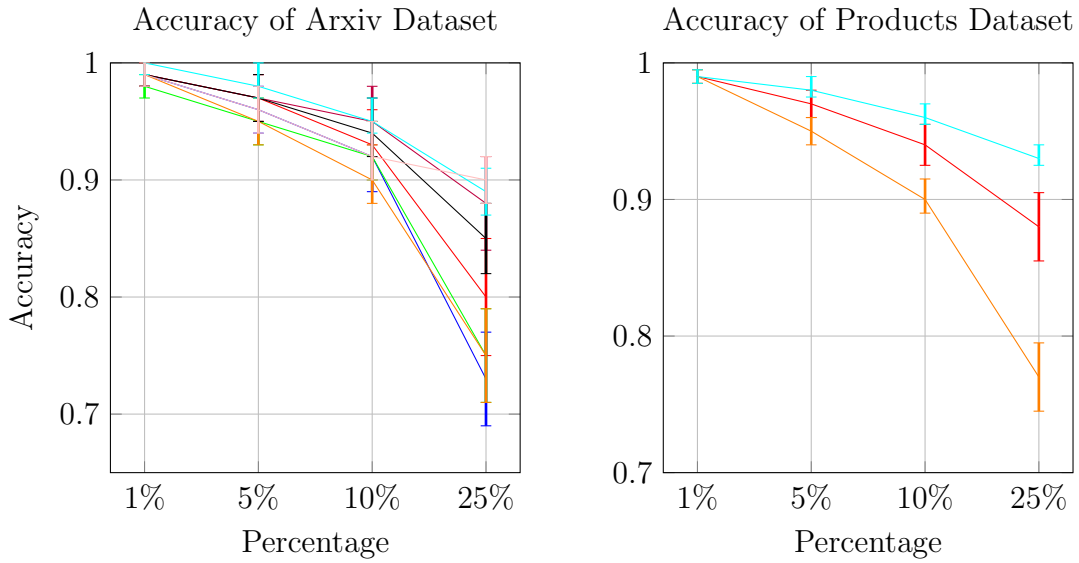


Figure 5.12: Accuracy of the datasets in predicting Heavy Nodes

embeddings would yield significant improvements in model accuracy and a reduction in the average loss (MAE).

For the `node2vec` algorithm, we can see in Table 5.9 a modest reduction of the loss when increasing the embedding size from 64 to 128 dimensions. However, beyond 128 dimensions, up to 256, the performance remained stable, showing no further gains. Notably, `SNOW` consistently outperformed `node2vec`, even with larger embedding dimensions. Although `node2vec` is designed to leverage larger dimensions to capture the overall graph structure, `SNOW` excels by focusing on a more localized neighborhood structure, which requires fewer dimensions to achieve high performance.

Dimension	GIN	VGAE	GraRep	node2vec	SNOW	HOPE	LE
d64	0.3919	0.5957	0.6940	0.4508	0.4175	0.4145	0.4557
d128	0.2915	0.5710	0.4827	0.4257	0.3753	0.3009	0.4408
d256	0.3015	0.5901	0.5356	0.4192	0.3704	0.3424	0.4450

Table 5.9: Average Loss (MAE) for Different Embedding Dimensions on Various Algorithms on Citeseer dataset. The best result in each row is highlighted in green.

The `HOPE` algorithm exhibited a consistent, albeit slight, improvement with larger embeddings. This trend was similarly observed with `VGAE` and `SNOW`, where performance increased incrementally with embedding size.

Laplacian Eigenmaps showed a gradual decrease in MAE with increased dimensions, although its accuracy did not improve significantly. `GIN`, which had the lowest MAE across all dimensions, improved in accuracy at 128 dimensions but stabilized at 256 dimensions, maintaining its position as the top performer across all metrics.

Overall, GIN achieved the highest accuracy and the lowest MAE across all embedding sizes in the tests, indicating its robustness and effectiveness in handling varying embedding sizes. Despite the various improvements observed across different algorithms and embedding sizes, the general shape of the performance graphs remained consistent, indicating that while embedding size influences model performance, the trend and structure of the results are relatively stable across different dimensions.

For most algorithms, increasing the embedding dimension up to 128 provides notable improvements in performance and reduction in MAE, but further increases up to 256 dimensions yield diminishing returns. This suggests that while larger embeddings can enhance model accuracy and reduce errors, the benefit of very large embeddings may be limited.
The general performance trends for the algorithms remained consistent regardless of embedding size, indicating that while embedding dimension does affect performance, the overall trends are stable. This underscores the importance of selecting an embedding size that balances performance gains with computational efficiency.

5.5 Predicting Node Heaviness with more complex Motifs

In this section, we analyze the ability of different embedding methods to predict more complex motifs, specifically 4-cycles and 4-cliques which can be seen in Figure 5.13.

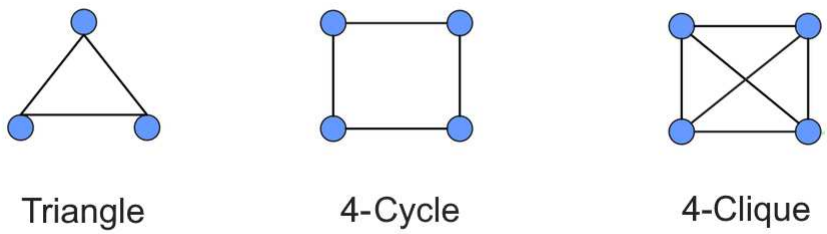


Figure 5.13: Illustration of the motif analysed.

Our results indicate that GIN and HOPE achieve the best performance in both 4-cycles and 4-cliques, significantly outperforming other embedding methods (As shown in Table 5.10).

We observe that Laplacian Eigenmaps (LE) drastically degrades in performance when compared to its predictions for triangles as seen in Figures 5.14 and 5.15. This drop in

accuracy suggests that LE struggles to capture more complex structures like 4-cycles and 4-cliques.

Motif	GIN	VGAE	GraRep	node2vec	SNOW	HOPE	LE	SDP
4-cycles	0.4097	1.2702	0.9053	1.1136	0.8975	0.4488	2.0298	7.3141
4-cliques	1.4873	3.6338	1.9208	3.0374	3.2801	1.4018	8.7963	6.8795

Table 5.10: Performance metrics for predicting 4-Cliques on PPI datasets and 4-Cycles on Citeseer dataset. The best result in each row is highlighted in green.

These results align with the concept of order proximity defined in Table 2.1. GIN and HOPE benefit from k-th order proximity, allowing them to capture deeper relational structures beyond the immediate neighborhood, which is crucial for predicting complex motifs. In contrast, LE, which operates with only 1st order proximity, encounters difficulties in identifying these more intricate patterns.

While SNOW performs reasonably well in predicting 4-cycles, it struggles with 4-cliques. This outcome is expected, as SNOW's architecture is more suited to predicting cyclic motifs rather than dense structures. Interestingly, node2vec shows a similar trend, with both SNOW and node2vec producing nearly identical results.

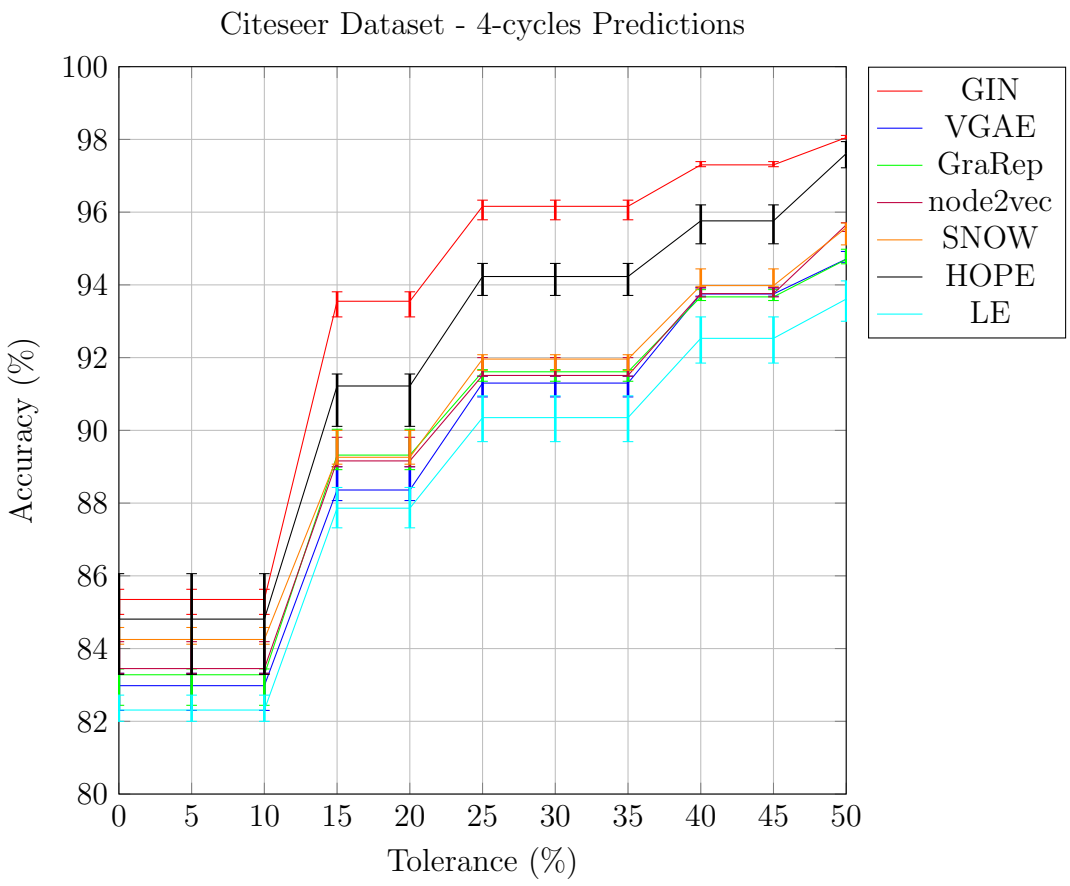


Figure 5.14: Citeseer Dataset - 4-cycles Predictions

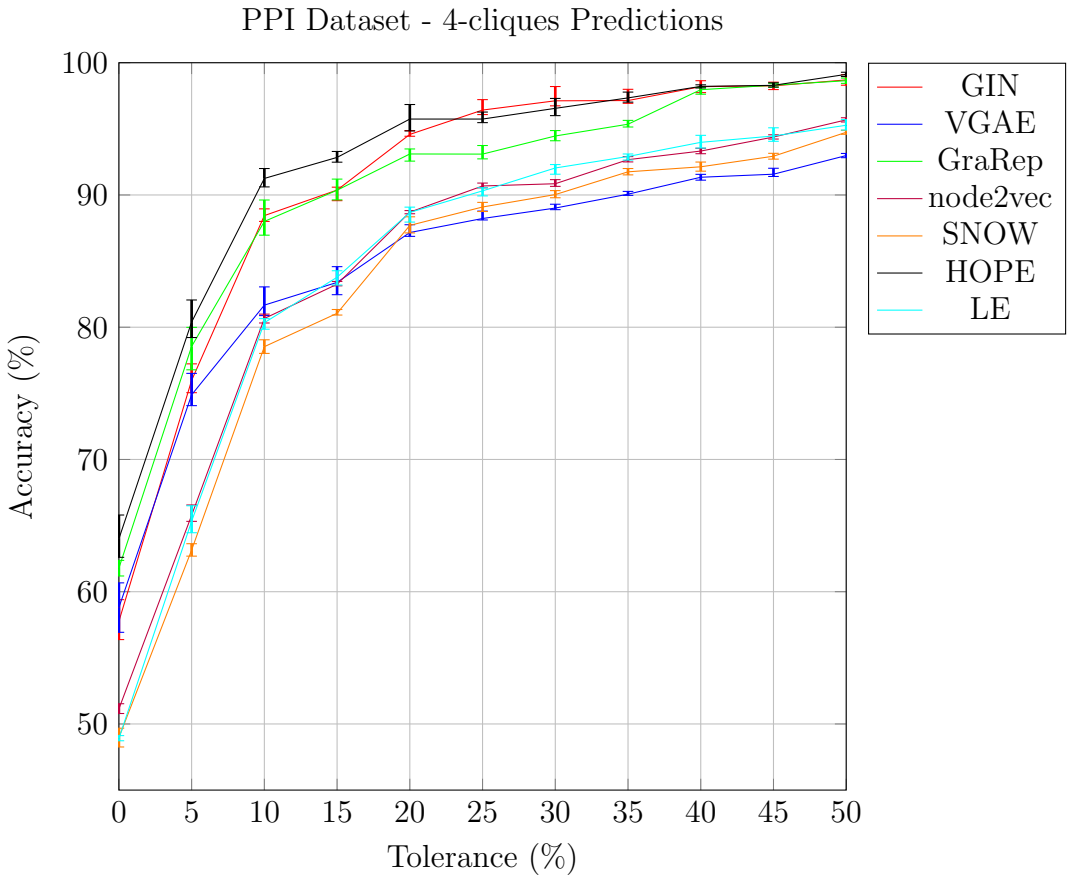


Figure 5.15: PPI Dataset - 4-cliques Predictions

Chapter 6

Conclusions

In this thesis, we explored the intricate relationship between node embedding techniques and the accuracy of motif estimation in complex networks. We began by discussing the significance and relevance of graph analysis across various domains, illustrating how graph embeddings have emerged as powerful tools for understanding and modeling complex networks.

Graphs, characterized by their non-Euclidean structure, provide a flexible and universal framework for representing complex relationships within data, making them essential in fields such as social network analysis, biological systems, and computational linguistics.

Our goal was to investigate how the choice of embedding methods impacts the effectiveness of motif estimation. By introducing the concept of Node Heaviness and a novel node embedding technique specifically designed for motif estimation, we aimed to contribute valuable insights to the field of network analysis.

6.1 Key Findings

Our experimental results indicate that different embedding techniques exhibit varying levels of effectiveness depending on the dataset and the specific characteristics of the graph.

While the Graph Isomorphism Network (`GIN`) demonstrated strong performance overall, it occasionally faced challenges with dense graphs. In contrast, dedicated methods like `SNOW` performed adequately but not at the top level, indicating room for improvement. This variability underscores the necessity for practitioners to choose embedding techniques tailored to the specific structure of their networks to maximize effectiveness.

In terms of computational trade-offs, we found that direct computation methods, such as those utilizing NetworkX, often provided the most accurate results with minimal computational effort. On the other hand, while deep learning methods offer greater scalability, they may require additional tuning to perform well with specific datasets.

The analysis of heavy node classification highlighted that GIN excels at lower thresholds but experiences a decline in accuracy when attempting to classify larger subsets of heavy nodes. This indicates the value of employing a combination of methods, such as matrix factorization techniques, to ensure robust heavy node identification, particularly at higher thresholds.

Finally, we observed that reducing graph visibility affects the accuracy of node embeddings. GIN was more sensitive to this reduction compared to VGAE, suggesting that when dealing with incomplete networks, selecting methods less impacted by missing data can enhance motif estimation accuracy.

6.2 Future Work

These findings have several important implications for the field of network analysis. The good performance of SNOW in the realm of shallow embeddings suggests that incorporating domain-specific modifications to existing algorithms can yield significant improvements in tasks such as link prediction and node classification, though there is still room for enhancement.

Additionally, the promising results obtained from GIN and VGAE indicate that further exploration into deep learning approaches for graph embeddings could unlock new capabilities for handling more complex network structures.

6.3 Conclusion

In conclusion, this thesis advances the understanding of graph embeddings by evaluating the efficacy of current methods for predicting node heaviness.

Our results underscore the potential of tailored approaches like SNOW in capturing critical graph properties. Moreover, deep learning techniques such as GIN show significant potential in advancing the field by effectively managing the complexity of dense graphs.

The scalability and computational efficiency of these methods also point toward potential applications in real-world scenarios, as large graphs demand not only accuracy but also manageable execution times. This work lays the groundwork for future research aimed at enhancing the accuracy, scalability, and efficiency of graph embedding methods, ultimately enabling more effective and powerful graph analysis.

Appendix A

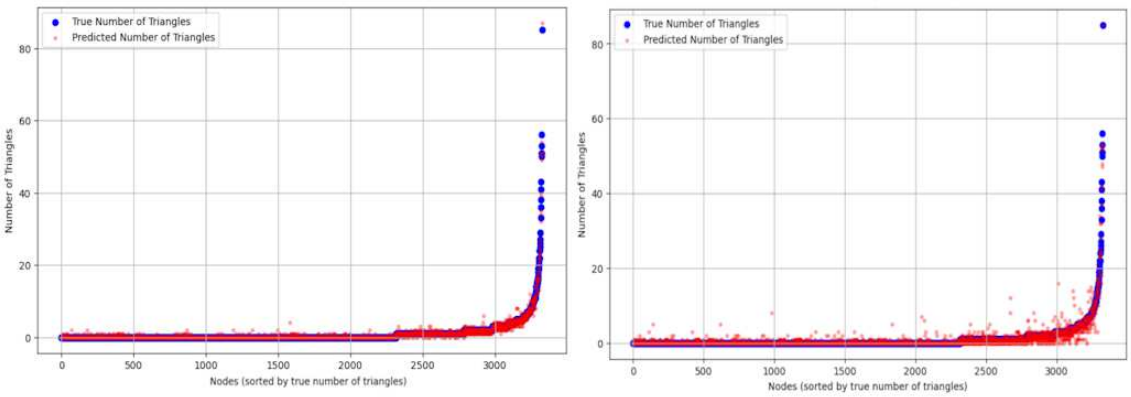
Plots

A.1 Plot of predictions on Citeseer dataset

In this section, we present the prediction plots for the various embeddings on the Citeseer dataset, where the embedding dimension is set to $d = 64$.

In the plots, blue points represent the correct values as determined by the ground truth, while red points indicate the predicted values.

The nodes have been ordered based on their correct values to make the distinction between light and heavy nodes more apparent. This ordering helps to clearly visualize the performance of the embeddings across different node categories.



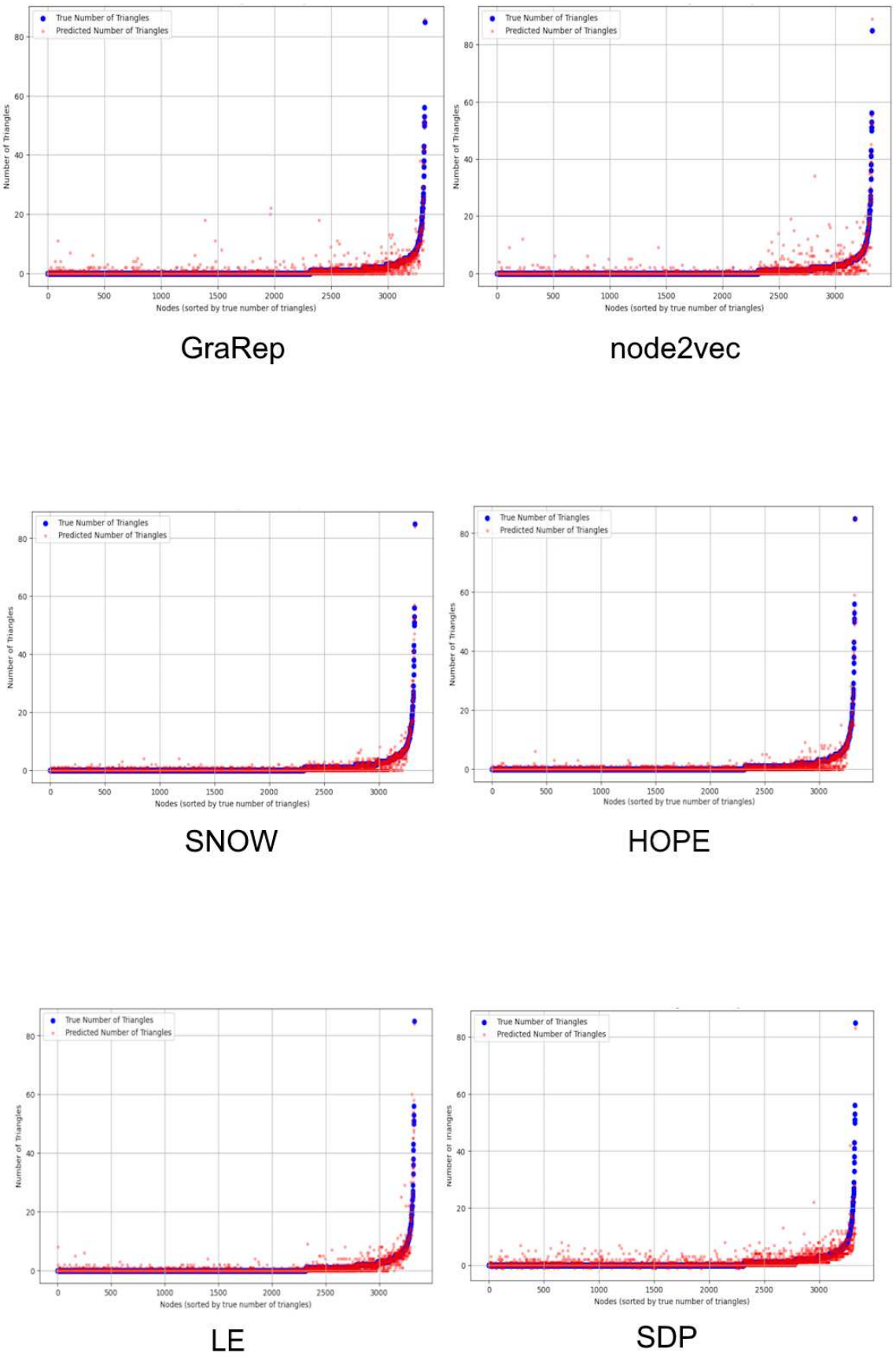
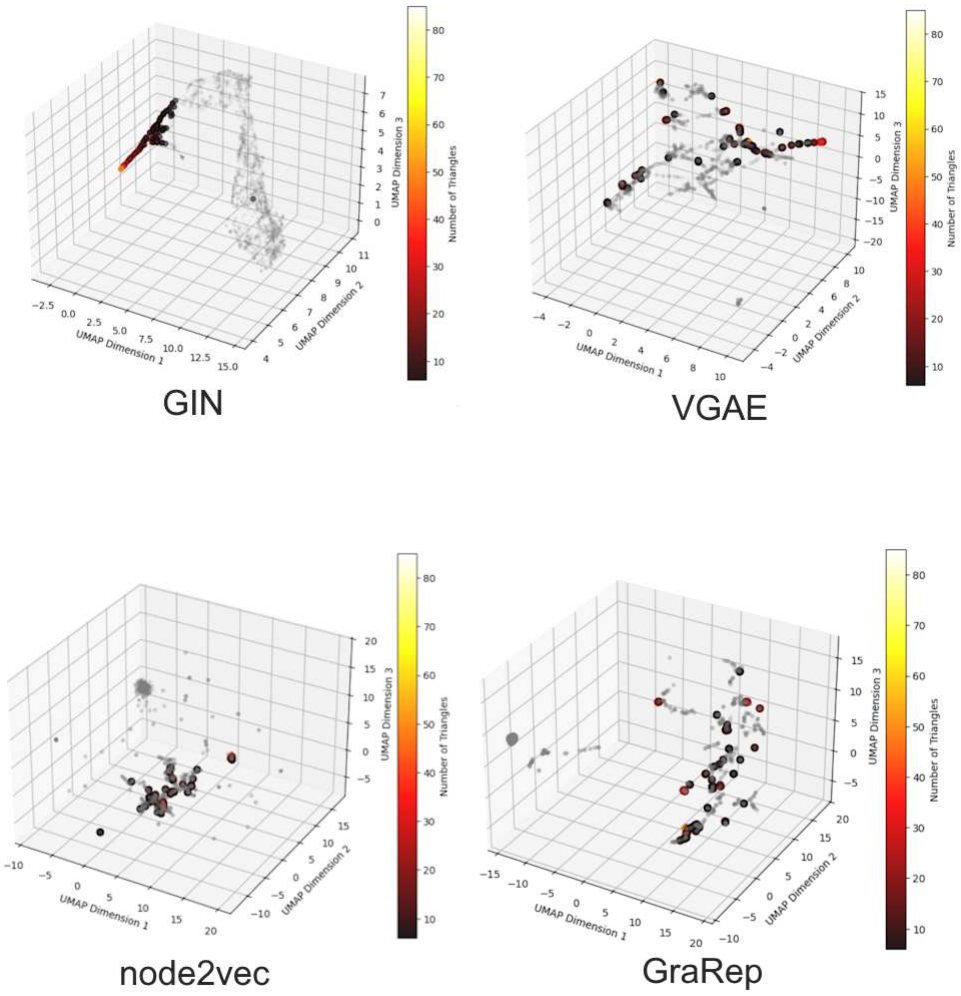


Figure A.1: Plot of predictions on Citeseer dataset

A.2 UMAP of embeddings on Citeseer dataset

In this section, we visualize the embeddings after applying UMAP to assess how effectively they distinguish between heavy and light nodes. Nodes with a heaviness value of 0 are marked in gray and are displayed with a smaller size, given their abundance in the dataset.

This visual adjustment ensures that they remain distinguishable from the heavier nodes, providing a clearer representation of the overall node distribution in the embedding space.



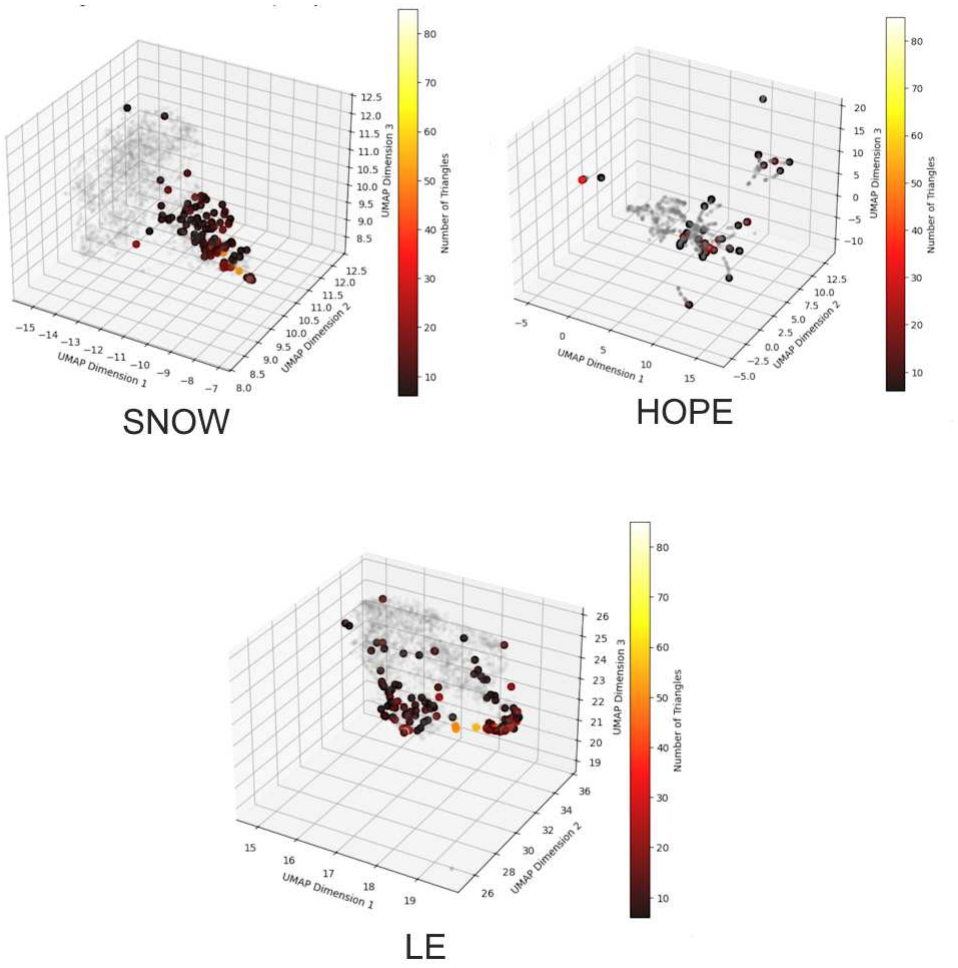


Figure A.2: UMAP of embeddings on Citeseer dataset

A.3 Plot of predictions on GIN embeddings across different datasets

In this section, we present the prediction plots for the `GIN` embedding across various datasets. As in the previous plots, blue points represent the correct values obtained from the ground truth, while red points indicate the predicted values. The nodes have been ordered according to their correct values to better highlight the distinction between light and heavy nodes. For datasets with a large number of nodes, a random subset of 15000 nodes has been selected for plotting to ensure the visual clarity of the results.

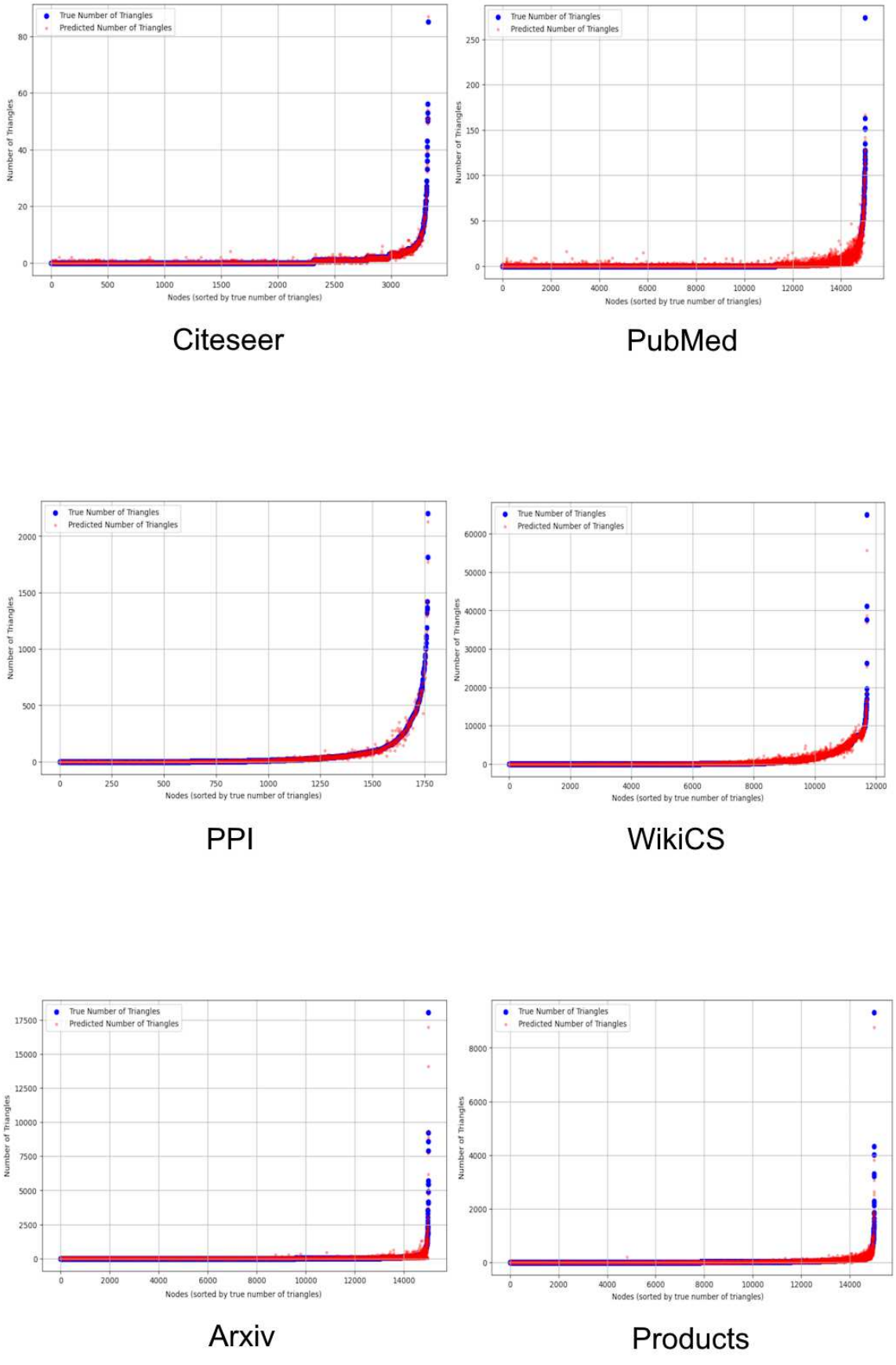


Figure A.3: Plot of predictions on GIN embeddings across different datasets

Bibliography

[1] Mengjia Xu, *Understanding Graph Embedding Methods and their Applications*, https://arxiv.org/pdf/2012.08019, SIAM Rewiew, 2021.

[2] Nurcan Durak, Ali Pinar, Tamara G. Kolda, and C. Seshadhri, *Degree Relations of Triangles in Real-world Networks and Graph Models*, https://arxiv.org/abs/1207.7125, Proceedings of the 21st ACM international conference on Information and knowledge management, 2012.

[3] Maciej Besta, Raphael Grob, Cesare Miglioli, Nicola Bernold, Grzegorz Kwasniewski, Gabriel Gjini, Raghavendra Kanakagiri, Saleh Ashkboos, Lukas Gianinazzi, Nikoli Dryden, Torsten Hoefler, *Motif Prediction with Graph Neural Networks*, https://arxiv.org/pdf/2106.00761, 2021.

[4] Elena Czeizler, Tommi Hirvola, Kalle Karhu, *A Graph-Theoretical Approach for Motif Discovery in Protein Sequences*, https://ieeexplore.ieee.org/document/7364223, IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2017.

[5] Aravind Sankar, Xinyang Zhang, Kevin Chen-Chuan Chang, *Motif-based Convolutional Neural Network on Graphs*, https://arxiv.org/abs/1711.05697, 2017.

[6] Anthony Baptista, Rubén J. Sánchez-García, Anaïs Baudot, Ginestra Bianconi, *Zoo Guide to Network Embedding*, https://arxiv.org/pdf/2305.03474, Journal of Physics: Complexity, 2023.

[7] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, Wenwu Zhu, *Asymmetric Transitivity Preserving Graph Embedding*, https://www.kdd.org/kdd2016/papers/files/rfp0184-ouA.pdf, Proceedings of the 22nd ACM international conference on Information and knowledge management, 2016.

[8] Daixin Wang, Peng Cui, Wenwu Zhu, *Structural Deep Network Embedding*, https://arxiv.org/pdf/2305.03474, Proceedings of the 22nd ACM international conference on Information and knowledge management, 2016.

[9] Zexi Huang, Arlei Silva, Ambuj Singh, *A Broader Picture of Random-walk Based Graph Embedding*,
https://dl.acm.org/doi/pdf/10.1145/3447548.3467300, Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, 2021.

[10] Palash Goyal, Emilio Ferrara, *Graph Embedding Techniques, Applications, and Performance: A Survey*,
https://arxiv.org/abs/1705.02801, Knowledge-Based Systems, 2018.

[11] Palash Goyal, Di Huang, Sujit Rokka Chhetri, Arquimedes Canedo, Jaya Shree, Evan Patterson, *Graph Representation Ensemble Learning*,
https://arxiv.org/pdf/1909.02811, 2020 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), 2020.

[12] William L. Hamilton, Rex Ying, Jure Leskovec, *Representation Learning on Graphs: Methods and Applications*,
https://arxiv.org/pdf/1709.05584, 2018.

[13] Keyulu Xu, Weihua Hu, Jure Leskovec, Stefanie Jegelka, *How Powerful are Graph Neural Networks?*,
https://arxiv.org/pdf/1810.00826v3, 2019.

[14] Mikhail Belkin, Partha Niyogi, *Laplacian Eigenmaps for Dimensionality Reduction and Data Representation*,
https://ieeexplore.ieee.org/abstract/document/6789755, Neural computation, 2003.

[15] Aditya Grover, Jure Leskovec, *node2vec: Scalable Feature Learning for Networks*,
https://arxiv.org/abs/1607.00653v1, Proceedings of the 22nd ACM international conference on Information and knowledge management, 2016.

[16] Bryan Perozzi, Rami Al-Rfou, Steven Skiena, *DeepWalk: Online Learning of Social Representations*,
https://arxiv.org/abs/1403.6652v2, Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, 2014.

[17] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, Emmanuel Müller, *VERSE: Versatile Graph Embeddings from Similarity Measures*,
https://arxiv.org/abs/1803.04742v1, Proceedings of the 2018 world wide web conference, 2018.

[18] Thomas N. Kipf, Max Welling, *Variational Graph Auto-Encoders*,
https://arxiv.org/abs/1611.07308v1, 2016.

[19] Shaosheng Cao, Wei Lu, Qiongkai Xu, *GraRep: Learning Graph Representations with Global Structural Information*, https://dl.acm.org/doi/10.1145/2806416.2806512, Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, 2015.

[20] William S. Cohen, R. E. Schapire, and Y. Singer, *Learning to Order Things*, 2003.

[21] B. Weisfeiler and A. Leman, *The reduction of a graph to canonical form and the algebra which appears therein*, Nauchno-Technicheskaya Informatsia, Seriya 2, no. 9, pp. 12-16, 1968.

[22] G. Namata, B. London, L. Getoor, B. Huang, and U. Dasgupta, *Query-driven active surveying for collective classification*, In 10th International Workshop on Mining and Learning with Graphs, 2012.

[23] S. Sen, A. Gallagher, M. Ali, and T. Eliassi-Rad, *Collective Classification in Network Data*, AI Magazine, 29(3):93, 2008.

[24] M. Zitnik and J. Leskovec, *Predicting Multicellular Function through Multi-layer Tissue Networks*, Bioinformatics, 33(14):190-198, 2017.

[25] M. Merz, B. Roessler, R. Yi, and A. Bojchevski, *Wiki-CS: A Novel Dataset for Graph Neural Networks*, https://github.com/pmernyei/wiki-cs-dataset, 2019.

[26] Jure Leskovec, A. Rajan, and L. Danai, *Open Graph Benchmark: A Large-Scale Benchmark for Graph Neural Networks*, https://ogb.stanford.edu/docs/nodeprop/, 2020.

[27] Shaosheng Cao, Wei Lu, Qiongkai Xu, *Deep Neural Networks for Learning Graph Representations*, https://ojs-aaai-org.translate.goog/index.php/AAAI/article/view/10179?_x_tr_sl=en&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=sc, Proceedings of the AAAI conference on artificial intelligence, 2016.

[28] Hongyun Cai, Vincent W. Zheng, Kevin Chen-Chuan Chang, *A Comprehensive Survey of Graph Embedding: Problems, Techniques and Applications*, https://arxiv.org/abs/1709.07604, IEEE transactions on knowledge and data engineering, 2018.

[29] C. Bayan Bruss, Anish Khazane, Jonathan Rider, Richard Serpe, Saurabh Nagrecha, Keegan E. Hines, *Graph Embeddings at Scale*, https://arxiv.org/abs/1907.01705, 2019.

[30] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, Xavier Bresson, *Benchmarking Graph Neural Networks*, https://arxiv.org/abs/2003.00982, Journal of Machine Learning Research, 2023.

[31] Lei Cai, Jundong Li, Jie Wang, Shuiwang Ji, *Line Graph Neural Networks for Link Prediction*,
https://arxiv.org/pdf/2010.10046, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2021.

[32] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, Yoshua Bengio, *Graph Attention Networks*,
https://arxiv.org/abs/1710.10903, 2017.

[33] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, Maosong Sun, *Graph Neural Networks: A Review of Methods and Applications*,
https://arxiv.org/abs/1710.10903, 2018.

[34] Jun Jin Choong, Xin Liu and Tsuyoshi Murata, *Optimizing Variational Graph Autoencoder for Community Detection with Dual Optimization*,
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7516625/, 2020.

[35] Yu Xie, Maoguo Gong, Yuan Gao, A. K. Qin and Xiaolong Fan, *A Multi-Task Representation Learning Architecture for Enhanced Graph Classification*,
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6962136/, Frontiers in neuroscience, 2020.

[36] NetworkX Developers, *NetworkX Documentation: Triangle Algorithms*, https://networkx.org/documentation/stable/_modules/networkx/algorithms/cluster.html#triangles, 2024.

Acknowledgments

I would like to express my sincere gratitude to Prof. Vandin, my supervisor, for his availability and professionalism throughout these months of work. His guidance has been invaluable.

I would also like to extend my heartfelt thanks to my grandmother, Lilli, for always being willing to listen whenever I needed someone to talk to, and for her steadfast belief in my potential.

Finally, I would like to express my gratitude to my parents, Barbara and Franco, for their sacrifices and for always providing me with their unwavering support and affection.