Spring 5-21-2020

Higher-order Link Prediction Using Graph Embeddings

Neeraj Chavan
San Jose State University

Higher-order Link Prediction Using Graph Embeddings

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Neeraj Chavan

May 2020

The Designated Project Committee Approves the Project Titled

Higher-order Link Prediction Using Graph Embeddings

by

Neeraj Chavan

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2020

Dr. Katerina Potika	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. William Andreopoulos	Department of Computer Science

ABSTRACT

Higher-order Link Prediction Using Graph Embeddings

by Neeraj Chavan

Link prediction is an emerging field that predicts if two nodes in a network are likely to be connected or not in the near future. Networks model real-world systems using pairwise interactions of nodes. However, many of these interactions may involve more than two nodes or entities simultaneously. For example, social interactions often occur in groups of people, research collaborations are among more than two authors, and biological networks describe interactions of a group of proteins. An interaction that consists of more than two entities is called a higher-order structure. Predicting the occurrence of such higher-order structures helps us solve problems on various disciplines, such as social network analysis, drug combinations research, and news topic connections. Moreover, we can use our methods to get more knowledge about news topics during the COVID-19 pandemic.

Higher-order link prediction can be accomplished using neural networks and other machine learning techniques. The primary focus of this project is to explore representations of three-node interactions, called triangles (a special case of higher-order structure). We propose new methods to embed triangles: by generalizing node2vec algorithm using different operators to learn an embedding for a triangle, and by using 1-hop subgraphs of the triangles to learn embeddings using graph2vec algorithm and graph neural networks. The performance of these techniques is evaluated against the benchmark scores on various datasets used in the bibliography. From the results, it is observed that the node2vec based triangle embedding algorithm performs better or similar on most of the datasets compared to benchmark models.

ACKNOWLEDGMENTS

I want to take this opportunity to express my gratitude to my advisor, Dr. Katerina Potika. Her numerous valuable inputs steered my research in the right direction and have helped me throughout this project. Her constant encouragement and support have been instrumental to me. I would also like to thank my committee members, Dr. Christopher Pollett and Dr. William Andreopoulus, for their valuable inputs and feedback on my project.

A special thanks to the authors of the paper "Simplicial closure and higher-order link prediction" [1] A. R. Benson, R. Abebe, M. T. Schaub, A. Jadbabaie, and J. Kleinberg for their datasets and insights, which significantly contributed to my research.

Lastly, this acknowledgment will be incomplete without mentioning my family and friends who stood by me like a rock throughout this process. I cannot thank you enough for your motivation and support, without which this would not have been possible.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Link prediction is an emerging field that shows if two nodes in a graph are likely to be connected or not. Link prediction needs to go beyond the pairwise associations and predict relationships among more than two nodes. However, much of the information in the graphs contain information between more than two nodes [1]. We can take some common examples such as, communication within a group of people, chemical reactions involving more than two chemicals, or a collaboration between multiple researchers. These kinds of interactions are omnipresent but have received little attention. Therefore, there is an increasing need for predicting these structures. This type of link prediction will help with applications such as involvement of multiple chemicals in reactions [2], predicting new types of drugs [1], suggesting groups in social media [3], a collaboration between researchers [1], and biological interactions between sets of molecules [3].

1.1 Pairwise Link Prediction

Given two nodes in a network, the problem of identifying if these two nodes will connect shortly is called as link prediction [4]. These interactions, if performed between any two nodes, is known as pairwise link prediction. The network models represent the relationships of the underlying system as nodes and to use links in the network to capture pairwise relationships. These pairwise interactions have applications in representing friendships between pairs of people in a social network, a research collaboration between pairs of researchers, and product recommendation in e-commerce [5].

1.2 Higher-order Link Prediction

Link prediction is a problem with increasing significance in network sciences that applies to many disciplines. Link prediction mainly captures the relationships between any two nodes. This pairwise link prediction ignores the fact that there are higher-order relationships and interactions involved in the formation of that graph or network [6]. This project focuses on the issue of extending link prediction even for higher dimensions, called higher-order link prediction [1]. The topology of the graph does not capture these higher-order structures. Hence, most of the time, these interactions are lost right at the data collection stage, where data is collected directly in the graph format. Higher-order structures can be modeled using different ways which include simplicial complexes [7], set systems [8], hypergraphs [9], and bipartite affiliation graphs [10]. In this project, we use simplicial complexes for modeling of higher-order structures.

1.3 Applications

Higher-order or group-based interactions are ubiquitous in networks. Even traditional network analysis datasets have these interactions [1]. For example, coauthorship networks often involve more than two people writing a paper together. In this example, if we represent the data topologically, group interactions between people are missed. Similarly, in email networks where messages have multiple recipients, higher-order interactions can provide more meaning to the relationships rather than just focusing on the pairwise interactions. Furthermore, this strategy can be applied to the activity of neurons in the human brain, multiple actors appearing in a film, drug networks involving multiple drugs, and group chats in social networks. Despite the importance of higher-order interactions in graphs, there is limited information

about higher-order link prediction in real-world datasets.

1.4 Challenges in applying Embedding techniques

A complex network of entities and relationships make the graph a robust and informative data structure. Analysis of graphs can be done to extract useful structural and functional properties of nodes and their interactions. In the recent past, Machine learning (ML) and deep learning techniques have been applied for graph analysis. These techniques require the data to be in the euclidean form (n-dimensional vectors) as feature vectors. The performance of the above models depends on the quality of these feature vectors.

The task of generating feature vectors is difficult because of the structure of graphs and has been a field of importance. This field is known as representation learning. This generation of feature vectors from graphs is known as graph embeddings. One such solution is to employ the ML techniques for representation learning. Much research has been carried out in this domain to learn the representation of the graph's nodes, and their interactions.

There are techniques like node2vec [11], graph2vec [12], Deepwalk [13], which are used for representing nodes and graphs as embeddings. These embeddings are generated for a single node or a graph. However, there is no direct mechanism that allows representing a higher-order structure as embedding. To fill this gap, we adapt the existing techniques of node2vec and graph2vec to generate embeddings for triangles in graphs. This method can be extended to other higher-order structures in the future.

1.5 Motivation and Problem Statement

Graphs are complex and dynamic objects which model relationships using nodes and vertices. The relationships between graphs frequently involve higher-order interactions, which provide a rich source of information for analyzing these graphs. One such task to analyze the graph is link prediction. In the same vein, higher-order link prediction is important to consider these interactions and predict the evolution of graphs. Motivated by the importance of higher-order structure properties and their importance in the evolution of the graph, higher-order link prediction is studied. The objective of this research is to employ graph embedding for the task of higher-order link prediction and analyze the performance on standard graph datasets. For the scope of this project, higher-order link prediction is restricted to the simplicial closure of triangles. This project will involve implementing and adapting graph embedding techniques like node2vec [11], graph2vec [12] and graph neural networks [14] for link prediction. The performance of these algorithms will be compared with benchmark results [1] performed for higher-order link prediction.

This report is organized into four chapters. The second chapter describes the necessary terminologies for this project. The third chapter discusses the graph embedding techniques and higher-order link prediction techniques. It discusses the related work useful for higher-order link prediction. The fourth chapter explains the methodology and algorithms used for this project. The fifth chapter describes the datasets and evaluation metrics used. The sixth chapter discusses the experiments and results. The last chapter gives the conclusion.

CHAPTER 2

Terminology

In this chapter, the necessary terminologies used throughout this project are defined and discussed.

- **Graph**: A graph is a network, containing a set of vertices or nodes and edges connecting these vertices. An entity is analogous to a node, and the relationship is analogous to an edge. Figure 1 gives an example of a graph with eight nodes. Circles in the graph represent the nodes and the lines connecting these circles represent edges.

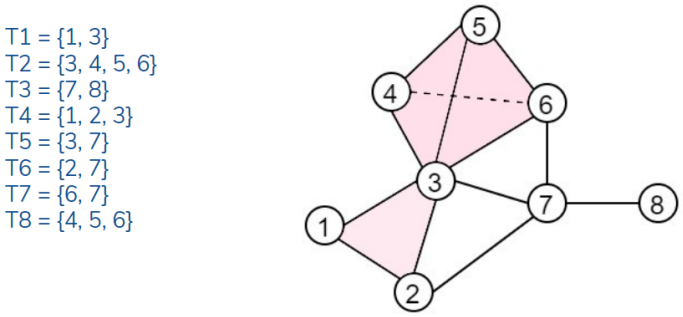


Figure 1: Example of a Graph

- **Weighted and Unweighted Graphs**: If the graph has weights specified for the edges, then it is called as weighted graphs, else it is called as an unweighted graph. The graph is shown in Figure 1 is an unweighted graph.

- **Directed and Undirected Graphs**: If the graph has directions specified on its edges, then it is a directed graph, else it is an undirected graph. The graph is shown in Figure 1 is an undirected graph.

- **Temporal Graph**: If the graph changes according to time, then it is known as a temporal graph. This is also known as a time-varying network. These graphs convey the information about the evolution of the network. For example, the graph in Figure 1 is a temporal graph. The instances T1, T2, etc. stated in the fig show the evolution of the graph according to time.

- **Link Prediction**: Link prediction can be best described as the process of predicting links between two nodes. This prediction is based on the structural properties and existing link interactions in the graph. For example, in Figure 2 there is no link between nodes B and C at time T, but will there be a link between those two nodes at some time $T + 1$ in the near future is called link prediction.

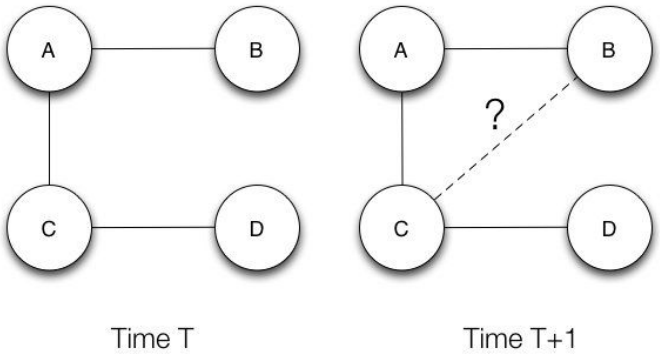


Figure 2: Link prediction in a graph

- **Higher-order Structure in Graph**: Higher-order structure is a structure having interaction between more than two nodes simultaneously. For example, in Figure 1 instance T4 involves three nodes interacting at once. Similarly, the instances T2, T4, and T8 in Figure 1 depicts higher-order structures.

- **Simplex**: A simplex is a way to represent triangles, tetrahedrons in terms of dimensions. In the context of graphs, a simplex is defined as any finite set of

nodes. In Figure 1, each timestamp represents a simplex. For example,

  - a 0-simplex is a point.
  - a 1-simplex is a line.
  - a 2-simplex is a triangle.
  - a 3-simplex is a tetrahedron.

- **Simplicial Closure**: If there is an open (i.e., all nodes have not appeared simultaneously) k-clique in the graph, then the appearance of a new simplex containing these k nodes is called as a simplicial closure instance. It is also a transformation of an open structure to a closed one.

- **Open Triangle**: In the observed graph, if there have been interactions only between pairs of nodes forming the triangle, but all the three nodes have not interacted simultaneously (i.e., appeared as a subset in one simplex), then it is called as an open triangle.

- **Closed Triangle**: Given an open triangle, if all the nodes involved in the triangle appear in a simplex by itself or as a subset, then the triangle undergoes closure. This triangle is called a closed triangle.

- **Embedding**: Embedding means converting data to a vector representation of features where the properties of this data can be represented by distance. For example, a word embedding creates an embedding for the word using euclidean distance. These embeddings are similar if the words are similar.

- **Graph embedding**: If the embedding is used to model a graph or subgraph, it is called graph embedding.

- **Node embedding**: If the embedding is used to model a node, it is called node embedding.

CHAPTER 3

Related Work

This chapter discusses the current approaches used for computing graph embeddings and methods used for higher-order link prediction. It also studies and discusses the techniques used for extracting relationships between nodes and ways to represent the embeddings for higher-order structures.

3.1 Simplicial Closure

Higher-order interactions are often encountered in all types of datasets. In [1], the importance of these interactions in analyzing the graphs is studied. They study the organizational principles of higher-order structures in real-world datasets. This paper focuses on the higher-order structures which are not captured by the topology of the graph. Motivated by the importance of triangular structures and triadic closure, they study this via simplicial closure. They state a simplicial closure as an instance where a group of nodes evolves until they coappear in a higher-order structure. They propose a higher-order link prediction problem, which predicts this simplicial closure.

To represent these higher-order structures, they use simplicial complex [7]. A simplex is a term used to represent multiple nodes occurring at one instance in temporal graphs. For example, $t_1 : 1, 2, 3$ represents a three-node simplex. Figure 3A gives an example of a higher-order network dataset represented as simplices. The data studied in this paper is of this nature. Figure 3B shows the graph of the dataset without the timestamps. The shading in the figure depicts simplices and is used to mark the difference between traditional graphs. For example, nodes 1, 7, and 8 form a closed triangle as they appear together in the same simplex at t_5. However,

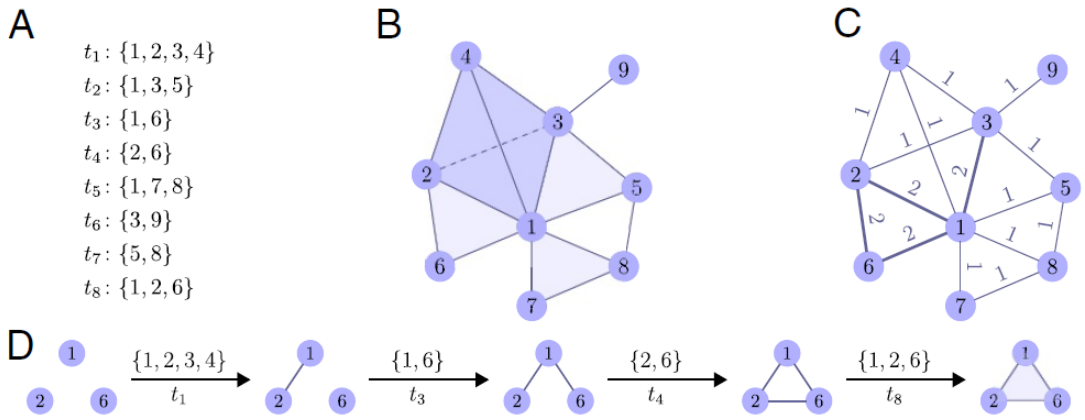


Figure 3: Higher-order network representation and simplicial closure example [1]

nodes 1, 5, and 8 form an open triangle as all three pairs of nodes coappeared in simplices at time t_2, t_5, t_7 respectively, but they do not coappear in a single simplex. This open triangle is shown in the figure by not shading the area. A projected graph with the weight of an edge representing the frequency of those two nodes coappearing in simplices is shown in Figure 3C. This graph ignores the higher-order structures, which are often used by traditional network science use cases. A simplicial closure event is shown as an example in Figure 3D. At the time t_4, the nodes form an open triangle, which eventually closes by a simplicial closure event occurring at time t_8. In [1] simplicial closures are studied, and they predict the occurrence of such closures using higher-order link prediction.

To evaluate the theory of the importance of higher-order structures in analyzing graphs, a higher-order link prediction problem is used. They restrict their link prediction to 3 nodes appearing together. This is a simplicial closure event on triples of nodes. The problem studied is of predicting which triples of nodes or open triangles that have not yet appeared in a simplex simultaneously will be a subset of some simplex in the future. They evaluate their algorithms on 19 different datasets.

These datasets comprise of Coauthorship networks, Drug Networks, US Congress data, Email Networks, and StackOverFlow posts. They use an 80/20 split of datasets. AUC-PR is used as a metric for prediction performance. Eight different models are compared for link prediction performance. No one model performs the best on all datasets. Analysis of the performance suggests that open triangles with strong ties are most likely for a closure. Also, generalized means of edge weights give a strong indication of closures of open triangles. They observe that higher-order link prediction is challenging because of the absolute performance achieved. They suggest a future application for embeddings in higher-order link prediction.

Higher-order link prediction gives a new dimension and breaks out of the phenomena of pairwise link prediction. The datasets are evaluated only using traditional models. There is a potential to employ graph embedding techniques to learn these higher-order structures and predict closures. In [1] it is suggested that structural features are useful in predicting higher-order links. We can make use of this to modify node2vec [11] and graph2vec [12] embedding techniques as they preserve the structural significance in their representations.

3.2 node2vec

An area of investigation in graph embedding focuses on node-level embedding. In [11], node2vec technique uses the local search method. This method is used to extract the neighboring node information and generating sequences from nodes. Depth-first search (DFS) and Breadth-first search (BFS) are mainly employed for exploring the local neighborhood. However, DFS traverses the nodes which are far away from the target node, which gives a high-level view of the network. DFS helps in preserving the homophily of the graph. On the other hand, BFS extracts the structural

equivalence as it only traverses the immediate neighborhood of the target node. The authors imply that these strategies extract and preserve only specific properties of the graph.

Considering the above reason as motivation, the authors introduce a new method called second-order random walks to explore neighborhoods. This method is similar to a random walk in [13] but has control to bias the behavior of walks and makes it more flexible in its strategy to explore the local neighborhood. To give this control, node2vec uses 2 additional parameters that are used to toggle exploration between BFS and DFS. One parameter is called the return parameter, denoted by p. The return parameter controls the frequency of visits to a node in the walk. And the other parameter is called the InOut parameter, denoted by q. The InOut parameter is used to switch the behavior between BFS and DFS.

where,

$q < 1$: more like BFS

$q > 1$: more like DFS

These two parameters combined are used to determine the next node to be explored in the walk, known as search bias α. The walk generates a sequence of length l, starting at some random node v. The probability of choosing the next node in the walk is given by the equation [11] as follows:

$$P\big(c_i = x | c_{i-1} = v\big) = \begin{cases} \pi_{vx}/Z & if (v,x) \in E \\ 0 & otherwise \end{cases}$$

where:

π_{vx}: unnormalized transition probability between v and x

Z: normalizing constant

There are problems with using normalized edge weight for transitions as in unweighted graphs all the neighboring nodes will have equal probability. To counter this problem, a search bias α is used. If node t precedes node v then the equation [11] for α is given by:

$$\alpha_{pq} = \begin{cases} 1/p & d_{tx} = 0 \\ 1 & d_{tx} = 1 \\ 1/q & d_{tx} = 2 \end{cases}$$

This random walk technique learns from diverse neighborhoods, which gives quality information of nodes. Once the corpus is generated, the skip-gram technique is used to obtain embeddings. The quality of embeddings generated preserves the structural as well as homophily properties of the graph. node2vec [11] justifies this claim by experimenting on the Les Miserables [15] dataset. This technique uses the embeddings generated for nodes to learn edge features. Figure 4 shows the different operations to learn edge features. node2vec has experimented on Facebook, Protein-Protein interactions (PPI), and arXiv ASTRO-PH datasets for link prediction. To generate the dataset, they remove 50% of randomly chosen edges such that the graph obtained after removal is connected. Similarly, they generate negative samples from the node pairs with no connecting edge, equal to the positive number of samples. AUC is used as a metric to evaluate the performance against other techniques like LINE, DeepWalk, and other heuristic techniques. The paper states that node2vec outperforms other feature learning techniques on all datasets. Also, all the binary

operators used with node2vec give equal or better performance than their peers.

Operator	Symbol	Definition
Average	$\boxplus$	$[f(u) \boxplus f(v)]_i = \frac{f_i(u)+f_i(v)}{2}$
Hadamard	$\boxdot$	$[f(u) \boxdot f(v)]_i = f_i(u) * f_i(v)$
Weighted-L1	$\|\cdot\|_{\bar{1}}$	$\|f(u) \cdot f(v)\|_{\bar{1}i} = \|f_i(u) - f_i(v)\|$
Weighted-L2	$\|\cdot\|_{\bar{2}}$	$\|f(u) \cdot f(v)\|_{\bar{2}i} = \|f_i(u) - f_i(v)\|^2$

Figure 4: Binary operators to learn edge features [11]

The way in which node2vec is employed for pairwise link prediction, in the same way, we can use it for higher-order link prediction. The quality of generated embeddings improves the link prediction performance of node2vec. The neighborhood exploration strategy used by node2vec extracts and preserves the relationships amongst nodes in the graph. Using this strategy, we can use the average, hadamard, l1, and l2 operators to generate embeddings for higher-order structures. Another strategy of generating embeddings directly for the higher-order structure is discussed in the next section of graph2vec.

3.3 graph2vec

The graph2vec [12] algorithm, as the name suggests, generates vectors for graphs. In contrast, node2vec [11] generates vectors for individual nodes. Graph structure representation learning is an upcoming research topic. But there has been little research in representing entire graphs as a single feature vector. Until now, graph kernels such as random walks, shortest paths, graphlets, etc. have been used to cater to entire graph analytics.

To overcome the above problem, [12] proposes a new neural embedding framework named graph2vec to learn representations of arbitrarily sized graphs. The em-

beddings are generated using an unsupervised learning approach and are task agnostic. This gives the liberty to use it for any task such as graph classification, link prediction, graph clustering.

Graph kernels have two important limitations, which make it all the more necessary to have graph2vec embeddings. Firstly, do not generate embeddings as feature vectors, which makes graph data unusable with ML techniques and neural networks for modeling. Secondly, the kernels use substructures that are handcrafted. This makes it difficult to generalize over all types of datasets. Learning substructure embeddings such as node, path, or subgraph cannot learn representation for entire graphs. Obtaining embedding for a graph through extensions like averaging or max-pooling over substructure embeddings gives sub-optimal results.

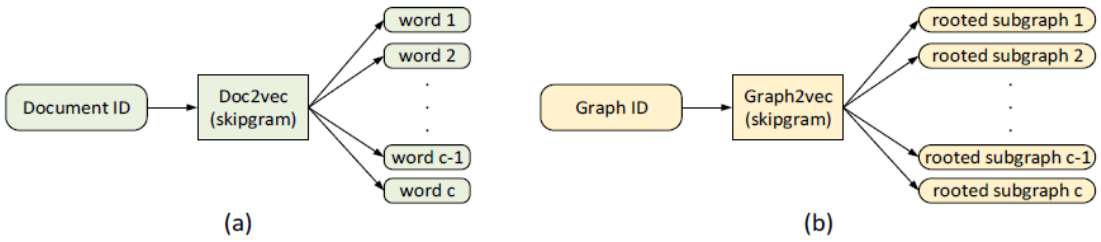


Figure 5: Shows the analogous behavior of graph2vec to that of doc2vec. [12]

In [12], it is stated that the graph2vec approach for learning the entire graph's representation which is inspired by the document embedding technique [16]. Doc2vec technique is extended to learn graph embeddings. An entire graph is viewed as a document and subgraphs around every node as words that make the document. Doc2vec uses a skip-gram model to learn the representation of the graph. The Skip-gram model provides a similar representation for words appearing in a similar context. This makes the learned embeddings using the skip-gram model preserve semantics of the document. The analogous nature of doc2vec and graph2vec is shown in Figure 5. For

example, doc2vec's skip-gram model samples c words from document d and considers these words as co-occurring in the same context to learn d's embedding. Similarly, graph2vec skip-gram model samples c rooted subgraphs around nodes occurring in the graph and then treat them as co-occurring words to learn graph's embedding.

The rooted subgraphs used as words in the skip-gram model are important. Compared to other substructures such as nodes, rooted subgraphs cover higher-order local neighborhoods. These higher-order features enable a rich representation of the structure and relationships in the graph. Hence, the composition of graphs is represented in a better way by these embeddings. Moreover, these subgraphs capture the non-linear features in the graphs better in comparison to linear features like paths or walks. The expectation of the learned embeddings is that structurally similar graphs will have similar embeddings.

The performance of the algorithm is evaluated on five benchmark datasets for graph classification. The benchmark datasets used are MUTAG, PTC, PROTEINS, NCI1, and NCI109. These datasets belong to the chemo-informatics and bio-informatics domain. They use a 90/10 split for training and testing data and use an SVM classifier for classification. Accuracy is used as the evaluation metric, and for efficiency, the time required to generate embeddings is considered. The graph2vec algorithm performs better or has comparable accuracy with respect to other techniques (node2vec, sub2vec, WL kernel). The performance of graph2vec is attributed to the data-driven and structure-preserving nature of embedding, which learns local and global similarities in graphs. This algorithm outperforms other methods in MUTAG, PTC, and PROTEINS datasets in particular.

To summarize, graph2vec is an innovative approach to learn the entire representation of a graph. As graph2vec is data-driven, it will perform better on large

datasets. This technique can be employed to work for higher-order link prediction by giving it each open triangle [1] enclosed subgraph as input. An enclosed subgraph around an open triangle is nothing but a h-hop subgraph rooted at the nodes involved in the triangle. Then graph2vec will generate representations for each of the open triangle. Then it can be used to classify if it is positive or negative for the existence of a simplicial closure event. In this way, we can leverage graph2vec for higher-order link prediction.

3.4 SEAL

Similar to graph2vec [14], SEAL can learn a representation for a subgraph. [14] uses the subgraph to interpolate and learn the graph structure features (used by traditional heuristic scores). Link prediction traditionally uses heuristic scores like PageRank [17] and Preferential Attachment [18] which fail on certain datasets. Due to this reason, learning graph structure features instead of using the heuristic scores is more viable. To learn these graph structure features, authors make the use of Graph Neural Network (GNN).

To solve this problem, graph structure features learned from local subgraphs using GNN can be used [14]. SEAL framework states that heuristic methods have strong assumptions while predicting links. These assumptions have the drawback of not working on certain types of data. Heuristic methods belong to a class of graph structure features that involve relationships observed between edges and nodes in the graph. These features can be learned automatically from the network. Learning features from the graph was given by Weisfeiler-Lehman Neural Machine [5], where they use a local enclosing subgraph around the target nodes and then use it as training data on a neural network for prediction.

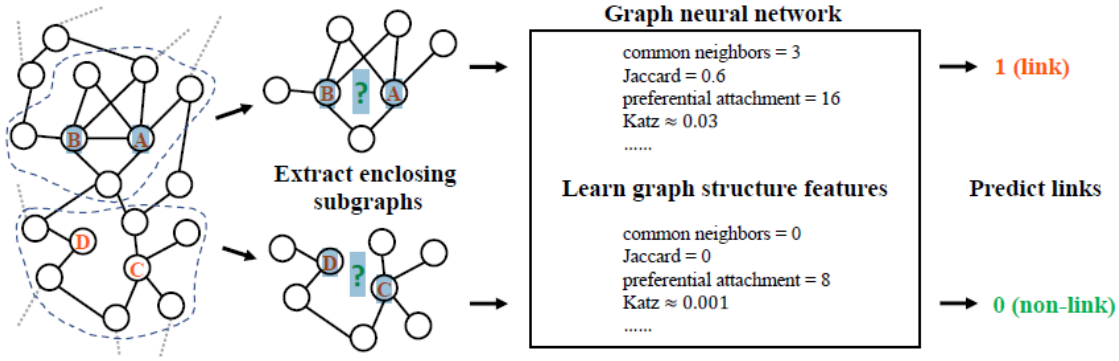


Figure 6: Example of local enclosing subgraph for learning graph structure features [14]

Figure 6 illustrates the method used to extract subgraphs. The enclosing subgraph can be extracted till a h hops from the target nodes, in this case, nodes (A, B) and (C, D). These enclosing subgraphs are 1-hop subgraphs. Similarly, 2-hop, 3-hop, and h-hop enclosing subgraphs can be extracted as per need. GNN makes use of subgraphs to learn graph structure features, as proposed in SEAL.

The subgraphs extracted are a rich source of information as all lower hop heuristics can be calculated from this. However, high-order heuristics such as Katz and PageRank need a large hop number h. This subgraph is as good as the entire network, which makes it unfeasible due to memory and time constraints. To overcome this [14] introduce a $\gamma - decaying\ theory$, which unifies most of the high-order heuristics. From an h-hop extracted subgraph, an approximation of γ-decaying heuristic can be made. Hop count h leads to approximation error reducing exponentially [14]. From this theory, it can be concluded that heuristics like Katz and PageRank can be learned from small enclosing subgraphs with some approximation error. SEAL [14] proves that the $\gamma - decaying\ theory$ applies to Katz Index, PageRank, and SimRank methods. [19] and [20] empirically validate the theory of approximating PageRank

and SimRank, respectively, from local methods. The reason for this exponentially smaller error can be attributed to the fact that remote nodes in the graph with respect to the target nodes are of little help to the existence of links. Whereas, nodes closer to target have more information about possible links in the future.

SEAL learns graph structure features for link prediction. The steps involved in prediction are as follows:

1 : *enclosing subgraph extraction*

2 : *node information matrix construction*

3 : *learning the subgraph using GNN*

The node information matrix is constructed by using node labeling. Double-radius Node Labeling (DRNL) technique is used for preserving information about which node is a target node. This labeling technique allows the GNN to differentiate between target nodes and neighboring nodes and helps in predicting the link existence. In addition to just features learned by GNN, there is the capability to concatenate latent features (graph embeddings) and explicit features in the SEAL framework. These features can be added to each row in the node information matrix with respect to the targets. This makes the SEAL framework robust and improves link prediction performance by combining all types of features in one algorithm. The algorithm is evaluated on eight different datasets. The datasets include USAir, Yeast, Power, Router, E.coli, C.ele, NS, and PB. 90% of the data is used as positive training data, and 10% is used as testing data. Similarly, non-existent links are generated equal to the number of training and testing data as negative samples. Experiments are performed using AUC and average precision as evaluation metrics.

Hop count for enclosing subgraphs is an important hyperparameter. The ex-

periments consider only 1-hop or 2-hop subgraphs. This limit of hop count is due to the empirical verification that the performance customarily does not increase after a hop count of 3 or more. The hop count is decided between 1 and 2 based on the performance of datasets on CN and Adamic-Adar heuristic, respectively. SEAL performs better than other heuristic methods like CN, Jacquard, Preferential Attachment, Adamic-Adar, Katz, and three more heuristics. This reveals that learned features are better at extracting relationships than the manually designed heuristic scores.

To summarize, small enclosing subgraphs which are extracted around the target nodes can calculate low-order heuristics accurately and also interpolate many high-order heuristics with small approximation errors. Therefore, local subgraphs contain rich information about graph structure features for predicting links. GNN performs better in graph feature learning ability in comparison to fully-connected neural networks and graph kernels [14]. This ability of GNN for link prediction is also validated in [21] and [22]. In addition to this, the graph structural features can be combined with latent features (graph embeddings) and explicit features.

The method of enclosing a subgraph can be used for higher-order link prediction to learn the graph structure features. Moreover, we can also combine graph structure features with embeddings from node2vec or graph2vec to make the prediction performance more robust. As stated by [1], structural information is important to indicate higher-order links. This property can be used to apply the local enclosing subgraph technique to learn structural graph features for higher-order link prediction.

CHAPTER 4

Methodology

4.1 Problem Definition

Given a timestamped simplices of a graph, $G = \{S_i, t_i\}$ where i belongs to the number of observed simplices, t_i represents the time at which S_i was observed. $S_i = \{n_1, n_2, ..n_j\}_i$, where S_i is a set representation of all nodes n_j interacting at i^{th} simplex. This representation gives a temporal network with higher-order interactions captured in it. Consider, $|S_i| = k$ then we can say that S_i is a k-node simplex. This can also be called as a k-clique. The process of predicting occurrence of more than two nodes simultaneously can be best described as the problem of higher-order link prediction [1]. For this project, we narrow it down to predicting the occurrence of *three nodes* simultaneously. This is referred to as *open triangle* when the event of all three nodes appearing simultaneously as a subset in a simplex has not happened. But, when they do appear, it is referred to as *closed triangle*. An example of this is shown in Figure 8. This problem can be split into three phases - enumerating all open triangles and closed triangles in training and testing dataset, representing triangles as embeddings, triangle closure prediction.

4.2 Implementation Workflow

The implementation of the project is divided into different modules. Figure 7 illustrates the workflow of the whole project. The steps in the workflow are as follows:

1. The first module takes in raw graph data as input and returns a list of timestamped simplices containing nodes.

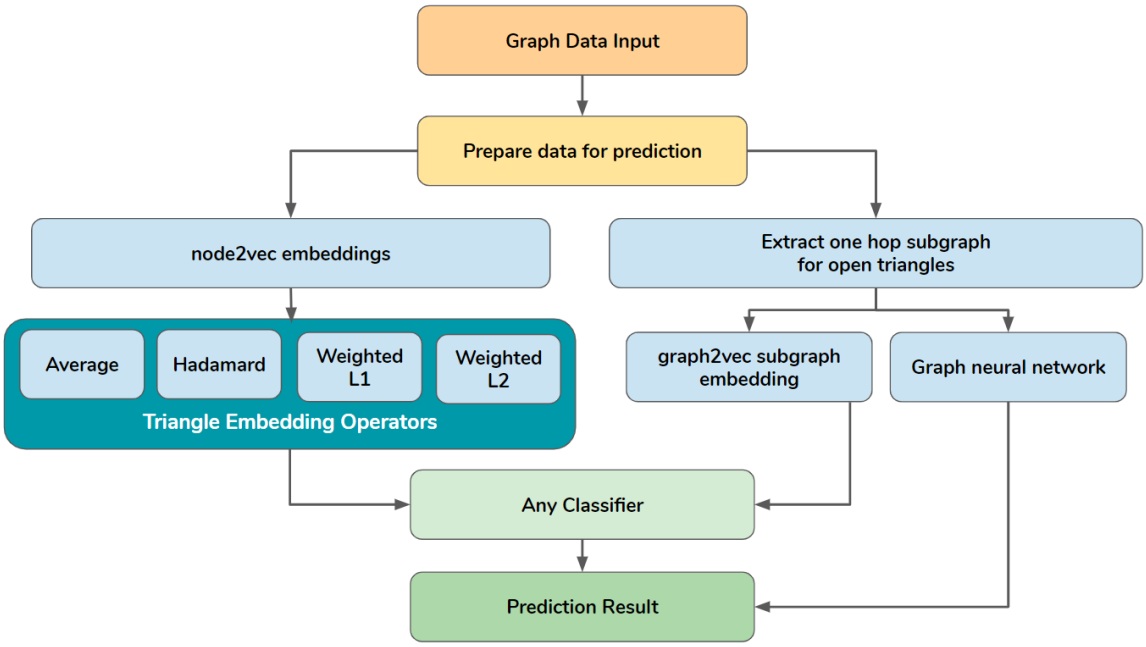


Figure 7: Workflow of algorithms to perform higher-order link prediction task

2. In the next module, this data is divided into training and testing split. Then a labeled dataset of open triangles is created based on this data (refer section 5.2).

3. Once we have the prediction data ready, node2vec embeddings are generated for the data. From these node embeddings, we learn the triangle embeddings using different operators specified in the figure. Using these learned triangle embeddings, any binary classifier can be trained, and the prediction result is returned.

4. Another module extracts 1-hop subgraphs for each of the open triangles for use in graph2vec and graph neural network algorithms.

5. Extracted subgraphs are passed onto graph2vec to learn embeddings for each subgraph. These embeddings are then used as an input to the binary classifier, which will predict if the open triangle undergoes closure or not.

6. Similar to the previous module, graph neural network receives extracted subgraphs, and we add some additional to the subgraph and pass it onto the graph neural network model for prediction. The model directly gives us the result.

The above explanation gives a brief overview of the implementation and algorithms used in the project. In the sections that follow, all the above algorithms are discussed in greater detail.

4.3 Enumerating Labelled Open Triangles

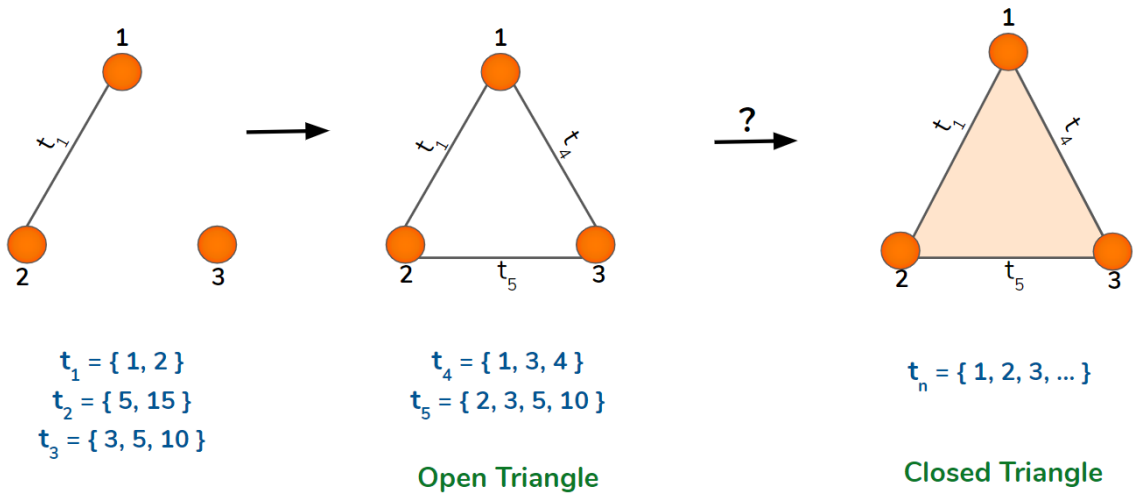


Figure 8: A sample lifecycle of triangle closure.

For the task of triangle closure prediction, embeddings of different types will be applied to represent an open triangle as an embedding. To begin with, let us take a look at how triangles can undergo closure. In Figure 8, t_1 shows the observed graph at that point. After t_4, t_5, the triangle converts into an open triangle (i.e., nodes have interacted in pairs at some point in time, but all nodes have not interacted simultaneously at a given timestamp t. The problem is to predict that at some point in the future i.e., t_n, this open triangle will undergo a closure or not). The triangle

closure is represented by the shading of that triangle. The ellipsis in t_n depicts that triangle closure will happen even if the three nodes appear in a bigger simplex as subsets.

Algorithm 1: Enumerating open triangles

```
1  function open_triangles(G, S):
   Input  : G: networkx graph representation of S,
            S: Vector representation of set of nodes for a data slice based on
            timestamp
   Output: open_triangles: vector representation of triangles that are still
            open in this data slice
2  // Set of triangles already gone through closure in the given simplices
3  closed_triangles ← get_closed_triangles(S)
4  open_triangles ← set()
5  // Run the loop in parallel
6  for each edge(u, w) in G do
7      // iterate over all graph nodes
8      for each vertex(v) in G do
9          if edge(u,v) in G and edge(v,w) in G then
10             if tuple(u,v,w) not in closed_triangles then
11                 open_triangles.add((u,v,w))
12             end
13         end
14     end
15 end
16 open_triangles ← list(open_triangles)
17 return open_triangles
```

Now, the next challenge is to prepare the data for prediction. For prediction, we have to split the dataset in training and testing, which is explained in section 5.2. In preparing the data, there are two essential algorithms to consider, enumerating open triangles and the closure of these open triangles. For example, if we consider the training data, the simplices are divided into the first 60 percent (S_{old}) and 60-80 percent (S_{new}) based on timestamps. The next step is to enumerate all the open triangles in S_{old}. Then all the new triangle closures are enumerated in S_{new}. Now to

create a labeled dataset, all the open triangles in S_{old} that undergo closure in S_{new} are labeled as positive(1) and others as negative(0). Similar steps are repeated for testing data, where S_{old} consists of 0-80 percent simplices, and S_{new} consists of 80-100 percent simplices.

Algorithm 1 explains how open triangles are enumerated. In the first step, we get all the triangles that have undergone closure in the data slice. The next step is to iterate over each edge (u, w) and then for each edge look for a node v, which has links with both u and w. This gives us a triangle. The triangle is then checked for closure, and if it has not yet closed, then the triangle is added to the list of open triangles. The process of enumerating open triangles is expensive for large datasets as the algorithm is $O(e * v)$, where e is the number of edges, and v is the number of vertices. To make the process faster, the for loop can make use of parallelism.

Algorithm 2 explains the process of enumerating newly closed triangles in the S_{new} simplices. It is then used to look up the open triangles which undergo closure to create a labeled dataset. The algorithm takes in S_{old} simplices, S_{new} simplices, and graph G built over S_{old}. The first step is to get closed triangles from S_{old}. Then, the algorithm iterates over all combinations of *nodes* of length 3 for each *simplex* in S_{new}. Next, it checks if all *nodes* from the combination $\in G|V|$ and that the combination has not undergone closure already. If these conditions are satisfied, the 3 node combination is added to a set of $new\_triangles$.

4.4 Algorithms for Higher-order Link Prediction

A higher-order link prediction task has been researched with standard features. In this project, the method of embeddings is leveraged to represent each triangle as an embedding, either by using node embedding or graph embedding. Three methods

Algorithm 2: Enumerating new triangle closures

```
function new_closures(G, S_old, S_new):
  Input  : G: networkx graph representation of S_old,
           S_old: Vector representation of set of nodes for old data slice based
           on timestamp,
           S_new: Vector representation of set of nodes for new data slice
           based on timestamp
  Output: new_triangles: set representation of triangles that have closed in
           this data slice
// Set of triangles already gone through closure in old simplices
closed_triangles ← get_closed_triangles(S_old)
new_triangles ← set()
for nodes in S_new do
    // iterate over all combinations of nodes of length 3
    for (i, j, k) in combinations(nodes, 3) do
        // skip if node has not yet appeared in the old simplices
        if i and j and k in G|v| then
            if (i, j, k) not in closed_triangles then
                new_triangles.add((i, j, k))
            end
        end
    end
end
return new_triangles
```

for embeddings representation are discussed in the subsections that follow.

4.4.1 Triangle Embedding using node2vec

node2vec [11] is an algorithm which learns structural features of the graph and creates an embedding for the nodes in vector space based on their structural similarity. These embeddings generated for a node can be converted into an embedding for the open triangle using four different operators, which are *Hadamard, Average, WeightedL1, and WeightedL2*. Figure 9 illustrates the steps in which an embedding is generated for an open triangle using node2vec. In the first step, an open triangle is

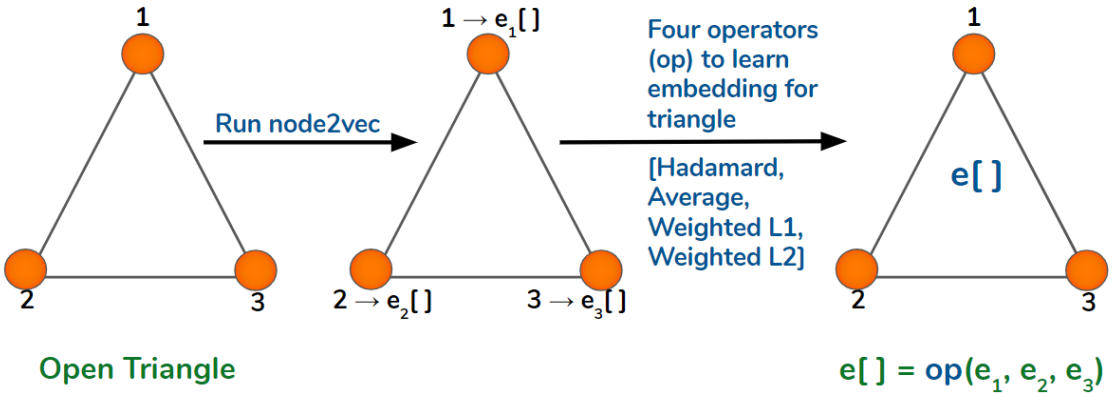


Figure 9: Workflow for generating node2vec embeddings

represented, which is to be classified. Then the node2vec algorithm is executed on the training graph, and output with node embeddings for each node is generated as shown in the second step of the figure. The last step is to combine the node embeddings using the specified operator and generate an embedding for the open triangle.

Algorithm 3 explains the procedure of representing triangles as embeddings using node2vec. A visual depiction of the algorithm is given in Figure 9. In the first step, the node2vec algorithm is run on the graph G to generate embeddings for each node. Parameters like the dimension of vector, length of the random walk, and the number of random walks can be passed to the algorithm for experimenting with different configurations. The next step is to iterate over the triangles and learn an embedding for each triangle. Table 1 shows the operators to learn the triangle features. These operators are adapted from binary in [11], to ternary operators for triangle embedding. Weighted L1 and L2 work with edges in the triangle, and then the embeddings are averages for each edge. In this way, a triangle embedding is learned from the node embeddings of node2vec.

Algorithm 3: Triangle node2vec Embedding

```
function triangle_node2vec(G, triangles, d, l, n, op)
  Input  : G: Networkx graph representation of training data,
           triangles: vector of open triangles,
           d: dimension of embeddings to be generated,
           l: length of random walks performed by node2vec,
           n: number of random walks,
           op: type of operator to combine node embeddings
  Output: E: vector of embeddings for triangles
model ← node2vec(G, d, l, n, p, q)
model ← node2vec.fit()
node_embeddings = model.vectors()
E ← []
for node set(u,v,w) in triangles do do
    u_vec ← node_embeddings(u)
    v_vec ← node_embeddings(v)
    w_vec ← node_embeddings(w)
    if op == "average": then
        E.append((u_vec + v_vec +w_vec)/3)
    else if op == "hadamard": then
        E.append(u_vec * v_vec * w_vec)
    else if op == "l1": then
        E.append((abs(u_vec − v_vec) + abs(v_vec − w_vec) + abs(u_vec − w_vec))/3)
    else if op == "l2": then
        E.append((abs(u_vec − v_vec)^2 + abs(v_vec − w_vec)^2 + abs(u_vec − w_vec)^2)/3)
    end
end
return E
```

4.4.2 Triangle Embedding using graph2vec

graph2vec [12] is an embedding technique that learns features of subgraphs and generates an embedding for each subgraph. This method makes use of the Weisfeiler Lehman Machine [5] for feature extraction and Doc2vec [16] model for implementation. graph2vec can be adapted to learn triangle embeddings by extracting a 1-hop subgraph around the nodes in the open triangle. An example of 1-hop subgraph is given in Figure 10. By using this method, subgraphs for all the triangles can be

Table 1: Operators to learn triangle features from node embeddings

Operator	Definition
Average	$E_{average} = \frac{f_i(u) + f_i(v) + f_i(w)}{3}$
Hadamard	$E_{hadamard} = f_i(u) * f_i(v) * f_i(w)$
Weighted-L1-average	$E_{l1} = \frac{\|f_i(u) - f_i(v)\| + \|f_i(v) - f_i(w)\| + \|f_i(u) - f_i(w)\|}{3}$
Weighted-L2-average	$E_{l2} = \frac{\|f_i(u) - f_i(v)\|^2 + \|f_i(v) - f_i(w)\|^2 + \|f_i(u) - f_i(w)\|^2}{3}$

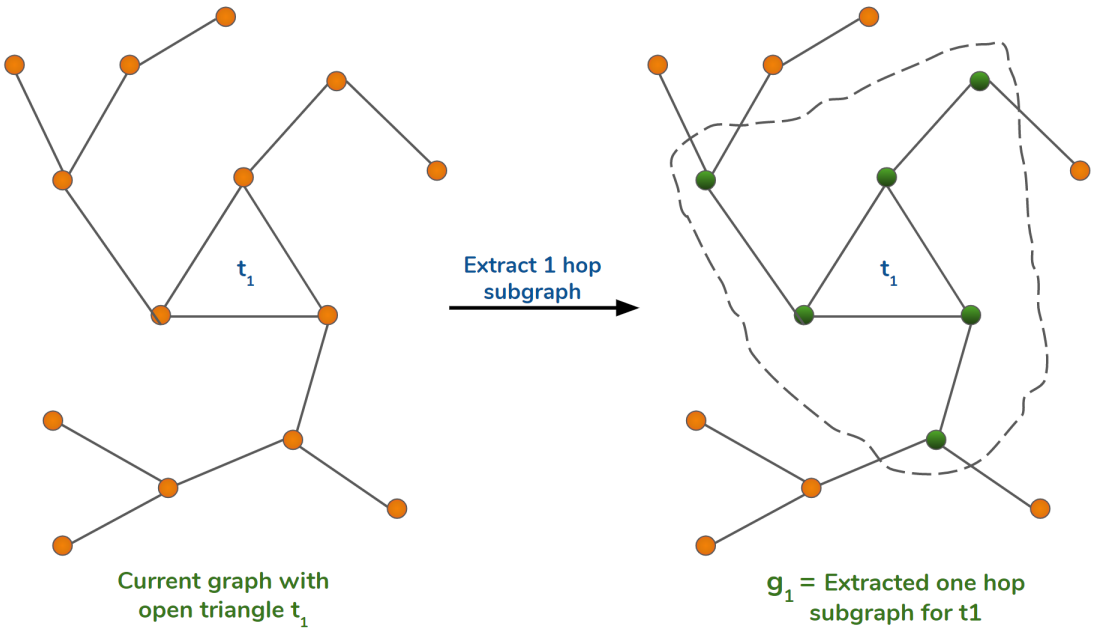


Figure 10: Subgraph extraction example

extracted, and an embedding can be generated for each of these subgraphs.

Algorithm 4 explains the procedure for extracting a 1-hop subgraph for the triangle. The first step is to iterate over the nodes in the triangle and for each node,

Algorithm 4: Extract one hop subgraph

```
function extract_subgraph(G, tri, max_nodes)
  Input  : G: observed graph based on simplices
           tri: vector representing 3 vertices in the triangle
           max_nodes: integer representing the maximum number of nodes
           to be included in the subgraph
  Output: G_s: networkx representation of subgraph
nodes ← set()
for each node(v) in tri do
    nodes.add(G.neighbors(v))
end
if max_nodes > 0 and max_nodes < length(nodes) then
    nodes = random.sample(nodes, max_nodes)
end
nodes.add(tri) // add triangle vertices
G_s = G.subgraph(nodes) //networkx function to get subgraph from nodes
return G_s
```

store their neighbors in a set. Now, all the 1-hop neighbors are extracted. The next step is to check if there is a limit on the maximum number of neighbor nodes that the subgraph can contain. The limit is applied by randomly sampling neighbor nodes equivalent to *max_nodes*. This limit is a vital feature to stop the subgraph from getting too big. For some graphs, the neighboring nodes can be in the order of hundreds. Hence it is crucial to limit the number of subgraphs. The final step is to return the networkx representation of the subgraph.

Figure 9 illustrates the graph2vec workflow for learning embeddings for the triangle. The procedure for learning triangle embeddings is given in Algorithm 5. The first step is to extract subgraphs for all the triangles. Then these subgraphs are passed to the graph2vec algorithm along with hyperparameters like dimensions, epochs, and learning rate. The output for this will be a list of embeddings learned by graph2vec for each triangle. These embeddings can then be passed to a classifier where we classify triangles as open or close. This algorithm is computation and memory intensive

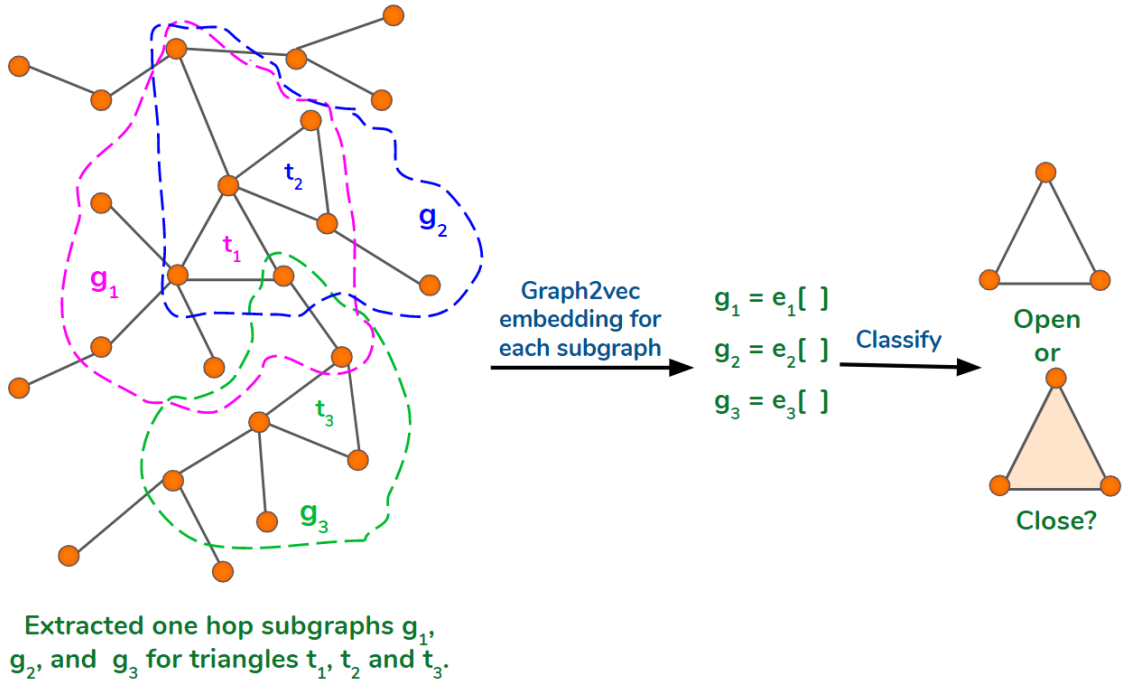


Figure 11: Example of Graph embeddings using graph2vec

Algorithm 5: Triangle graph2vec Embedding

```
function triangle_graph2vec(G, triangles, d, e, l)
  Input  : G: graph representation of simplices,
           triangles: Vector of triangles,
           d: dimension of embeddings to be generated,
           e: number of epochs to generate embeddings,
           l: learning rate of embeddings
  Output: E: Vector representation of embeddings for given triangles
subgraphs ← []
for each tri in triangles do
    g_s ← extract_subgraph(G, tri, max_nodes)
    subgraphs.append(g_s)
end
model ← graph2vec(subgraphs, d, e, l)
E ← model.embeddings
return E
```

because, subgraph extraction is a memory-intensive step, and generating embeddings is computationally expensive.

4.4.3 Triangle Embedding using Graph Neural Network

Graph neural networks can be used for the task of node or graph classification. For this project, the graph classification neural network is employed. More specifically, Deep Graph Convolutional Neural Network (DGCNN) [23] implementation is used. This algorithm is not limited to use with DGCNN. Any other graph neural network can be substituted in place of DGCNN.

Algorithm 6: Triangle graph neural network

```
1  function triangle_gnn(G, train_triangles, test_triangles,
     node_embeddings)
   Input  : G: Graph based on simplices
            train_triangles: Vector of triangles for training data
            testing_triangles: Vector of triangles for testing data
            node_embeddings: Vector of node embedding corresponding to
            nodes in G
   Output: train_graphs: Vector of training graphs as GNN objects
            test_graphs: Vector of testing graphs as GNN objects
2  for each tri in train_triangles do
3  |  g_s ← extract_subgraph(G, tri, max_nodes)
4  |  node_label ← triangle_node_labelling(g_s, tri)
5  |  if node_embeddings not None: then
6  |  |  e ← triangle_node_labelling(g_s)
7  |  end
8  |  g_GNN ← GNN_object(g_s, node_label, e)
9  |  train_graphs.append(g_GNN)
10 end
11 Repeat lines 2 to 9 for test_graphs
12 return train_graphs, test_graphs
```

The steps involved in triangle representation for the DGCNN are given in Algorithm 6. DGCNN makes use of the sort pooling as its graph aggregation layer. Triangle embedding using graph neural network algorithm can also take in node embeddings as an additional feature for classification similar to the implementation in [14]. The first step is to create an object that can be consumed by DGCNN. To

accomplish that, a 1-hop subgraph of the triangle is extracted. The nodes in the subgraph are labeled using the Triple-Radius Node Labelling strategy adapted from Double-Radius Node Labelling in [14]. Node labeling is essential for GNN to mark differences between the target nodes and the neighboring nodes. Additionally, if latent features i.e., node embeddings, are to be added, then embeddings corresponding to all the nodes in subgraph are extracted. The above steps are repeated for all the training and testing triangles. Once these training and testing triangles are returned, GNN can be trained on this data for the classification of open and closed triangles. In addition to this, it is essential to note that the algorithm is memory intensive because of the subgraph extraction step, as each subgraph also stores the embeddings for all the nodes involved in the subgraph.

CHAPTER 5

Datasets

5.1 Datasets

Experiments for this project are done based on the timestamped links in x datasets. Hence, every dataset is a list of timestamped nodes. A simplex is formed by a set of nodes which interact at a timestamp as shown in Figure 3. For example, in email network, a simplex comprises of sender and recipients for an email at a give timestamp. A simplex enables to represent more than two interactions at any given timestamp. Overview of dataset statistics is given in Table 2. All the datasets are limited to a maximum of 25 nodes in a simplex. The reason for this exclusion is that, simplices with nodes more than 25 are sparse. A brief description of the datasets used in this project is given below:

1. **Email Networks** (email-Enron [24] and email-Eu [25]): The simplex represent the email-addresses of senders and recipients. Email address is considered as the node. 2 years of data interaction is included in email-Eu.

2. **DAWN** (Drug Abuse Warning Network [1]): The simplex contains the drugs used by the patient. Time is determined by the emergency department visit.

3. **Coauthorship Network** (coauth-DBLP, coauth-MAG-History, coauth-MAG-Geology [1]): Simplex comprises of authors as node and publication date as a timestamp. These datasets have interactions where more than 100 authors have collaborated for a publication.

4. **Thread Participation Network** (threads-math-sx, threads-ask-ubuntu [1]): Node is represented by the users of the platform. Simplex denotes the users

Table 2: Dataset statistic overview

Dataset name	Number of nodes	Number of timestamped simplices
email-Enron	143	10,883
email-Eu	998	234,760
DAWN	2,558	2,272,433
coauth-DBLP	1,924,991	3,700,067
coauth-MAG-History	1,014,734	1,812,511
coauth-MAG-Geology	1,256,385	1,590,335
threads-ask-ubuntu	125,602	192,947
threads-math-sx	176,445	719,792
tags-ask-ubuntu	3,029	271,233
tags-math-sx	1,629	822,059
NDC-classes	1,161	49,724
NDC-substances	5,311	112,405
contact-primary-school	242	106,879
contact-high-school	327	172,035

who have participated in answering a question on the platform. The dataset contains simplices for all the questions on these platforms.

5. **Tagging Network** (tags-ask-ubuntu, tags-math-sx, tags-stack-overflow [1]): A simplex is denoted by the set of annotations (nodes) for a question on the platforms. The dataset contains simplices for all the questions on these platforms.

6. **Drug Networks** (NDC-classes, NDC-substances [1]): These datasets are collected from the National Drug Code Directory. For classes dataset, a node is the class label, and a simplex is the list of class labels for a drug. For substances dataset, a node is the drug substance, and a simplex is the list of substances constituting the drug.

7. **Contact data** (contact-primary-school [26], contact-high-school [27]): In these datasets, a simplex is a collection of students (nodes) who were close to each

Table 3: Training and Testing Data Samples Statistics

Dataset name	Open triangles in training data (first 60%)	Triangles closed between 60-80% (training labels)	Open triangles in testing data (80%)	Triangles closed between 80-100% (testing labels)
email-Enron	4,991	497	6,918	372
email-Eu	190,290	28,323	257,978	25,638
DAWN	3,680,410	191,821	4,891,393	274,728
coauth-DBLP	5,759,224	195,517	8,716,463	270,863
coauth-MAG-History	1,926,010	8,175	2,568,644	4,554
coauth-MAG-Geology	3,938,499	121,161	6,913,171	170,405
threads-ask-ubuntu	141,539	23	173,703	53
threads-math-sx	6,360,454	3,810	9,183,013	4,029
tags-ask-ubuntu	1,813,528	28,618	2,599,696	30,798
tags-math-sx	1,358,225	35,856	1,980,691	40,168
NDC-classes	19,493	5,859	27,701	6,067
NDC-substances	869,008	123,119	1,419,325	95,332
contact-primary-school	53,865	775	82,933	877
contact-high-school	19,593	309	26,506	298

other at a given time.

5.2 Data Preparation

Raw data with timestamped simplices is needed to prepare the dataset for prediction. The raw data is stored in three files, namely *nverts*, *simplices*, *timestamps*. To get one timestamp data, first, a line from *nverts* is read to get the number of nodes in that simplex, and simultaneously a line from timestamp is read, then n number of lines corresponding to the number of nodes are read from *simplices*. Following the steps above gives us all the timestamped simplices in ascending order of time.

Now, to predict the closure of triangles, we need to prepare the simplices. For prediction, the data is split into training and testing dataset by slicing data based on

Table 4: Training and Testing samples with Labels

Nodes tuple for open triangle	Label	Meaning
(1, 2, 3)	1	positive or closure
(4, 5, 15)	0	negative or remains open

timestamps. The statistics for training and testing data are given in Table 3

- **Training data:** A list of open triangles from the first 60 percent of the dataset with respect to time are enumerated. Then a label *one* for these triangles is given based on the condition that the triangle will undergo closure between 60 and 80 percent slice of data. Otherwise, the enumerated triangle is labeled *zero*. For example, as given in Table 4, triangle (2, 5, 10) is open in the first 60 percent of the data and then closes between 60 and 80 percent of the data based on timestamps. Similarly, triangle (4, 5, 15) does not close.

- **Testing data:** A list of open triangles are enumerated from the data slice 0 - 80 percent with respect to time. These triangles are labeled *one* if they undergo closure between 80 - 100 percent data slice. Otherwise, the sample is labeled *zero*. An example is shown in Table 4. The algorithms are never trained on the 80-100 percent data slice as they contain the testing data.

5.3 Training models

Based on the observed interactions until 60 percent of the timestamped simplices, a graph is created. This graph is used then used to generate different types of embeddings. Embeddings efficiently learn the features of the node or graph. Hence, any additional feature is not added for prediction.

Using node2vec, graph2vec, and Graph Neural Network embeddings techniques, an embeddings for each sample is generated. These embeddings, which represent the sample, are then given as input to a binary classifier for prediction. Logistic Regression classifier is used as the binary classifier. The configuration used for this is: *liblinear* as solver, 1000 as maximum iterations, *true* for fit_intercept, and rest of the parameters set to default values as in sklearn library. In this way, the models are trained for link prediction.

5.4 Evaluation metric

The results in [1] are used to compare the performance of the algorithms proposed in this research. The evaluation metric used is the area under the precision-recall curve (AUC-PR). As we can see in Table 3, because of the imbalance in data, AUC-PR serves as a good metric to check how many of the triangle closures are correctly predicted. Using this metric gives us how many triangles closures were predicted correctly. For AUC-PR random baseline is given by,

$$\text{random\_baseline} = \frac{\text{open triangles in test going through closure}}{\text{number of triangles in test set}}$$

The performance of algorithms is the AUC-PR score relative to the random baseline. performance is given by,

$$\text{performance} = \frac{AUC - PR \text{ score}}{\text{random\_baseline}}$$

5.5 Experimental Setup

The implementation language for this project is Python. Python is chosen because a wide variety of machine learning libraries are readily available. For data preparation, feature learning, and classification libraries such as networkx, NumPy,

scipy, pandas, sklearn, Keras, PyTorch are used. node2vec, graph2vec, and Pytorch_DGCNN are used for creating different types of embeddings for the triangles. All the experiments were performed on an AWS EC2 Ubuntu instance with 16 processors and 128GB internal memory. Also, exploratory experiments were performed on Thinkpad t540p with 16GB internal memory.

CHAPTER 6

Experiments and Results

This chapter provides analysis and information about experiments performed on datasets. The first, second, and third section discusses the results obtained from experiments using triangle node2vec embeddings, triangle graph2vec embeddings, and triangle SEAL embeddings, respectively. The experiments focus on applying different embedding techniques to solve the higher-order link prediction (triangle closure) problem. The algorithms are compared against the results in [1]. All the benchmark scores are taken from this paper for comparison. The metric used for comparison is the AUC-PR score relative to the random baseline of the data (refer 5.4).

6.1 Results for Triangle node2vec Embeddings

For this project, various experiments are performed for triangle embedding using the node2vec algorithm. Various datasets are used for experimentation. Information about the datasets is discussed in Section 5.1. The datasets have a problem of imbalance due to a lesser number of triangles going under closure. To tackle this and focus on the positive sample results (triangle closure prediction), the metric used to evaluate is the AUC-PR score relative to the random baseline. Once the triangle embeddings are generated, a simple logistic regression classifier is used to predict the positive and negative samples.

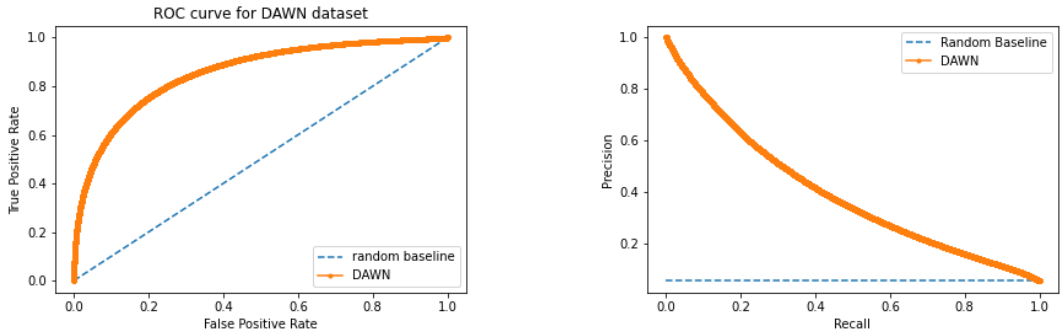
Table 5 summarize the experiment results for triangle embedding using node2vec approach. Results for different types of operators used to learn the triangle embedding are also presented. The algorithms performing the hyperparameter setup for learning node2vec embeddings is configured with 128 dimensions for features, random walk

Table 5: Comparison of results for triangle embeddings with different operators using node2vec. Scores listed are AUC-PR relative to random baseline.

Dataset name	Random baseline	Average	Hadamard	Weighted-L1-average	Weighted-L2-average
email-Enron	0.0537	2.33	**3.54**	1.50	1.58
email-Eu	0.0993	3.17	**3.48**	1.99	1.96
DAWN	0.0561	**7.13**	6.84	3.87	2.88
coauth-DBLP	0.0310	2.16	**3.34**	1.34	1.67
coauth-MAG-History	0.0017	5.17	**7.09**	2.63	2.42
coauth-MAG-Geology	0.0246	**4.34**	4.30	2.23	2.19
threads-ask-ubuntu	0.0003	79.65	**99.88**	60.21	82.23
threads-math-sx	0.0004	**20.52**	16.71	9.53	9.39
tags-ask-ubuntu	0.0118	6.86	**9.12**	4.21	4.03
tags-math-sx	0.0202	2.68	**4.46**	2.07	1.65
NDC-classes	0.2190	1.68	**1.88**	1.78	1.70
NDC-substances	0.0671	**1.53**	1.50	1.13	1.22
contact-primary-school	0.0105	1.30	1.53	**2.36**	2.28
contact-high-school	0.0112	0.90	**1.34**	1.22	1.28

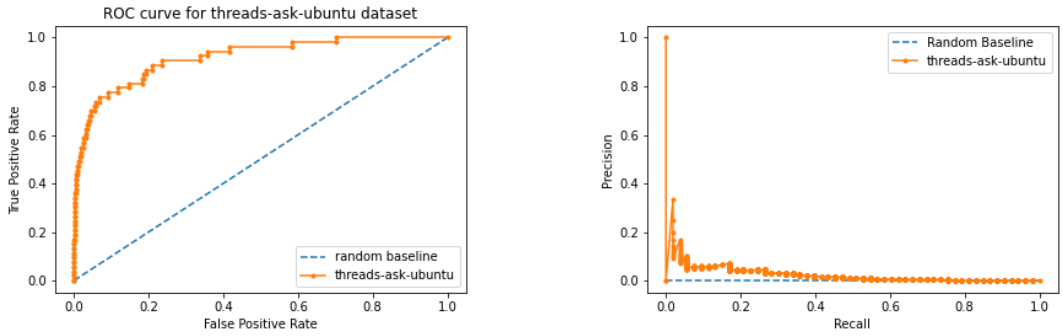
length as 32 or 48, the number of random walks as 10, and return and in-out parameter set as 1. These values for hyperparameters are chosen on the basis of results shown in Table 6, 7 and 8. All values above are the maximum values for the experiments.

From Table 5, we can observe that the Hadamard operator performs the best out of all other operators. This can be attributed to the fact that the Hadamard operator performs better even on edge prediction task. Hadamard operator simply multiples the vector elements, which tend to magnify the features learned by node2vec. On a



(a) ROC-AUC curve (b) AUC-PR curve

Figure 12: AUC scores for triangle embedding with node2vec for DAWN dataset



(a) ROC-AUC curve (b) AUC-PR curve

Figure 13: AUC scores for triangle embedding with node2vec for threads-ask-ubuntu dataset

few datasets, Average operator used to learn embeddings for the triangle performs better than Hadamard. From the results we can observe that node2vec embeddings tend to perform better on larger datasets. The performance of node2vec embeddings can be attributed to the graph structural feature learning capability of node2vec embeddings.

The sample plots of AUC curves for DAWN and threads-ask-ubuntu datasets are given in Figure 12 and Figure 13 respectively.

Table 6 presents the result of triangle embedding using node2vec based on dif-

Table 6: Comparison of results for triangle embeddings with varying embedding dimensions

Dataset name	64	128	192	256
email-Enron	2.62	**3.54**	3.20	3.19
email-Eu	2.19	**3.48**	2.64	2.70
DAWN	6.33	**6.84**	6.62	6.55

Table 7: Comparison of results for triangle embeddings with varying length of random walks.

Dataset name	16	32	48	64	80
email-Enron	3.46	**3.54**	3.40	2.96	2.80
email-Eu	3.44	**3.48**	3.14	3.06	2.98
DAWN	6.62	6.84	**7.01**	6.74	6.69

ferent sizes of dimensions of the embeddings. For all these experiments, random walk length is 32, and the number of random walks is 10. Hadamard operator is used to learn the triangle embeddings from the node2vec embeddings. Specifically, three datasets of varying sizes are chosen for this experiment to generalize over other datasets. These experiments serve as a purpose to identify the ideal length of dimensions for our algorithm. From the results, it can be observed that for all the three datasets, having dimension length as 128 is the best choice. Therefore, all other experiments can be limited to the dimension length of 128.

Table 7 summarizes the results for varying length of random walks. For all the experiments presented in this table, embedding's dimension size is set to 128, and the number of random walks is set to 10. Hadamard operator is used to learning the triangle embeddings from the node2vec embeddings. It can be observed from the results above that a random walk length of 32 or 48 is best suited for our algorithm. One more interesting observation is that on smaller datasets, higher random walk

Table 8: Comparison of results for triangle embeddings with varying number of random walks

Dataset name	10	20	30	40
email-Enron	**3.54**	3.38	2.30	2.85
email-Eu	**3.48**	3.44	2.97	2.73
DAWN	**6.84**	6.90	6.45	6.56

length does not perform well. On the other hand, on the DAWN dataset, a random walk of length 48 performs the best. Therefore, the experiments performed on all other datasets use random walk length as 32 or 48.

Table 8 lists the observation of scores for the algorithm with varying numbers of random walks. Configuration used for these experiments is Hadamard operator, embedding dimension size of 128, and random walk length as 32. From the results, we can conclude that keeping the number of random walks as 10 gives the best results. The scores obtained from these results show that more number of random walks might generalize the features learned by all the nodes as more neighboring node data that is captured by the random walks. Hence, for all other experiments for triangle embedding with the node2vec algorithm, we set the number of random walks as 10.

Thus, it can be concluded from the experiments and results above that triangle embeddings using node2vec perform better or similar on most of the datasets. In addition to this, the Hadamard operator for learning triangle embeddings performs the best, amongst others. Triangle embeddings using node2vec can be used as an alternative for predicting triangle closures. Furthermore, this method can be extended even to tetrahedron closure (4 nodes interacting simultaneously).

Table 9: Comparison of results for triangle classification using graph neural network with varying maximum number of nodes in subgraph

Dataset name	25	50	100	No limit
email-Enron	1.12	**2.46**	2.29	2.29
email-Eu	1.98	**2.90**	2.49	2.56
contact-high-school	**1.65**	1.59	1.57	1.49
contact-primary-school	1.93	2.13	**2.36**	2.23
NDC-classes	1.39	**1.68**	1.66	1.62
threads-ask-ubuntu	1.04	5.24	**9.12**	7.56

6.2 Results for Triangle GNN Embeddings

This algorithm experiments with graph neural networks. The SEAL framework established by [14] makes use of DGCNN and also gives the capability to add latent features in addition to features learned by the neural network. The subgraphs are extracted for each triangle, and the results mostly vary based on the number of nodes in the subgraph. The hyperparameters are kept at the default values of DGCNN i.e., at first, four layers of graph convolution with dimension (32,32,32,1), a SortPooling layer, two 1D convolution layers with 16 and 32 output channels respectively, and lastly a dense layer of 128 neurons. Moreover, a layer of dropout is used to tackle the problem of over-fitting. In addition to this, all experiments are run for 50 epochs, and the best loss and best epoch is determined based on the validation loss achieved. The best epoch is updated only if validation loss is less than the previously stored best loss. To evaluate the model, the AUC-PR score relative to the random baseline is calculated based on the best epoch. Experiments are not performed on all the datasets due to limited computation capacity.

Table 9 summarizes the results for triangle closure prediction using graph neural

Table 10: Comparison of results for triangle classification using graph neural network with varying maximum number of nodes in subgraph and node2vec embeddings used as additional feature.

Dataset name	25	50	100	No limit
email-Enron	1.36	**2.53**	2.27	2.27
email-Eu	2.81	3.11	**3.19**	2.95
contact-high-school	1.07	**1.32**	1.25	1.19
contact-primary-school	1.69	1.97	**2.05**	1.99
NDC-classes	1.70	1.71	1.78	**1.87**
threads-ask-ubuntu	6.95	**7.46**	5.28	6.45

network. The primary objective of these experiments was to get more information about the effect of the maximum number of nodes allowed in each hop. From the results in this table, we can see that it is not always true to include more and more data to get the best results. Generally, the algorithm performs best when we limit the maximum number of nodes in the hop to 50 or 100. Another observation is that the performance is not good compared to scores for triangle embedding using the node2vec algorithm.

Table 10 lists the results for this algorithm using node2vec embeddings as an additional latent feature. The results consider a varying number of maximum nodes contained by the subgraph of each triangle sample. Once the subgraph is extracted based on the limit, then node2vec embeddings for the nodes selected in the subgraphs are concatenated in addition to the features learned by graph neural network to create an embedding for the triangle. These experiments focus on experimenting with node2vec embeddings as they have proven to perform well on this problem, as discussed in Section 6.1. From the result data, we can infer that the performance

of the graph neural network algorithm does not provide us with any substantial improvement. Instead the performance without node2vec embeddings as an additional feature (refer Table 9) is better than this. Moreover, these experiments also have an overhead of additional memory requirements as node2vec embeddings need to stored in memory for all the open triangles. Owing to this, we do not experiment further with this algorithm.

From the experiments and results in this section, it can be concluded that triangle embedding using graph neural network fails to give any performance improvements in higher-order link prediction. In addition to this, when compared to triangle embedding using node2vec, this algorithm is more computation and memory intensive. The failure of this technique can be attributed to the fact that we are extracting subgraphs for each open triangle, which might generalize the features learned for each sample as most of the open triangles can have overlapping subgraphs.

6.3 Results for Triangle graph2vec Embeddings

This section experiments with graph2vec [12] algorithm to generate embeddings for triangles. Extracted subgraphs for each triangle as passed as input to graph2vec for generating embeddings for each triangle. The default configuration of graph2vec is used for the experiments i.e., dimension size of embeddings is 128, the number of epochs is set as 10, and Weisfeiler Lehman feature extraction iterations to 2. Once embeddings for each subgraph are learned, they are passed as input to a simple Logistic Regression classifier for triangle closure prediction. To evaluate the model, the AUC-PR score relative to the random baseline is calculated for model comparison. For this algorithm, experiments are not performed on all the datasets owing to the computational capacity needed by this algorithm.

Table 11: Comparison of results for triangle classification using graph2vec embeddings with varying maximum number of nodes in subgraph

Dataset name	25	50	100	No limit
email-Enron	0.95	**1.19**	1.08	1.12
contact-high-school	1.12	1.14	**1.25**	1.21
contact-primary-school	1.33	**1.57**	1.33	1.49
NDC-classes	1.17	**1.38**	1.37	1.35

Table 11 summarizes the results for triangle embedding with graph2vec technique. The experiments focus on the effect of subgraph size on the results. For this, we test the algorithm on different limits for maximum nodes in the subgraph. From the results, we can conclude that including all the nodes in subgraphs hampers the performance of the algorithm. Therefore, 50 or 100 is an optimal limit for the number of nodes in the subgraph for each triangle. This is consistent with observation in Table 9, and 10 for graph neural network algorithm. As compared to the scores for triangle embedding using node2vec model score (refer Table 5), this algorithm performs worse. Another important observation is that, triangle embedding using graph2vec performs just better than the random baseline on almost all the datasets.

6.4 Results Comparison

Table 12 summarize the results for different triangle embedding algorithms discussed above with the prediction models in previous work i.e. the benchmark scores. The table also specifies the scores for previous work for each dataset, as given in [1]. The classification models that are used in the previous work are: Logistic Regression for email-Eu, coauth-DBLP, coauth-MAG-History, coauth-MAG-Geology, threads-math-sx, tags-math-sx, NDC-substances, and contact-primary-school; Ge-

Table 12: Comparison of results for all triangle embeddings algorithms with previous work. Scores listed are AUC-PR relative to random baseline.

Dataset name	Random Baseline	Previous work scores [1]	triangle-node2vec	triangle-gnn	triangle-graph2vec
email-Enron	0.0537	3.16	**3.54**	2.53	1.19
email-Eu	0.0993	3.47	**3.48**	3.19	x
DAWN	0.0561	4.77	**7.13**	x	x
coauth-DBLP	0.0310	**3.37**	**3.34**	x	x
coauth-MAG-History	0.0017	6.75	**7.09**	x	x
coauth-MAG-Geology	0.0246	**4.74**	4.34	x	x
threads-ask-ubuntu	0.0003	80.94	**99.88**	9.12	x
threads-math-sx	0.0004	**47.18**	20.52	x	x
tags-ask-ubuntu	0.0118	**12.64**	9.12	x	x
tags-math-sx	0.0202	**13.99**	4.46	x	x
NDC-classes	0.2190	**4.43**	1.88	1.87	1.38
NDC-substances	0.0671	**8.17**	1.53	x	x
contact-primary-school	0.0105	**6.91**	2.36	2.36	1.57
contact-high-school	0.0112	**4.16**	1.34	1.65	1.25

ometric mean for tags-ask-ubuntu threads-ask-ubuntu, and contact-high-school; Harmonic mean for tags-stack-overflow, and NDC-classes, Adamic-Adar for DAWN; PageRank for email-Enron. Experiments not performed for the dataset are marked with 'x'. These experiments were not performed because of limited computation and memory resources.

From Table 12 we can observe that triangle embedding using node2vec performs better compared to graph2vec, and graph neural networks algorithms. The best scores from all of the experiments discussed are compared with the prediction model scores from previous work [1]. We can conclude from the results that triangle embedding using node2vec performs better or similar on most of the datasets when compared to the prediction models in previous work. For Coauthorship networks, triangle embedding performs better or similar to the benchmark scores. Also, for email networks, our algorithm tends to perform slightly better. The best performance is achieved for

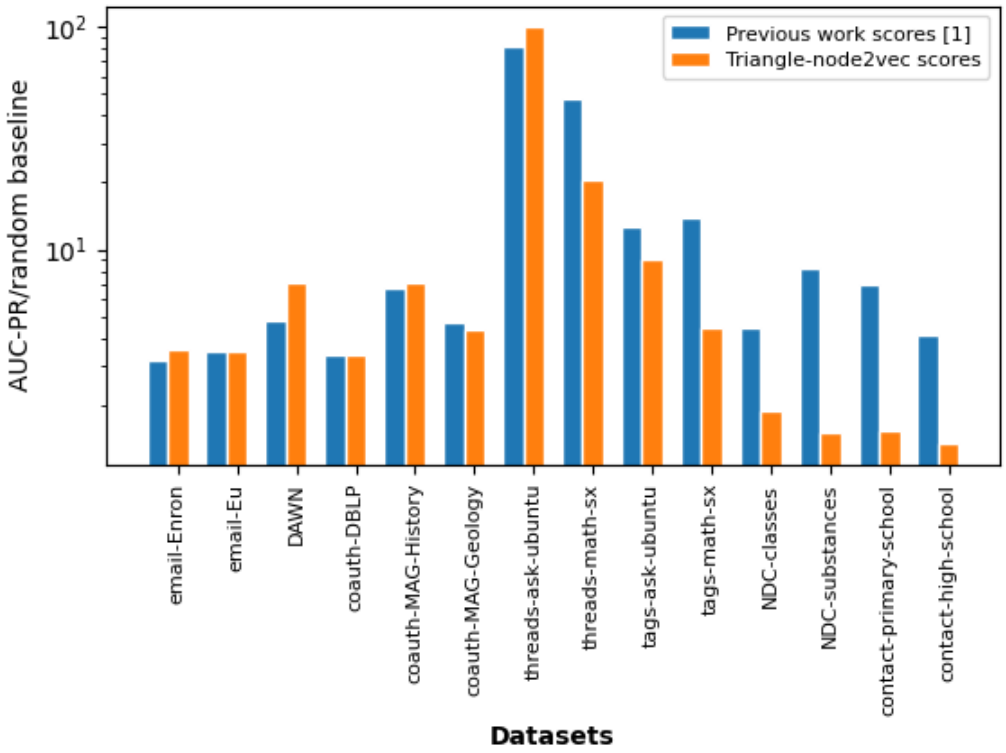


Figure 14: Score comparison between previous work and triangle embeddings using node2vec. Log scale is used to enhance lower magnitude scores

the DAWN dataset, which shows a 43% increase from the benchmark score. Previous work scores rely heavily on the graph structural features in the dataset. Similarly, node2vec embeddings are known to learn representation for each node and preserve the graph structural features. Therefore, we can attribute the performance of triangle embeddings using node2vec to the above characteristics.

Results from Table 12 are visualized in Figure 14. The figure compares the best scores for the prediction models used in previous work and triangle embedding using the node2vec. A log scale is used on the y-axis to enhance the smaller performance improvement scores in the plot.

Table 13: Comparison of number of edges and open triangles in testing data

Dataset name	nodes	edges	open triangles
email-Enron	140	1,607	6,918
email-Eu	952	26,582	257,978
contact-high-school	327	5,225	26,506
contact-primary-school	242	7,575	82,933
NDC-classes	1,084	5,593	27,701
threads-ask-ubuntu	80,258	168,758	173,703

Table 13 gives the statistics of the datasets used in experiments for graph neural networks and graph2vec based triangle closure prediction. From the data presented in this table, we can see that there is an exponential increase in the number of interactions in proportion to the number of nodes involved in an interaction. For example, the email-Enron dataset has 140 nodes i.e., single node interactions, 1,607 edges i.e., two-node interactions, and 6918 open triangles i.e., possible three-node interactions. Hence, when we consider higher-order structures, there are more combinations possible for the interactions between nodes. We can correlate this data with the failure of subgraph embedding methods (graph neural networks and graph2vec). When we consider the subgraphs of triangles, there is a higher possibility of overlapping [1] between the extracted subgraphs. This can make it difficult for graph neural network and graph2vec to learn features optimally for the open triangles. As opposed to this, the chance of overlapping when we consider edge interactions is relatively less because of the small magnitude of the number of edges. Therefore, link prediction using subgraphs and graph neural networks gives better results for pairwise interactions, as seen in [14]. To make these methods work, we need to find a way to extract features that represent the higher-order nature of the data.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

This project focuses on solving the problem of higher-order link prediction, and more specifically, closure of open triangles. The problem is solved using different types of embeddings, particularly node embeddings, graph embeddings, and graph neural networks. Triangle embeddings using node2vec leverages the node embeddings generated for the graph and combines it using different operators to represent an open triangle. This algorithm performs substantially better on some of the datasets tested. The reason for this can be attributed to the basis that node2vec learns graph structural features of higher quality. Additionally, it is discussed in the related work that generally graph structural features perform better for the task of link prediction.

On the other hand, the other two algorithms i.e., triangle embeddings using graph2vec and graph neural network, suffer in the prediction performance. The reason for that can be attributed to the similarity and overlapping of subgraphs for open triangles in the datasets. Another reason is that we are extracting subgraphs for higher-order structures from a 2D graph, which cannot represent higher-order structures efficiently. Experiments performed with a varying number of nodes in 1-hop subgraphs give better results where smaller values of nodes are chosen, which indicates that having less context about neighborhoods is better. From the experiments performed, we can conclude that learning triangle embeddings using node2vec performs better than the other two proposed solutions. Most importantly, triangle embeddings using node2vec give better or similar performance than the current benchmark models on most of the datasets.

7.2 Future Work

Future direction to improve the results is to add data specific attributes to node2vec embeddings to increase prediction performance. Edge weights in the projected graph of all simplices can be a potential attribute to concatenate with node2vec embeddings. In addition to this, all the proposed techniques are generic enough to be extended for the task of other higher-order structure predictions like tetrahedron closures. Moreover, the research in this area will hugely benefit from an embeddings approach that extracts features from higher-order structures. Exploring Hasse diagram and random walks on those diagrams can be a way forward to obtaining embeddings for higher-order structures. Higher-order structures like simplices will prove useful in the research for social network analysis, news topic connections, and drug combination research.

LIST OF REFERENCES

[1] A. R. Benson, R. Abebe, M. T. Schaub, A. Jadbabaie, and J. Kleinberg, "Simplicial closure and higher-order link prediction," *Proceedings of the National Academy of Sciences*, vol. 115, no. 48, pp. E11221–E11230, 2018.

[2] J. C. W. Billings, M. Hu, G. Lerda, A. N. Medvedev, F. Mottes, A. Onicas, A. Santoro, and G. Petri, "Simplex2vec embeddings for community detection in simplicial complexes," *arXiv preprint arXiv:1906.09068*, 2019.

[3] M. T. Schaub, A. R. Benson, P. Horn, G. Lippner, and A. Jadbabaie, "Random walks on simplicial complexes and the normalized hodge laplacian," *CoRR*, vol. abs/1807.05044, 2018.

[4] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Journal of the American society for information science and technology*, vol. 58, no. 7, pp. 1019–1031, 2007.

[5] M. Zhang and Y. Chen, "Weisfeiler-lehman neural machine for link prediction," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, (New York, NY, USA), pp. 575–583, ACM, 2017.

[6] A. R. Benson, D. F. Gleich, and J. Leskovec, "Higher-order organization of complex networks," *Science*, vol. 353, no. 6295, pp. 163–166, 2016.

[7] A. Hatcher, *Algebraic topology*. Cambridge: Cambridge University Press, 2002.

[8] R. L. Graham, M. Grötschel, and L. Lovász, eds., *Handbook of Combinatorics (Vol. 2)*. Cambridge, MA, USA: MIT Press, 1995.

[9] C. Berge, *Graphs and Hypergraphs*. Oxford, UK, UK: Elsevier Science Ltd., 1985.

[10] M. E. J. Newman, D. J. Watts, and S. H. Strogatz, "Random graph models of social networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. suppl 1, pp. 2566–2572, 2002.

[11] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 855–864, ACM, 2016.

[12] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *CoRR*, vol. abs/1707.05005, 2017.

[13] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," *CoRR*, vol. abs/1403.6652, 2014.

[14] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Advances in Neural Information Processing Systems*, pp. 5165–5175, 2018.

[15] D. E. Knuth, *The Stanford GraphBase: A Platform for Combinatorial Computing*. New York, NY, USA: ACM, 1993.

[16] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pp. II–1188–II–1196, JMLR.org, 2014.

[17] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[18] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[19] Y.-Y. Chen, Q. Gan, and T. Suel, "Local methods for estimating pagerank values," in *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, CIKM '04, (New York, NY, USA), pp. 381–389, ACM, 2004.

[20] X. Jia, H. Liu, L. Zou, J. He, X. Du, and Y. Cai, "Local methods for estimating simrank score," in *Proceedings of the 2010 12th International Asia-Pacific Web Conference*, APWEB '10, (Washington, DC, USA), pp. 157–163, IEEE Computer Society, 2010.

[21] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *CoRR*, vol. abs/1901.00596, 2019.

[22] E. C. Mutlu and T. A. Oghaz, "Review on graph feature learning and feature extraction techniques for link prediction," *CoRR*, vol. abs/1901.03425, 2019.

[23] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification.," in *AAAI*, 2018.

[24] B. Klimt and Y. Yang, "The enron corpus: A new dataset for email classification research," in *European Conference on Machine Learning*, pp. 217–226, Springer, 2004.

[25] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pp. 601–610, 2017.

[26] J. Stehlé, N. Voirin, A. Barrat, C. Cattuto, L. Isella, J.-F. Pinton, M. Quaggiotto, W. Van den Broeck, C. Régis, B. Lina, *et al.*, "High-resolution measurements of face-to-face contact patterns in a primary school," *PloS one*, vol. 6, no. 8, 2011.

[27] R. Mastrandrea, J. Fournet, and A. Barrat, "Contact patterns in a high school: a comparison between data collected using wearable sensors, contact diaries and friendship surveys," *PloS one*, vol. 10, no. 9, 2015.