Higher-order Link Prediction Using Triangle Embeddings

Neeraj Chavan
Department of Computer Science,
San Jose State University,
San Jose, USA
Email: neerajpadmakar.chavan@sjsu.edu

Katerina Potika
Department of Computer Science,
San Jose State University,
San Jose, USA
Email: katerina.potika@sjsu.edu

2020 IEEE International Conference on Big Data (Big Data) | 978-1-7281-6251-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/BigData50022.2020.9377750

***Abstract*—Higher-order structures, like triangles, in networks provide rich information about a network. Usually, the focus is on pairwise interactions that are modeled as edges. However, many interactions may actually involve more than two nodes 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. Predicting the occurrence of such higher-order structures helps us solve problems in various disciplines, such as social network analysis, drug combinations research, and news topic connections.**

The primary focus of this paper is to explore representations of three-node interactions, called triangles (a special case of higher-order structures) in order to predict higher-order links. We propose new methods to embed triangles by generalizing the node2vec algorithm under different operators, by using 1-hop subgraphs in the the graph2vec algorithm, and in graph neural networks. The performance of these techniques is evaluated against some benchmark scores on various datasets used in the bibliography. From the results, it is observed that our node2vec based triangle embedding method performs better or similar on most of the datasets compared to previous models.

I. Introduction

Link prediction is an important problem that deals with the task of how likely it is that two nodes in a graph will be connected at some point in the future. Link prediction may go beyond the pairwise associations and predict relationships among more than three nodes. Moreover, much of the information in graphs contain information between more than two nodes [1]. We can consider some simple 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 higher order interactions are omnipresent but have received little attention. As a sequence, recently there has been an increasing amount of research papers that focus on such interactions.

Here we study the problem of predicting higher-order links, i.e., co-appearance relationships among three or more nodes [1]. This version is an extension of the classical link prediction, which is between two nodes. One of the problems when working with higher-order structures is that the usual representations of graphs do not capture such structures. Hence, most of the time, these interactions are lost right at the data collection stage, where data is collected directly in a graph format. The existing models of higher-order structures use different representation ways, some include simplicial complexes [2], set systems [3], hypergraphs [4], and bipartite affiliation graphs [5]. In this paper, we use simplicial complexes for modeling higher-order structures.

Higher-order link prediction problems have many applications in fields such as the involvement of multiple chemicals in reactions [6], in the study of new types of drugs [1], in suggesting groups in social media [7], in collaborations among researchers [1], and in biological interactions between sets of molecules [7].

One way to solve network analysis problems, like the link prediction, that has gained popularity in recent years is that of the use of Machine learning (ML) and Deep Learning (DL) techniques. In order to use these techniques graphs and nodes are embedded in a d-dimensional euclidean space creating graph or node feature vectors. The performance of the above techniques depends on the quality of these feature vectors. The task of generating feature vectors is difficult because of the structure of graphs and the corresponding field is known as representation learning.

Known representation techniques are node2vec [8], graph2vec [9], Deepwalk [10], which are used for representing nodes and graphs as embeddings. These embeddings are generated for nodes or graphs. However, embedding a higher-order structure is not straightforward. To fill this gap, we extend existing node and graph embeddings to generate higher-order structure embeddings, specifically edge and triangle embeddings. Our methods can be further extended to other higher-order structures (like cliques of any size) in the future.

We are proposing a framework that combines various graph and node embedding techniques, like node2vec [8], graph2vec [9], and graph neural networks [11] for higher-order link prediction. The performance of these algorithms are compared against benchmark results [1] that are performed for higher-order link prediction.

This paper is organized as follows: Section II provides the necessary terminologies; Section III discusses the existing graph embedding techniques, and higher-order link prediction techniques; Section IV explains the proposed methodology and

algorithms; Section V describes the datasets and evaluation metrics; Section VI contains the experiments, and results; Section VII gives the conclusion.

II. Terminology

Let us first define the necessary terminologies used throughout this work. A graph $G = (V, E)$ is contains a set of nodes V, and edges E connecting pairs of nodes. 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.

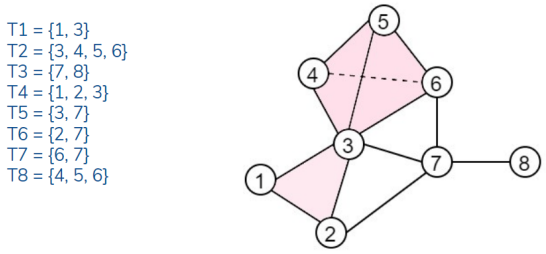

Fig. 1: Example of a temporal graph with simplices.

If a graph changes according to time, then it is known as a *temporal graph*. 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. are the co-appearance set of nodes and show the evolution of the graph according to time.

Moreover, a higher-order structure is a structure having interaction between more than two nodes simultaneously (co-appearance). In our example of Figure 1 instance $T4$ involves three nodes interacting at once. All three $T2$, $T4$, and $T8$ depict higher-order structures, and are shaded.

Some definitions related to higher-order structures from [1] follow:

- **Simplex**: A simplex is a way to represent triangles, and tetrahedrons in terms of dimensions. In Figure 1, each T (timestamp) represents a simplex. More, specifically a 0-simplex is a point, 1-simplex is a line, 2-simplex is a triangle, and a 3-simplex is a tetrahedron. In our example $T4$ is a 2-simplex.
- **Open Triangle**: 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 a simplex), then it is called as an open triangle.
- **Closed Triangle**: Given an open triangle, if all the nodes of it appear together in a simplex the triangle undergoes closure, i.e., is now a closed triangle.
- **Simplicial Closure**: If there is an open k-clique (similar to open triangle but with k nodes) then the appearance of a new simplex containing these k nodes is called as a simplicial closure instance. This transforms an open structure to a closed one.

Recall, that we represent nodes, edges, and triangles using embeddings. An embedding is a vector representation of features.

III. Related Work

In this section we briefly describe current approaches used for computing node and graph embeddings, and methods used for higher-order link prediction. Also, we mention studies about the techniques used for extracting relationships between nodes, and various ways to represent the embeddings for higher-order structures.

In [1], the importance of higher-order interactions in analyzing graphs is considered. They study the organizational principles of higher-order structures in real-world datasets. Motivated by the importance of triangular structures, and the triadic closure, they introduce the simplicial closure. They propose a higher-order link prediction problem, which predicts this simplicial closures. They use the projected graph with the weight of an edge representing the frequency of those two nodes co-appearing. 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, and AUC-PR is used as a metric for the prediction performance. Eight different models are compared. None model performs better on all datasets. Analysis of the performance suggests that open triangles with strong ties are most likely for a closure. Additionally, using generalized means of edge weights gives a strong indication of closures of open triangles. They observe that the higher-order link prediction is challenging because of the absolute performance achieved. They suggest a future application for embeddings in higher-order link prediction.

We work on their ideas and incorporate node2vec [8], and graph2vec [9] as the embedding techniques as they preserve the structural significance in their representations.

An area of investigation in graph embedding focuses on node-level embeddings. In [8], the node2vec technique uses the local search method. This method is used to extract the neighboring node information and generating sequences from nodes using second-order random walks.

The way in which node2vec is employed for pairwise link prediction, is the same way we can use it for performing higher-order link prediction. The quality of the 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.

The graph2vec [9] algorithm, as the name suggests, generates vectors for graphs. It is an innovative approach to learn the entire representation of a graph. This technique can be employed to work for higher-order link prediction by giving to it each open triangle's 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.

Similar to graph2vec [11], SEAL can learn a representation for a subgraph. Furthermore, [11] uses the subgraph to interpolate and learn the graph structure features (used by traditional

heuristic scores). For link prediction traditional heuristic scores like PageRank [12], and Preferential Attachment [13] are used, but these 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, one can use Graph Neural Networks (GNNs). Moreover, graph structure features learned from local subgraphs using GNN are used in [11].

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. GNN performs better in the graph feature learning ability in comparison to fully-connected neural networks and graph kernels [11]. This ability of GNN for link prediction is also validated in [14] and [15]. 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.

IV. Methodology

Given a timestamped simplex $\{S_i, t_i\}$ of a graph $G = (V, E)$, 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 $(k-1)$-simplex. This can also be called as a k-clique. The process of predicting the occurrence of more than two nodes simultaneously can be best described as the problem of higher-order link prediction [1]. For this work, 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 3. This problem can be split into three phases: enumerating all open triangles and closed triangles in the training and testing dataset, representing the triangles as embeddings, and preforming the triangle closure prediction.

The framework is divided into different modules as shown in Figure 2. The steps in the workflow are as follows:

1) The first module takes in the raw graph data as input and returns a list of timestamped simplices containing nodes.
2) In the next module, the data is divided into training and testing. Then, a labeled dataset of open triangles is created based on the data.

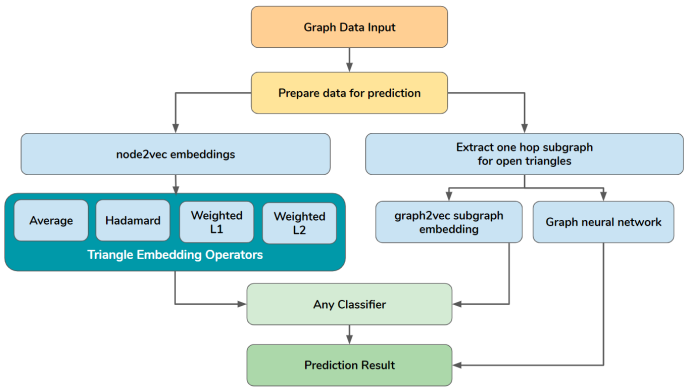

Fig. 2: Workflow of algorithms to perform higher-order link prediction task

3) Once we have the prediction data ready, embeddings are generated for the data. Starting from the node embeddings, we learn the triangle embeddings using different operators specified later. Using the 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 the graph2vec, and the 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 a binary classifier, which will predict if the open triangle undergoes closure or not.
6) Similar to the previous module, the graph neural network receives extracted subgraphs, and we add some additional info 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 steps and algorithms used in this study. Next, all the above algorithms are discussed in greater detail.

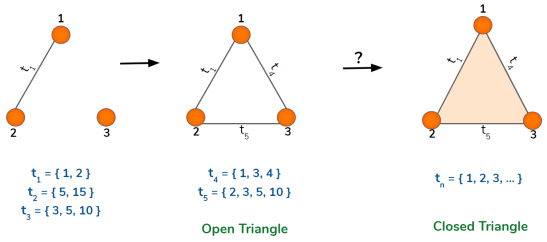

Fig. 3: A sample lifecycle of triangle closure.

For the task of the triangle closure prediction, embeddings of different types are applied to represent an open triangle. Before we begin, let us take a look at how triangles can undergo closure. In Figure 3, t_1 shows the observed graph at that point, use of edge $(1, 2)$. After t_4, t_5, the triangle converts into an open triangle, edges $(1, 3)$ and $(2, 3)$. 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 $1, 2$, and 3 appear in a bigger simplex as subsets.

Algorithm 1: Enumerating open triangles

```
function openTriangles(G, S):
Input  : G: networkx graph representation of S,
         S: Vector representation of set of nodes for a
         data slice based on timestamp
Output: openTriangles: vector representation of
         triangles that are still open in this data slice
// Set of triangles already gone through closure
closedTriangles ← getClosedTriangles(S)
openTriangles ← set()
for each edge(u, w) in G do
    // iterate over all graph nodes
    for each vertex(v) in G do
        if edge(u,v) in G and edge(v,w) in G then
            if tuple(u,v,w) not in closedTriangles then
                openTriangles.add((u,v,w))
openTriangles ← list(openTriangles)
return openTriangles
```

Algorithm 2: Enumerating new triangle closures

```
function newClosures(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: newTriangles: set representation of triangles
         that have closed in this data slice
// Set of triangles already gone through closure in S_old
closedTriangles ← getClosedTriangles(S_old)
newTriangles ← 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 V then
            if (i, j, k) not in closedTriangles then
                newTriangles.add((i, j, k))
return newTriangles
```

Now, the next challenge is to prepare the data for prediction. For prediction, we have to split the dataset in training and testing. In preparing the data, there are two essential algorithms to consider, enumerating all 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 to 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, in order 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 the testing data, where S_{old} consists of 0 to 80 percent simplices, and S_{new} consists of 80 to 100 percent simplices.

Algorithm 1 explains how all the 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 all open triangles is expensive for large datasets as the algorithm is $O(m \cdot n)$, where m is the number of edges, and n is the number of nodes. To make the process faster, the for loop can be parallelized.

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 $\in V$ and that the combination has not undergone closure already. If these conditions are satisfied, the 3 node combination is added to a set of $newTriangles$.

Three methods for triangle embeddings are discussed in the subsections that follow.

A. Triangle Embeddings using node2vec - Method 1

The embeddings generated by node2vec can be converted into an embedding for an open triangle using four different operators: *Hadamard, Average, WeightedL1, and WeightedL2*. Figure 4 illustrates the steps in which an embedding is generated for an open triangle using node2vec. In the first step, an open triangle is represented, which is to be classified. Then the node2vec algorithm is executed on the training graph, and the output with the node embeddings for each node is generated as shown in the second step of Figure 4. The last step is to combine the node embeddings using the specified operator and generate an embedding for the open triangle.

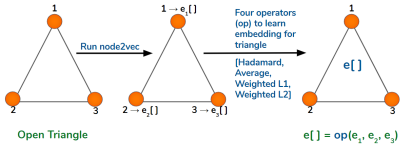

Fig. 4: Workflow for generating node2vec embeddings

A visual depiction of the algorithm is given in Figure 4. In the first step, the node2vec algorithm is run on the graph G to generate embeddings for each node. Parameters like the dimension of the vector, the length of the random walk, and the number of random walks can be passed to the algorithm

TABLE I: Operators to learn triangle features from node embeddings

Operator	Definition
Average (O_1)	$E_{average} = \frac{\sum_{j=0}^{2} f_i(v_j)}{3}$
Hadamard (O_2)	$E_{hadamard} = \prod_{j=0}^{2} f_i(v_j)$
Weighted-L1-average (O_3)	$E_{l1} = \frac{\sum_{j=0}^{2} \lvert f_i(v_j) - f_i(v_{(j+1)\%3})\rvert}{3}$
Weighted-L2-average (O_4)	$E_{l2} = \frac{\sum_{j=0}^{2} \lvert f_i(v_j) - f_i(v_{(j+1)\%3})\rvert^2}{3}$

for experimenting with different configurations. The next step is to iterate over the triangles and learn an embedding for each triangle. Table I shows the different operators to learn the triangle features. These operators are adapted from binary in [8], to ternary operators for triangle embeddings. 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.

B. Triangle Embedding using Graph Neural Network - Method 2

We choose Deep Graph Convolutional Neural Network (DGCNN) [16] in our approach. This algorithm is not limited to use with DGCNN. Any other graph neural network can be substituted in place of DGCNN.

Algorithm 3: Triangle graph neural network

function gnnTriangle(G, trainTriangles, testTriangles, nodeEmbeddings)
Input : G: Graph based on simplices
$trainTriangles$: Vector of triangles for training data
$testingTriangles$: Vector of triangles for testing data
$nodeEmbeddings$: Vector of node embedding corresponding to nodes in G
Output: $trainGraphs$: Vector of training graphs as GNN objects
$testGraphs$: Vector of testing graphs as GNN objects
for *each tri in trainTriangles* **do**
  $g_s \leftarrow$ extractSubgraph(G, tri, maxNodes)
  nodeLabel $\leftarrow$ triangleNodeLabelling(g_s , tri)
  if *nodeEmbeddings not None:* **then**
    e $\leftarrow$ triangleNodeLabelling(g_s)
  $g_{GNN} \leftarrow$ GNNObject(g_s, nodeLabel, e)
  trainGraphs.append(g_{GNN})
Repeat lines 2 to 9 for $testGraphs$
return $trainGraphs$, $testGraphs$

Algorithm 3 gives the steps for the DGCNN triangle representation. DGCNN makes use of the sort pooling as it's graph aggregation layer. Triangle embeddings using the graph neural network algorithm can also take in the node embeddings as an additional feature for the classification similar to the implementation in [11]. 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 [11]. Node labeling is essential for GNN to mark differences between the target nodes and the neighboring nodes. Additionally, if some latent features i.e., node embeddings, are to be added, then the embeddings corresponding to all the nodes in the 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.

C. Triangle Embedding using graph2vec - Method 3

Next, we use the graph2vec [9] embedding technique that learns features of subgraphs and generates an embedding for each subgraph. The graph2vec is 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 5. By using this method, subgraphs for all the triangles are extracted, and an embedding is generated for each of these subgraphs.

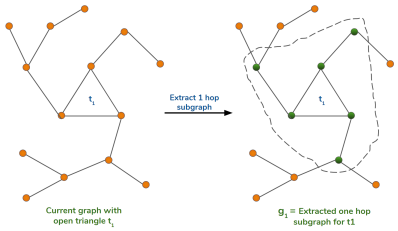

Fig. 5: Subgraph extraction example

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, 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 $maxNodes$. 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 6 illustrates the graph2vec workflow for learning embeddings for the triangle. The procedure for learning triangle embeddings is given in Algorithm 5. The first step

Algorithm 4: Extract one hop subgraph

function extractSubgraph($G, tri, maxNodes$)
Input : G: observed graph based on simplices
tri: vector representing 3 vertices in the triangle
$maxNodes$: 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))
if *maxNodes > 0 and maxNodes < length(nodes)* **then**
  nodes = random.sample(nodes, maxNodes)
nodes.add(tri) // add triangle vertices
G_s = G.subgraph(nodes) //networkx function to get subgraph from nodes
return G_s

Algorithm 5: Triangle graph2vec Embedding

function graph2vecTriangle($G, triangles, d, e, l$)
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 ← extractSubgraph(G, tri, maxNodes)
  subgraphs.append(g_s)
model ← graph2vec(subgraphs, d, e, l)
E ← model.embeddings
return E

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

V. Datasets

Experiments are done based on the timestamped links in the 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 1. For example, in the email networks, a simplex comprises of a sender and a recipients for an email at a give timestamp. A simplex enables to represent more than two interactions at any given timestamp.

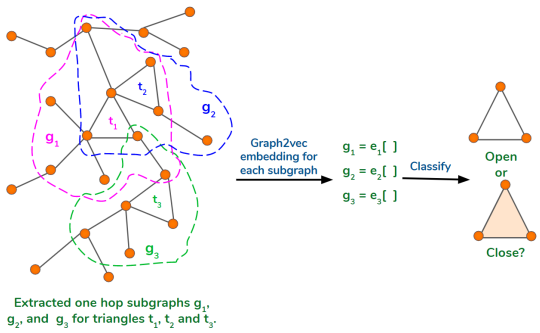

Fig. 6: Example of Graph embeddings using graph2vec

TABLE II: Dataset statistic overview

Dataset name	# nodes	# 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

An overview of the datasets statistics is given in Table II. All the datasets are limited to a maximum of 25 nodes in each 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 work is given below:

1) **Email Networks** (email-Enron [17] and email-Eu [18]): The simplex represent the email-addresses of senders and recipients. The email address is considered the node, and 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 nodes and publication date as the 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]): Nodes represent the users of the platform. Simplex denotes the users 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 all these simplices.

6) **Drug Networks** (NDC-classes, NDC-substances [1]): These datasets are collected from the National Drug Code Directory. For the classes dataset, a node is the class label, and a simplex is the list of class labels for a drug. For the 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 [19], contact-high-school [20]): In these datasets, a simplex is a collection of students (nodes) who were close to each other at a given time.

First, we have to prepare our data. The raw data with timestamped simplices must be prepared to a 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 the timestamp is read, then n number of lines corresponding to the number of nodes are read from *simplices*. Following all the above steps 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 timestamps.

TABLE III: 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

- **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 III, triangle $(2, 5, 10)$ is open in the first 60% 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 $0 - 80\%$ with respect to time. These triangles are labeled *one* if they undergo closure between $80 - 100\%$ data slice. Otherwise, the sample is labeled *zero*. The algorithms are never trained on the $80 - 100\%$ data slice as they contain the testing data.

In order to train our models we have the following steps. Based on the observed interactions until 60% of the timestamped simplices, a graph is created. This graph is used then 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. A 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.

For the evaluations we used the results in [1] and compare against the performance of our algorithms. The evaluation metric used is the area under the precision-recall curve (AUC-PR). Because of the imbalance in data, AUC-PR serves as a good metric. Using this metric gives us how many triangles closures were predicted correctly. For AUC-PR random baseline is given by,

$$\text{randomBaseline (RB)} = \frac{\text{open triangles in test through closure}}{\text{number of triangles in test set}}$$

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

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

VI. Experimental Setup

The implementation language for this study is Python. For data preparation, feature learning, and classification libraries such as networkx, NumPy, scipy, pandas, sklearn, Keras, PyTorch are used. We use node2vec, graph2vec, and Pytorch_DGCNN 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.

Let us start by providing the analysis and information about the experiments performed on the datasets. The three methods we have in our experiments using triangle node2vec embeddings (Method 1), triangle SEAL embeddings (Method 2), and triangle graph2vec embeddings (Method 3), 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.

A. Results for triangle node2vec Embeddings - Method 1

Various experiments are performed for triangle embeddings using the node2vec algorithm. Various datasets are used for experimentation. 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 IV summarizes the experimental results for triangle embedding using Method 1. Results for different types of operators used to learn the triangle embeddings are also presented.

TABLE IV: Comparison of results for triangle embeddings with different operators using node2vec. Scores listed are AUC-PR relative to random baseline.RB: Random baseline

Dataset name	RB	O_1	O_2	O_3	O_4
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

The algorithms performing the hyperparameter setup for learning node2vec embeddings is configured with 128 dimensions for features, random walk 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 various tries. We can observe that the Hadamard operator (O_2) performs the best out of operators. This can be attributed to the fact that the Hadamard operator performs better even on the edge prediction task. The Hadamard operator simply multiples the vector elements, which tend to magnify the features learned by node2vec. On few datasets, the Average operator used to learn embeddings for the triangle performs better than the Hadamard. From the results we can observe that the node2vec embeddings tend to perform better on larger datasets. The performance of the node2vec embeddings can be attributed to the graph structural feature learning capability of node2vec embeddings.

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. 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).

B. Results for Triangle GNN Embeddings (Method 2)

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 V: 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

TABLE VI: 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

Table V summarizes the results for triangle closure prediction using graph neural 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 optimal 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.

Table VI 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 the 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. 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 V) 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 results in this section, it can be concluded that triangle embeddings using graph neural network fails to give any performance improvements in higher-order link prediction. In

TABLE VII: Comparison of Method 3 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

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.

C. Results for Triangle graph2vec Embeddings (Method 3)

We use the graph2vec [9] algorithm to generate embeddings for triangles. Extracted subgraphs for each triangle are 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 the embeddings for each subgraph are learned, they are passed as input to a simple logistic regression classifier for the 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 VII 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 a 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 the observations in Table V, and VI for Method 2. As compared to the scores for Method 1 (refer Table IV), this algorithm performs worse. Another important observation is that, triangle embeddings from Method 3 perform just better than the random baseline on almost all the datasets.

D. Results Comparison

Table VIII summarizes the results for different Methods proposed above with the prediction models in previous work, i.e., the benchmark scores. This 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; geometric 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 datasets are marked with 'x'. These experiments were not performed because of limited computation and memory resources.

From Table VIII we can observe that creating triangle embeddings using Method 1 performs better compared to Method 2, and Method 3. 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 Method 1 performs better or similar on most of the datasets when compared to the prediction models in previous work. For the Coauthorship networks, our Method 1 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 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 of Method 1 to the above characteristics.

Table IX gives the statistics of the datasets used in the experiments for Method 2 and Method 3. 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 $6,918$ 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 the subgraph embedding methods (Method 2 and Method 3). When we consider the subgraphs of triangles, there is a higher possibility of overlapping [1] between the extracted subgraphs. Therefore, it is difficult for graph neural network of Method 2, and graph2vec of Method 3 to learn features optimally for the open triangles.

VII. Conclusion and Future Work

We propose a framework that uses various triangle embedding methods for solving the problem of higher-order link prediction, and more specifically, the closure of open triangles. The different types of embeddings we consider are node embeddings, graph embeddings, and graph neural networks. Triangle embeddings using node2vec leverage the node embeddings generated for the graph and combines it using four different operators to represent an open triangle. This method performs substantially better on some of the datasets tested. The reason for this can be attributed to the basis that this Method 1 uses node2vec that learns graph structural features of higher quality.

On the other hand, the other two methods, 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

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

Dataset name	RB	Scores [1]	Method 1	Method 2	Method 3
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

TABLE IX: 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

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.

Another application we plan to consider is on predicting groups (higher-order structures) of news articles and social media posts about important issues, like the COVID-19 pandemic, that will co-appear in a source at some point. Through that grouping one can detect and find fake news.

Future direction is to explore if Hasse diagrams 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.

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.