Karlsruhe Institute of Technology

Towards the Evaluation of Graph Embeddings with Deep Decoders

Bachelor's Thesis of

Paul Johannes Wagner

At the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	T.T.-Prof. Dr. Thomas Bläsius
Second reviewer:	Prof. Dr. Dorothea Wagner
Advisors:	Dr. Maximilian Katzmann

18.01.2023 – 19.05.2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text. I also declare that I have read and observed the *Satzung zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie.*

Karlsruhe, 19.05.2023

. . . . . .

(Paul Johannes Wagner)

Abstract

Graphs are ubiquitously used to represent relationships between objects and data in real-world scenarios. They are, for example, used for road networks, citations of scientific publications, social networks, and program code dependencies. Relevant problems when handling graphs like link prediction and node classification involve predicting, comparing, and visualizing graphs and their nodes. These tasks can be addressed using graph embeddings. To achieve accurate results, the embedding has to capture as much information about the graph as possible while keeping the dimension of the embedding low. While existing methods exhibit biases when evaluating these embedding qualities, we are interested in an unbiased evaluator. To this end, we explore a general-purpose approach to quantifying the quality of graph embeddings that makes no assumptions over underlying models, distances, or applications. We propose using a neural network as a decoder that predicts the existence of an edge based on the embedding coordinates and use the achieved average precision to measure the quality of an embedding.

Using our implementation, we can distinguish between known good and bad graph embeddings. We also show how graph subsampling reduces the required training time of the decoder by an order of magnitude, while also slightly increasing the average precision.

Zusammenfassung

Graphen werden allgegenwärtig verwendet, um Beziehungen zwischen Objekten und Daten in realen Szenarien darzustellen. Sie werden beispielsweise für Straßennetze, Zitationen wissenschaftlicher Veröffentlichungen, soziale Netzwerke und Abhängigkeiten von Programmcode verwendet.

Relevante Aufgaben auf Graphen wie Kantenvorhersage und Knotenklassifizierung beinhalten die Vorhersage, den Vergleich oder die Visualisierung. Diese Aufgaben können mithilfe von Graph-Einbettungen gelöst werden. Um dies effizient und akkurat durchzuführen, muss die Einbettung so viele Informationen über den Graphen wie möglich erfassen, während die Dimension der Einbettung niedrig gehalten wird. Während die existierenden Methoden dazu neigen, Annahmen über zugrunde liegende Modelle, Kantenlängen oder Anwendungen zu machen, bemühen wir uns um ein möglichst unbefangenes Qualitätsmaß. Zu diesem Zweck untersuchen wir einen allgemein anwendbaren Ansatz zur Quantifizierung der Qualität von Graph-Einbettungen, der diese Annahmen nicht macht. Dafür schlagen wir vor, ein neuronales Netzwerk als Decoder zu verwenden, das darauf trainiert wird, Kanten anhand von Einbettungs-Koordinaten vorherzusagen, und die erreichte Präzision zur Messung der Qualität einer Einbettung zu nutzen.

Mit unserer Implementierung können wir zwischen bekannten guten und schlechten Graph-Einbettungen unterscheiden. Wir zeigen auch, dass Graphen-Subsampling die erforderliche Trainingszeit des Decoders um eine Größenordnung reduziert und gleichzeitig die durchschnittliche Präzision leicht erhöht.

Contents

1 Introduction

1.1 Motivation

Graphs are an efficient method of representing relationships between objects and data in real-world scenarios [Slu14 | CZC18]. They are, for example, used for road networks, citations of scientific publications, social networks, and program code dependencies [Sal+21 | CZC18]. While social researchers might be interested in discovering friendships in social networks, biologists use graphs to model and analyze the interaction patterns of proteins [BNRF21 | Yue+19]. These tasks can be approached using *graph embeddings*. Graph embeddings compress large and complex graphs into low-dimensional and fixed-size vectors [CZC18 | Dev22]. We use graph embeddings to predict, compare, and visualize the characteristics of graphs and their nodes. To perform these tasks efficiently, the embedding has to capture as much information about the graph as possible while keeping the dimension of the embedding low.

That raises the question *how we quantify how good an embedding is.* There are several methods to measure the quality of an embedding. Maximum likelihood estimation (MLE), for example, assumes that the graph emerged from an underlying geometry according to some rule defined by a model. Given an embedding of the graph, we measure the probability of the graph emerging again when using the embedded coordinates. The disadvantage is that we do not know the underlying model of real-world graphs (if one even exists) [BFKL16]. Another method, the edge-length histogram, measures the length of edges and non-edges. We then count how many edges are longer than non-edges. It assumes that distances are the main criterion for adjacency, which may not always be the case [BFK21]. Otherwise a downstream task could evaluate the embedding based on the task performance [ZZP20]. But all these methods cannot be used on arbitrary embeddings without making any prior assumptions, and therefore without any bias. Theoretically, as long as the relevant information of the graph structure is captured by the embedding, we can perform tasks like link prediction (for discovering relations) and classifications (for labeling nodes) with confidence assuming that we know how to interpret it. In this thesis we explore general-purpose approaches to quantifying the quality of graph embeddings that make no assumptions about underlying models, distances, or applications.

We propose using a *neural network* (NN) as a *decoder* and using the performance of the decoder to measure the quality of an embedding. The decoder is trained on the graph and embedding to predict if two nodes are adjacent. But using a NN provides its own challenges. First and foremost, neural networks require ample computational effort. Training the decoder naively takes time proportional to the number of nodes in the graph squared. Especially for graphs with millions of nodes, as in social networks like Twitter [Deg23], the training time quickly grows into infeasible scales. This raises the second question, *how can we speed up the decoder*? Additionally, the datasets generated from real-world graphs have an inherently large class imbalance. The amount of edges is considerably lower than the amount of non-edges. Theoretically, a NN that never predicts an edge to exist can easily reach a high prediction accuracy. We look at how graph subsampling can increase the performance of the decoder in both speed and accuracy. Lastly, NNs are black boxes. It is hard to tell why we get certain

outputs. So when we receive bad results, it is unclear whether the embedding was bad or the NN performed poorly. We therefore also look at how the decoder performs on known good and bad embeddings to visualize its effectiveness.

We implement our findings as a Python framework[1]. Our results show that the decoder is able to distinguish known good and bad graph embeddings accurately. Additionally, we show that training the decoder on subgraphs generated with our alteration of the random jump algorithm reduces the required training time of the decoder by an order of magnitude, while also slightly increasing the average precision.

1.2 Outline

First, the preliminaries declare the notation and concepts of the thesis. It explains graphs and graph embeddings, neural networks, the torus, random geometric graphs, and geometric inhomogeneous random graphs. In Chapter 3 we explain our framework that implements our research questions as well as its important hyperparameters. In Chapter 4 we present our results with the framework in its different configurations. In Chapter 5 we conclude our results and point out possible future work.

[1] See the implementation at *https://github.com/HydrofinLoewenherz/embedding-eval-framework*.

2 Preliminaries

In this chapter we give general information about the notation and concepts used in the thesis. We describe graphs and embeddings, as well as neural networks, and a selection of graph types.

2.1 Graphs and Embeddings

We denote a *graph* $G = (V, E)$, with a set V of *nodes* and a set E of *edges*. An edge is a pair $\{u, v\} = e \in E$ of nodes $u, v \in V$. Two nodes are *adjacent* if there exists an edge between them. The *size* $n = |V|$ of a graph is the number of nodes in the graph. The *degree* k of a node u is the number of edges $|\{e : \{u, v\} = e \in E\}|$ that connect u with another node. A *sparse graph* is a graph with only $\Theta(n)$ edges. In contrast, a general graph can have up to n^2 edges.

An *induced graph* $G[V']$ of a graph G and a node set $V' \subset V$ is defined as $G[V'] = (V', E')$ with $E' = \{e : \{u, v\} = e \in E \text{ and } u, v \in V'\}$.

A *path* $P = (w_0, \dots w_i)$ is a collection of nodes $w_0, \dots, w_i$ for that $\forall_{1 \leq j \leq i} \{w_{j-1}, w_j\} \in E$. We call a graph *connected* if, for any pair of nodes u, v, there exists a path $P_{uv} = (u, \dots, v)$ between them. A *graph component* $G' = G[V']$ is a subgraph induced by a node set V' such that $G[V']$ is connected, while $G[V' \cup \{v\}]$ for any $v \in V \setminus V'$ is not.

A graph embedding assigns a feature vector $f_u \in F = \mathbb{R}^d$ to every node u, with d being the dimension of the embedding.

2.2 Neural Networks

An *artificial neural network* (ANN) or *neural network* (NN) is a *machine learning* (ML) model inspired by the networks of biological neurons. One of the simplest ANN architectures is the perceptron, modelling only a single artificial neuron. A *perceptron* takes inputs $(x_0, x_1, \dots, x_c) = x$, computes the weighted sum, and applies an *activation function*, for example a step or sigmoid function [Gér19]. The activation function is chosen depending on the task, the NN should perform. A perceptron with weights $w = (w_0, \dots, w_i)$, w_b, and bias b can be described as

$$h_{w,b}(x) = \text{act}(\sum_{0 \leq i \leq c} w_i \cdot x_i) + w_b \cdot b = \text{act}(x^\top w) + w_b \cdot b. \tag{2.1}$$

Adjusting these weights to achieve the desired output is the core element of the training process.

Usually, multiple perceptrons are bundled in parallel into layers, which are then stacked on top of each other. The perceptrons' outputs in one layer are fed into the perceptrons of the next. Layers are typically *fully connected*, meaning that the output of each perceptron of one layer is fed into every other perceptron in the next layer. The operation performed by a single layer can be described as a matrix multiplication. Graphical processing units (GPUs) are especially suited for this task and are hence often used to train and operate NNs.

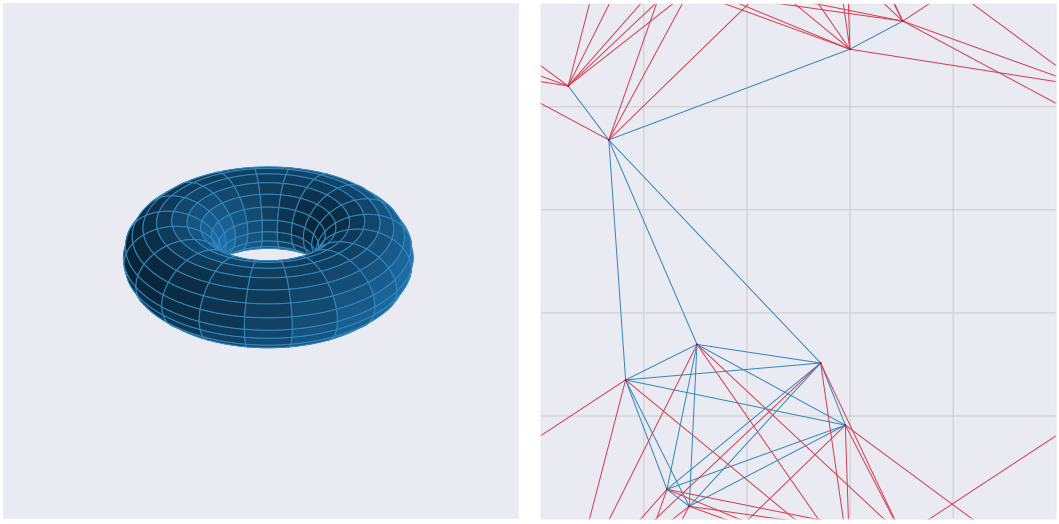

Figure 2.1: The plot shows a three-dimensional visualization of a torus on the left and a graph on a flat torus on the right. The graph edges in red wrap around the opposite side.

A *feed forward network* (FNN) is a type of ANN in which the information in the network moves only in one direction. That means, that the output of a layer is not fed back into itself, nor any other layer before itself, which may be done in more complex architectures such as *recurrent neural networks* (RNN). The resulting network consists of the first layer (input layer), a number of intermediate layers (hidden layers), and the last layer (output layer). A NN that has multiple hidden layers is called a *deep neural network* (DNN). Feeding a feature vector into a DNN can be expressed as applying multiple matrix multiplications in succession.

A DNN is trained by feeding it a feature vector $x = (x_0, x_1, \ldots, x_c)$ to produce an output vector. A *loss function* then scores the output vector against the desired output vector. Using the *backpropagation* algorithm [RHW86], the gradients of the loss function with respect to all weights are computed. The weights are then slightly adjusted, following the gradient, to reduce the result of the loss function. The relative amount by which the weights are adjusted is described by the *learning rate* and may further be controlled using dedicated *optimizers* (e.g. ADAM [KB17]). By repeating this process multiple times on a whole dataset of input vectors, the loss function is minimized, with the process also being known as *stochastic gradient descent* (SGD).

A dataset for ML is typically split into three disjoint sets, the *train set*, the *validation set*, and the *test set*. The train set is directly used to train the NN. The validation set is used to validate the quality of the training while the NN is trained. It is used to update hyperparameters, such as the number of *epochs*. Lastly, the test set is used to test the generality of the NN after training. For training the NN, it is typically presented with the whole train set in mini-batches of multiple input vectors each epoch. SGD is applied after each mini-batch.

We generally want to prevent the NN from overfitting and losing generality. An overfitted NN gives good scores on data it has been trained on but is bad at labeling previously unseen data.

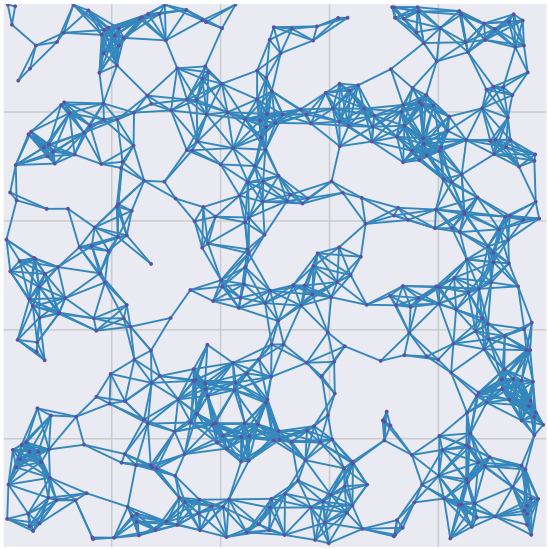

Figure 2.2: The plot shows an RGG with 500 nodes and a degree of 10.

2.3 Torus

A torus is a hypercube $[0, 1]^d$ with the opposite sides being identified. We see an example of a two-dimensional torus in Figure 2.1. On the left, we see a three-dimensional visualization of the two-dimensional torus as a donut. On the right, we see a two-dimensional visualization of the torus, called a *flat torus*. The graph on its surface shows how the edges (in red) wrap around the opposite side.

2.4 Random Geometric Graphs (RGG)

Random Geometric Graphs (RGGs) are undirected graphs that resemble real-world networks [Pen03]. They can be used to model ad hoc networks [Nek07]. RGGs are constructed by placing n nodes on a unit square at random. Two nodes are adjacent if their distance is smaller than some threshold r. An example can be seen in Figure 2.2. We approximate[1] the average node degree k for a graph of size n and a threshold radius r on a unit square to be $k = (|V| - 1) \cdot (\pi \cdot r^2)$. We can rearrange to

$$r = \sqrt{\frac{k}{(|V| - 1) \cdot \pi}} \,. \tag{2.2}$$

If we choose a constant target for the average degree, the created graph has $(n \cdot k)/2 = \Theta(n)$ edges. The generated graph is therefore sparse.

We define the feature vector for a node of an RGG with position $p = (p_x, p_y)$ as

$$f = \begin{bmatrix} p_x & p_y \end{bmatrix}^\top . \tag{2.3}$$

[1] Nodes close to the boundary of the square have a lower node degree

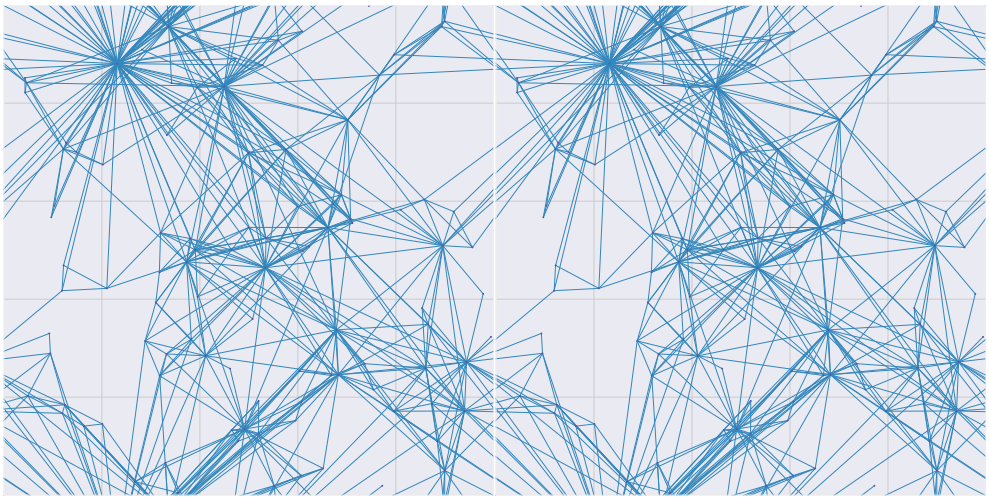

Figure 2.3: The plot shows a GIRG with 100 nodes and a degree of 10 duplicated next to itself. By showing it next to itself, we see the wrapping characteristic of the flat torus.

2.5 Geometric Inhomogeneous Random Graphs (GIRG)

Geometric Inhomogeneous Random Graphs (GIRGs) are undirected graph structures that approximate complex real-world networks [BKL19]. The adjacency of nodes is not only defined by their proximity, but also by weighting of individual nodes. They have a power-law degree distribution and high clustering [BKL19]. They can be generated efficiently programmatically [Blä+19]. We see an example of GIRGs in Figure 2.3, visualized on a torus.

We chose GIRGs as an example of complex real-world networks because we know the ground-truth embedding of the graph, allowing us to evaluate the framework on good embeddings resembling real-world networks.

We define the feature vector for a node of a GIRG with position $p = (p_x, p_y)$ and weight w as

$$f = \begin{bmatrix} p_x & p_y & w \end{bmatrix}^\top . \tag{2.4}$$

3 Framework

The basic approach of the framework[1] is that based on a graph and its embedding in a coordinate system, a decoder is trained to predict if two nodes are adjacent, given only their features. Essentially, we train the decoder to reconstruct the graph given its embedding in a feature space. We then measure the quality of the embedding by how good the reconstruction of the decoder is compared to the original graph. The idea is that if the embedding efficiently reflects the graph, the decoder should be able to reconstruct the graph. Should the embedding not capture all necessary information about the graph or capture too many unnecessary features, the decoder might struggle or even be unable to create a good reconstruction.

In the following sections we describe key features of the framework, such as dataset generation, subsampling, and quality quantification.

3.1 Feature Preprocessing

Feature preprocessing is an important step to achieve good classification performance in ML exercises. While deep learning does not depend on feature preprocessing as much as other ML exercises, it still benefits from it. Dataset *normalization* is one approach where the dataset feature vectors are either scaled or transformed to give each feature an equal contribution [SS20]. As this framework is built to make no assumptions, we use the *standardization* transformation, also called *z-score normalization*. It normalizes each feature row to have a mean of 0 and a standard deviation of 1. In Equation (3.1), we see three feature vectors (left) and their standardization (right) with the last feature row highlighted in red. First, we calculate the mean m and the standard deviation s for each row and then transform all entries f in a row with their respective standard deviation and mean such that $f' = {}^{(f-m)}/_{s}$.

$$\begin{bmatrix} 0.252 \\ 0.816 \\ 1.125 \end{bmatrix}, \begin{bmatrix} 0.489 \\ 0.094 \\ 0.919 \end{bmatrix}, \begin{bmatrix} 0.851 \\ 0.75 \\ 0.957 \end{bmatrix} \xrightarrow{standardize} \begin{bmatrix} -1.131 \\ 0.87 \\ 1.393 \end{bmatrix}, \begin{bmatrix} -0.17 \\ -1.4 \\ -0.908 \end{bmatrix}, \begin{bmatrix} 1.301 \\ 0.53 \\ -0.485 \end{bmatrix} \tag{3.1}$$

3.2 Dataset Generation

We want the decoder to be able to decide if two nodes $u, v \in V$ are adjacent, given their feature vectors $f_u, f_v \in F$. For this, we denote a data point as a tuple $(\{f_u, f_v\}, l)$ with feature vectors $f_u, f_v \in F$ and a label $l \in \{0, 1\}$. The label is set to 1 if u and v are adjacent, otherwise we set the label to 0, resulting in two classes, the 1-labeled class, and the 0-labeled class. Our classification task is therefore a *supervised binary classification*, as we know the labels of our data (supervised) and want to decide whether two nodes are adjacent or not (binary classification).

[1] See the implementation at *https://github.com/HydrofinLoewenherz/embedding-eval-framework*.

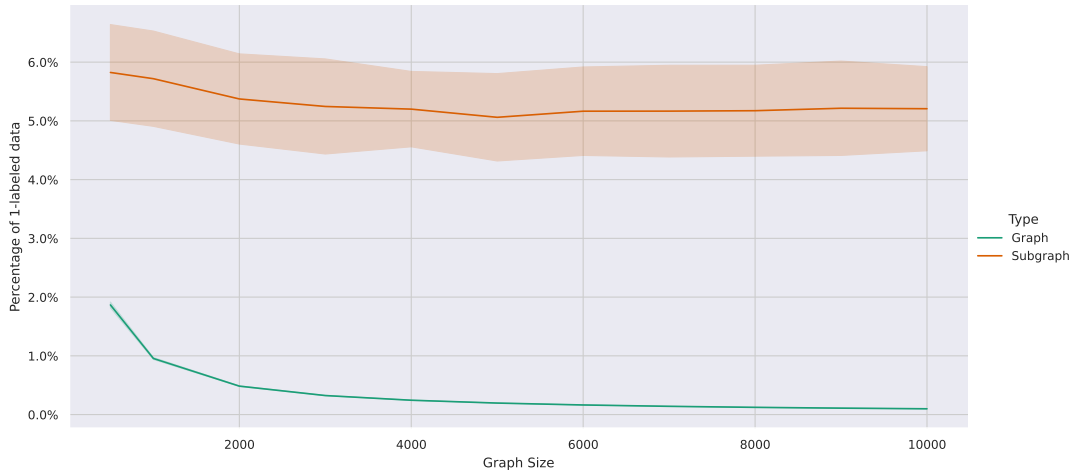

Figure 3.1: The plot shows the percentage of 1-labeled data for an RGG (green) and a subgraph (orange) for different graph sizes over 100 iterations. The RGG has an average node degree of 10, and the subgraph is generated with RJ*-subsampling at $\alpha = 0.0$. The percentage of 1-labeled data for the graph approaches 0% with increasing graph size. For the subgraph, it is steadily higher at around 5%.

A NN expects a single vector as its input. We therefore concatenate the feature vectors f_u, f_v of a data point such that

$$f_{uv} = \begin{bmatrix} f_{u_1} & \dots & f_{u_n} & f_{v_1} & \dots & f_{v_n} \end{bmatrix}^\top. \tag{3.2}$$

With n nodes, we have up to n^2 inputs for the decoder, one for each combination $u, v \in V$. We reduce the amount of inputs by skipping one of each pair f_{uv}, f_{vu}. For this, we have two options, we either skip every f_{uv} if $f_{uv} < f_{vu}$, or we skip one of the two at random. But even after skipping one of each f_{uv}, f_{vu} and all self-loops f_{uu}, we still have $\Theta(n^2)$ inputs.

$$\frac{n(n-1)}{2} = \Theta(n^2). \tag{3.3}$$

We also notice an increasing class imbalance with increasing graph size. For sparse graphs, the amount of 1-labeled data points is only $\Theta(n)$. The remaining $\Theta(n^2) - \Theta(n) = \Theta(n^2)$ are 0-labeled. We can see in Figure 3.1 how the percentage of edges in the graph (green) approaches 0% with increasing graph size. In Section 3.4 we explain how to mitigate this issue.

3.3 Dataset Splitting

As described before, we have to split the dataset into the train set, validation set, and test set. We split the dataset by subsampling the graph and subtracting the sample from the graph. First, we subsample the test set and then the validation set. The remaining graph is used as the train set. Generally, we use the same subsampling algorithm for splitting the dataset as for subsampling each epoch, as described in the following section.

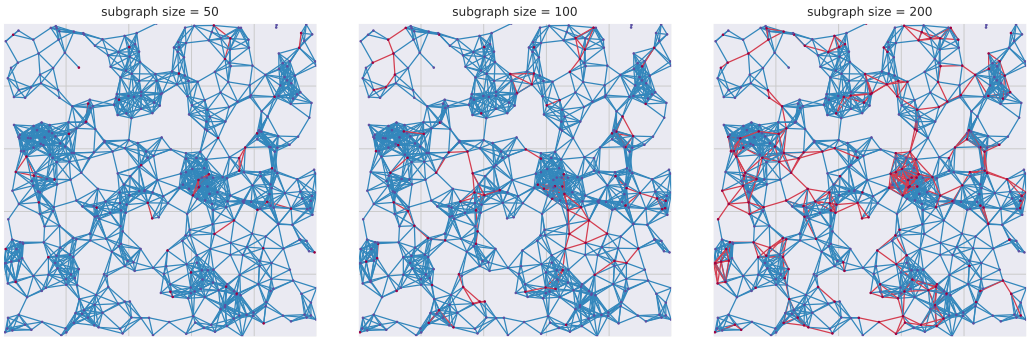

Figure 3.2: The plot shows WRS subgraphs on an RGG with 500 nodes for different subgraph sizes. The nodes and edges in red represent the subgraph. We see that smaller subgraphs have a relatively low number of edges compared to the larger subgraphs. The chance of two adjacent nodes to be chosen at random is considerably lower with a smaller subgraph size and a larger graph size.

3.4 Epoch Graph Subsampling

In a common ML setup, the NN trains on the complete train set in each epoch. As described in Section 3.2, this results in $\Theta(n^2)$ inputs for each epoch. For graphs with millions of nodes, the time needed to train the decoder would become infeasible.

Subsampling graphs is a commonly used method to speed up processing [Sal+21]. We therefore generate a new fixed-size subgraph for each epoch. We sample a subset of nodes $V' \subset V$ and use the induced subgraph $G[V']$ as the dataset for that epoch.

By subsampling the graph before generating the dataset, we can limit both the number of inputs and the class imbalance. We see in Figure 3.1 that the percentage of edges in the subgraph (orange), obtained by subsampling, is generally higher and steadier, here at around 5%, while the percentage of edges in the graph (green) approaches 0% with increasing graph size. We can subsample with different algorithms, e.g.

- Weighted Random Sampling (WRS)
- Breadth-first search (BFS) and Depth-first search (DFS)
- Random Walk (RW)
- Random Jump (RJ)
- Random Jump Star (RJ*)

In the following sections we give an overview of how these algorithms may be implemented and used for subsampling in this framework.

3.4.1 Weighted Random Sampling (WRS)

Weighted Random Sampling (WRS) assigns a weight to each node in the graph. It then chooses nodes at random with a probability that is proportional to their weight. Higher weighted nodes have a higher chance to be sampled. We could also assign the weights to the dataset directly, however this is out of scope for this thesis.

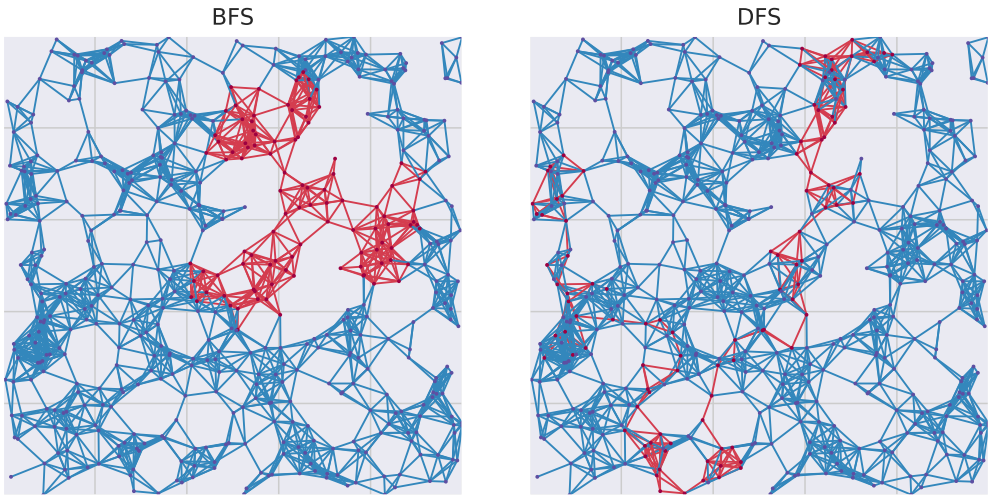

Figure 3.3: The plot shows two subgraphs on an RGG generated with a BFS (left) and DFS (right). The subgraph nodes and edges are highlighted in red.

Based on the weights assigned, the sampled subgraph has different properties. In general, we want to assign weights, that are easy to compute, to minimize the overhead [Sal+21]. The easiest weights are uniform weights, that is, the same weight for all nodes. The resulting subgraph is easily sampled but shows low connectivity as seen in Figure 3.2. Low connectivity means a higher class imbalance and is therefore not preferred. Additionally, as we can see in Figure 3.2, the connectivity strongly correlates with the subgraph size relative to the graph size. The connectivity in the left-most subgraph is considerably lower than on the right, where the subgraph size is higher. To use WRS effectively, we have to choose a subgraph size dependent on the graph size.

3.4.2 Breadth-first search (BFS) and Depth-first search (DFS)

Breadth-first search (BFS) and depth-first search (DFS) are search algorithms that generate trees from graphs. We use these algorithms to generate subgraphs by limiting the number of nodes visited to the target subgraph size and using the nodes of the tree as the subset V'.

The DFS generates line-shaped subgraphs, while the BFS-generated subgraphs are disc-shaped. We see this in Figure 3.3. Although both subgraphs are generated on the same graph, the BFS subgraph contains many more edges than the DFS subgraph.

A problem we encounter with BFS and DFS is, that the root node of the tree has to be on a graph component that has at least the target subgraph size. The algorithms are not able to jump between graph components[2].

[2]BFS and DFS are able to detect this and could be altered to use additional tree roots.

Algorithm 3.1: Graph subsampling with RW

Input: Graph G and target size s
Output: $V' \subset V$ with $|V'| = s$

```
V_visited ← []
α ← 0.15
choose v ∈ V at random
v_s ← v
while |V_visited| < s do
    if v ∉ V_visited then
        append v to V_visited
    choose r ∈ [0, 1] at random
    if r ≥ α then
        v ← v_s
    else
        v ← v_n ∈ V neighbor of v at random
return V_visited
```

Algorithm 3.2: Graph subsampling with RJ

Input: Graph G and target size s
Output: $V' \subset V$ with $|V'| = s$

```
V_visited ← []
α ← 0.15
choose v ∈ V at random
while |V_visited| < s do
    if v ∉ V_visited then
        append v to V_visited
    choose r ∈ [0, 1] at random
    if r ≥ α then
        v ← v_r ∈ V at random
    else
        v ← v_n ∈ V neighbor of v at random
return V_visited
```

3.4.3 Random Walk (RW)

Random Walk (RW) is an algorithm to traverse graphs. We choose a starting node v_s at random and traverse along its edges at random. At every step, with a probability α, we jump back to the starting node instead of walking an edge. It can be implemented as shown in Algorithm 3.1.

One issue of RW is that we can no longer detect if there are any nodes reachable from the current node that have not been traversed yet. If v_s is part of a graph component that is smaller than the target size s, the algorithm will end up in an infinite loop.

3.4.4 Random Jump (RJ) and Random Jump Star (RJ*)

Random Jump (RJ) is similar to RW, but instead of going back to the starting node v_s with a probability α, a random node is chosen instead. This resolves the problem of RW getting stuck on a graph component smaller than the target size, but decreases connectivity of the subgraph. It can be implemented as shown in Algorithm 3.2.

Note that if we set the jump probability $\alpha = 0.0$, the algorithm no longer performs jumps and gets stuck on small graph components. To resolve this, we introduce an additional element, the (relative) boredom threshold β. If the algorithm does not visit a node not previously traversed in $\beta \cdot s$ steps, it jumps to a node in the graph at random. This adjustment has the additional advantage that we can choose small values of α without having to worry about the runtime. We call this alteration Random Jump Star (RJ*). It can be implemented as shown in Algorithm 3.3. The jump probability α allows us to influence the number of disconnected (or loosely connected) components in the subgraph. Setting the jump probability to $\alpha = 0.0$ gives us a method close to the base implementation of RW that shows structures similar to BFS/DFS. Setting it to $\alpha = 1.0$ essentially produces a WRS with uniform weights. It allows us to smoothly transition between the two extremes.

Algorithm 3.3: Graph subsampling with RJ*

Input: A graph G and a target size s
Output: A subset V' of V with $|V'| = s$

$V_{visited} \leftarrow []$; $\alpha \leftarrow 0.15$; $\beta \leftarrow 0.90$; $b \leftarrow 0$
choose $v \in V$ at random
while $|V_{visited}| < s$ **do**
    if $v \notin V_{visited}$ **then**
        append v to $V_{visited}$
        $b \leftarrow 0$
    else
        $b \leftarrow b + 1$
    choose $r \in [0, 1]$ at random
    if $r \geq \alpha$ **or** $b > \beta \cdot s$ **then**
        $v \leftarrow v_r \in V$ at random
    else
        $v \leftarrow v_n \in V$ neighbor of v at random
return $V_{visited}$

Depending on the structure of the original graph that is sampled, we can choose different settings for the jump probability α and subgraph size s. We see these subgraph structures in Figure 3.4 for both RGGs and GIRGs. Low values of α (left) result in subgraphs with more edges, while higher values of α (right) have fewer edges but a broader graph coverage.

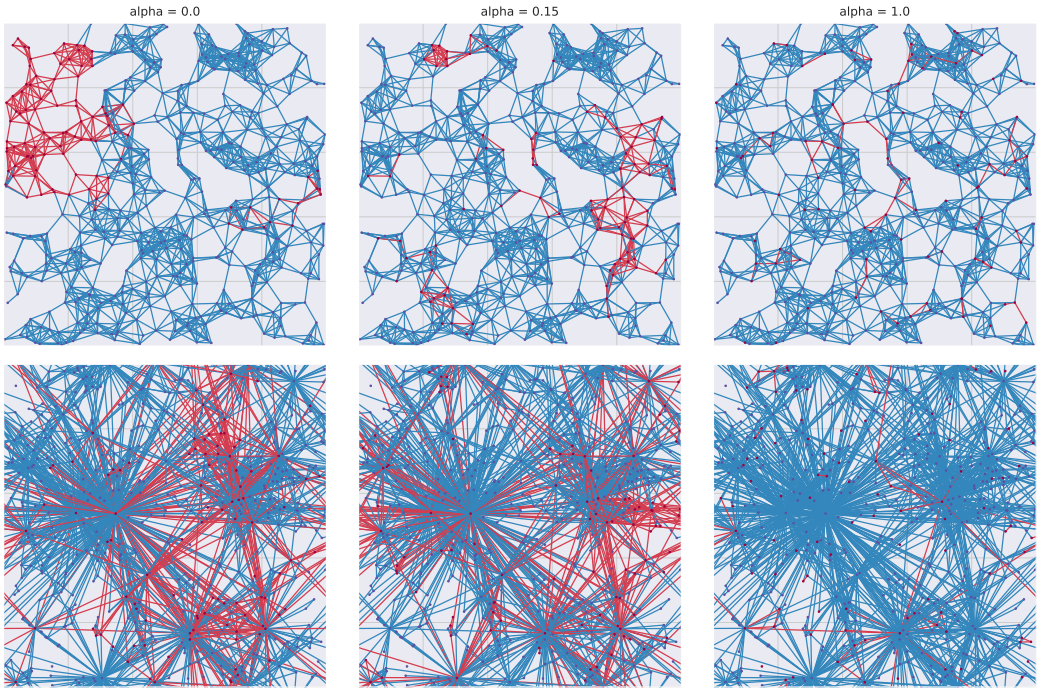

Figure 3.4: The plot shows an RGG (top) and GIRG (bottom) with subgraphs generated for different jump probabilities α at 0.0 (left), 0.15 (center), and 1.0 (right). The RGG is set to have an average degree of 10. The red nodes and edges represent the generated subgraphs.

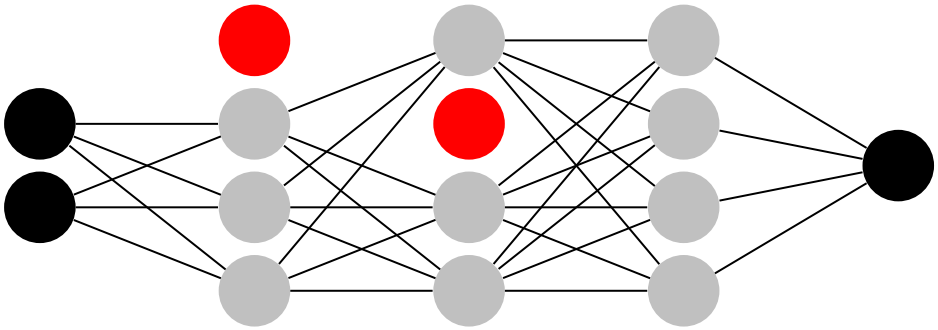

Figure 3.5: The plot shows a simplification of the NN used in the framework. The NN has two input perceptrons on the left, three hidden layers (gray) with each consisting of four perceptrons, and one output perceptron on the right. The layers are fully connected, excluding the perceptrons in red that have been dropped by the dropout layers.

3.5 Neural Network Structure

We use a FNN in the framework. It consists of one input layer, three hidden layers, and one output layer, with each layer being fully connected. Each hidden layer has 32 perceptrons. The activation function is a *rectified linear unit* (ReLU) function. The input layer size is fixed to $|f_{uv}|$, while the output layer consists of only one perceptron for the binary classification. By applying a sigmoid function to the last perceptron, a probability for binary classification is produced.

To decrease the risk of overfitting, we integrate *dropout layers* [Sri+14]. The dropout layers randomly set some layer inputs to zero with a set probability of 20% while training, preventing the NN from co-adapting too much (ensuring that the perceptrons work independently of each other) [Sri+14]. Figure 3.5 shows a simplification of the NN, where the perceptrons in red have been set to zero by the dropout layers.

We use *binary cross entropy with logits*[3] as our loss function and the *ADAM* algorithm as our optimizer [KB17].

3.6 Quality Quantification

Because we have a high class imbalance, we chose the *precision-recall curve* to measure our decoder performance as opposed to, for example, a *ROC curve*, which is more suited for datasets with low class imbalance. The precision-recall curve visualizes the trade-off between *precision* and *recall* for different *decision thresholds*. A high precision relates to a low false positive rate, and a high recall relates to a low false negative rate.

$$P = \frac{T_p}{T_p + F_p} \qquad R = \frac{T_p}{T_p + F_n} \tag{3.4}$$

[3] See PyTorch *https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html*.

With the true positive rate T_p, true negative rate T_n, false positive rate F_p, and false negative rate F_n. The decision threshold $t \in [0, 1]$ is used to decide if a prediction $p \in [0, 1]$ indicates either a 1-label or a 0-label. We use all individual probability values produced by the decoder as decision thresholds. The *average precision* (AP) summarizes the precision-recall-curve, for the different precision-recall pairs at the different decision thresholds t_i. As such, we use the AP as our decoder performance score.

$$\text{AP} = \sum_{i \in I} (R_i - R_{i-1}) P_i \tag{3.5}$$

For generating a reconstruction we have to decide on one decision threshold. We choose to use the threshold that maximizes the *F1-score*. The F1-score is the harmonic mean of precision and recall.

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{3.6}$$

3.7 Early Stopping

Early stopping is commonly used to prevent the NN from overfitting on the train set and losing generality. We detect overfitting by evaluating the NN after every epoch on the validation set. Should the validation loss not improve in a set amount of epochs the training is stopped and the NN state that scored the best on the validation set is restored.

Additionally, we stop the training early if the average precision on the validation set reaches 99%. Some embeddings and input parameters are able to reach scores above 99%, with any further improvement not being worth the additional training time.

4 Results

In this chapter we explore results that have been generated by the framework. The main questions we answer are how the framework performs on *good embeddings*, how the framework performs on *bad embeddings*, and how *subsampling* affects the framework results. All our trainings were conducted on an *NVIDIA GeForce RTX 2060 SUPER* GPU.

4.1 Good Embeddings

As described before, we are interested in how the framework performs on good embeddings. By looking at the framework's performance on good embeddings, we are able to better interpret the results of embeddings of unknown qualities. We look at two graph types with ground truth embeddings, RGGs and GIRGs. They have ground truth embeddings, in that the information used to decide if two nodes are adjacent is already associated with the nodes (see Sections 2.4 and 2.5). To get a general idea about the performance of the decoder for these two embeddings, we perform a grid search for different graph sizes and configurations for RJ*-subsampling, namely the subgraph size and jump probability α.

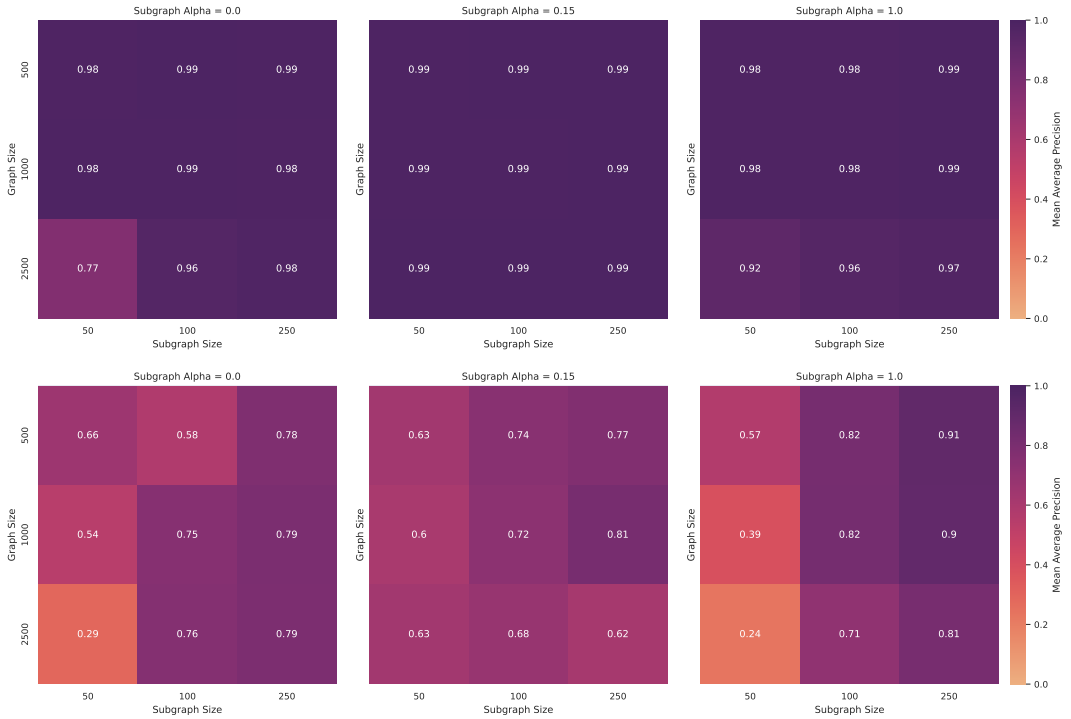

Figure 4.1: The plot shows the mean average precision for different graph sizes, subgraph sizes, and alphas over five repetitions. The RGG (top) and GIRG (bottom) both have an average degree of ten. The subgraph is generated with RJ*-subsampling.

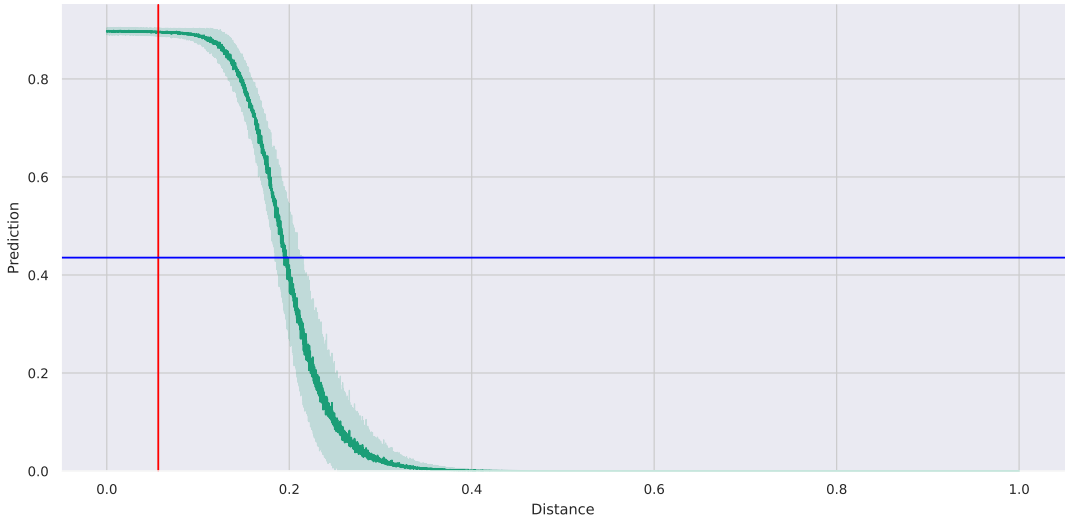

Figure 4.2: The plot shows the mean prediction (green) for node pairs with different distances over 100 repetitions. The NN is trained on an RGG with a threshold radius of around 0.056. The radius can be seen as the red vertical line. The blue horizontal line is the optimal threshold that maximizes the f1-score.

We see in Figure 4.1 (top) that the framework often achieves a high average precision of around 99%. For large differences between graph and subgraph size a drop to $70 - 80\%$ can be observed. By looking closely, we also notice how the results for $\alpha = 1.0$ are slightly worse than their counterparts for $\alpha = 0.0$ and $\alpha = 0.15$. We see in Figure 4.7 that the mean average precision reaches the 99% threshold relatively quickly. This shows that the decoder is able to efficiently learn RGG embeddings.

Next, we are interested to see how the framework performs on more complex graphs like GIRGs. We see in Figure 4.1 (bottom) how the results spread to a wider range. As seen for RGGs, the mean average precision for graph size 2 500 and subgraph size 50 show the worst results, dropping to around $30 - 60\%$ and therefore even lower than for RGGs. Additionally, the whole column with subgraph size 50 shows considerably worse results. Generally, the decoder is still able to learn GIRG embeddings though.

From the plots we can deduct that RJ*-subsampling performs best with a jump probability $\alpha = 0.15$.

4.1.1 RGG threshold

After seeing the results from the grid search, we are interested in what the decoder actually learned. For RGGs, two nodes are adjacent if their distance is smaller than a predefined radius. We visualize what the decoder has learned by letting it predict the labels of node pairs of different distances. We expect that the decoder predicts every pair with a distance smaller than the radius as adjacent and all pairs with distances larger than the radius as nonadjacent.

In Figure 4.2, we see the decoder predictions (green) for different node pair distances, as well as the RGG radius (red) and a decision threshold (blue) for classifying. The decoder was able to learn that node pairs with distances below a particular threshold are 1-labeled, while the rest are 0-labeled. The decision threshold (blue) shown predicts more node pairs to be adjacent than they actually are.

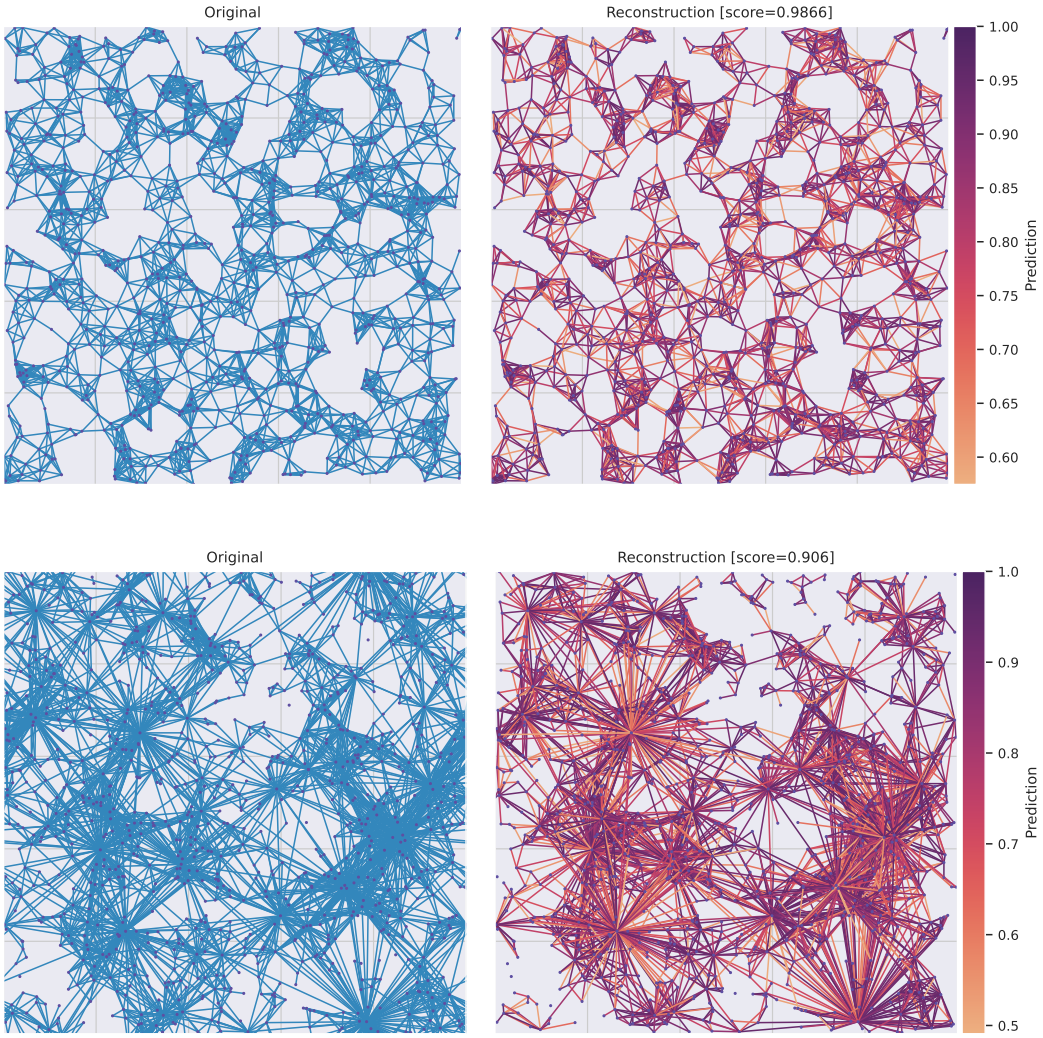

Figure 4.3: The plot shows a graph reconstruction for both an RGG and a GIRG. Note that the scales and thresholds for the predictions differ.

4.1.2 Reconstruction

We use the decision threshold described in Section 3.6 to visualize a graph reconstruction. In Figure 4.3 we see a reconstruction for both an RGG and a GIRG. Both reconstructions show large similarities with their original graph. For the RGG reconstruction, similar to what we saw in Figure 4.2, the decision threshold results in edges with lengths larger than the original radius to be generated. The reconstruction therefore contains more edges than the original. We see similar results for the GIRG reconstruction.

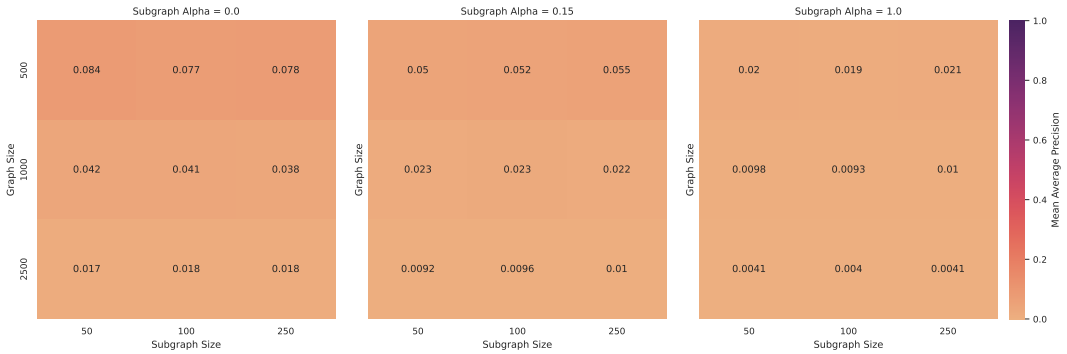

Figure 4.4: The plot shows the average precision for different graph sizes, subgraph sizes, and alphas, averaged over five repetitions. The graph is generated by taking an RGG with an average degree of ten and replacing the node features with random features of the same size. The subgraph is generated with RJ*-subsampling. We observe that not a single configuration reaches a mean average precision of over 10%. RJ* with a jump probability of $\alpha = 0.0$ achieves compared to the other alphas the best score. This is because, without jumps, the generated subgraph has more edges.

4.2 Bad Embeddings

After seeing how the framework is able to score good embeddings and reconstruct the graph with high precision, we are interested in how the framework performs on bad embeddings. To get a bad embedding, we use a graph embedding that is known to yield good scores like RGGs, but replace the node features with random features. The resulting embedding does not represent the original graph in any way. As such, the decoder should not be able to generalize.

We see in Figure 4.4 how the framework yields considerably lower scores for random graph embeddings in comparison to RGG and GIRG embeddings (compare Figure 4.1). Not a single configuration reaches a mean average precision of over 10%. We conclude that the framework is not able to learn the random embedding, which can be used to distinguish bad embeddings.

4.3 Epoch Subsampling

A central component of the framework is epoch subsampling. We already explored how subsampling improves the runtime and performance in theory. The idea is that the subsampled graph used as the epoch dataset is considerably smaller than the complete train set while also having a lower class imbalance (see Section 3.4). But how does subsampling compare to no subsampling at all? In Figure 4.5 we see the runtime and average precision results for different graph sizes, both with and without subsampling. Notably, the runtime for RJ*-subsampling is much lower compared to no subsampling. RJ*-subsampling results in a runtime of around 100 to 500 seconds, while without subsampling the runtime reaches 3 000 seconds (around 50 minutes). Additionally, the average precision, as seen in the lower plot, is generally higher with subsampling. We see an average improvement of around 10%.

Note that the framework depends on subsampling for splitting the dataset. For the experiment with no subsampling, we still have to use RJ*-subsampling to split the dataset.

In conclusion, we see that epoch subsampling leads to a considerable improvement in speed and precision.

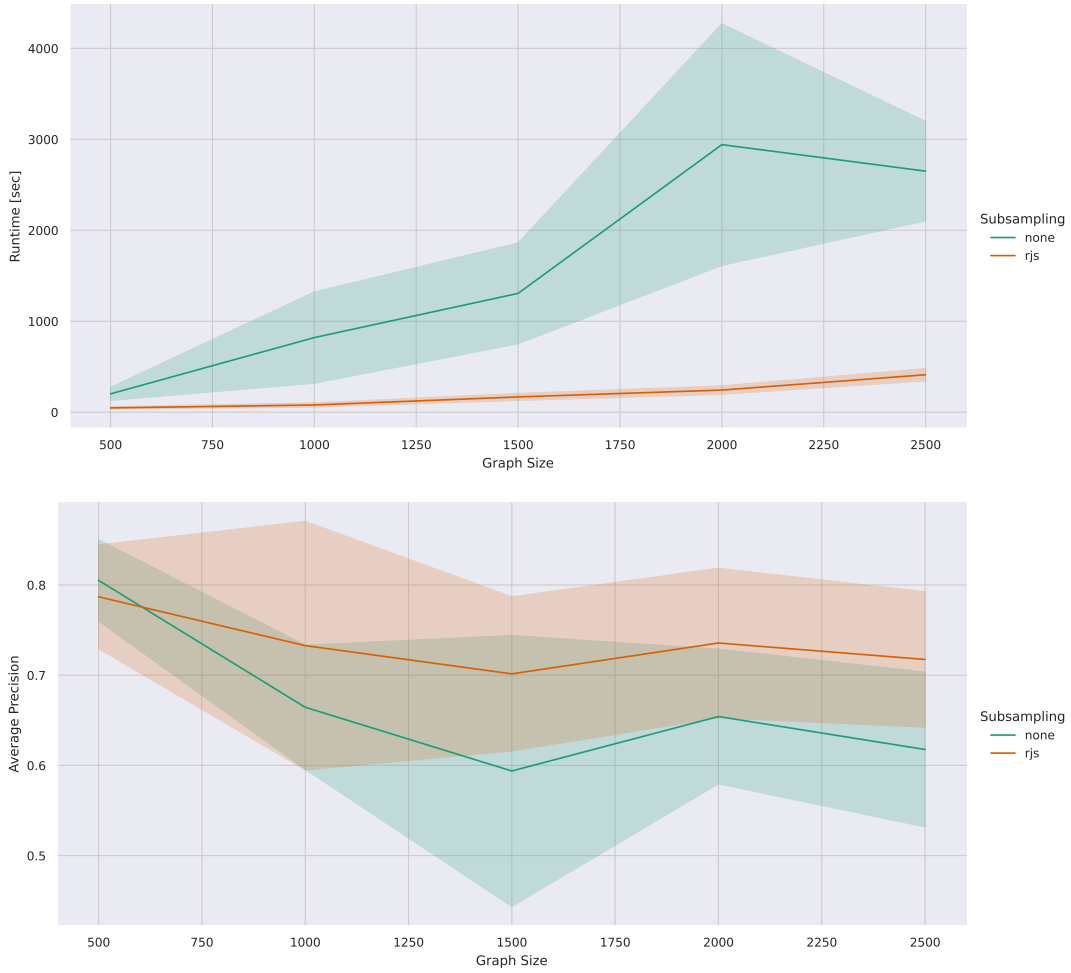

Figure 4.5: The plot shows the average runtime and precision for different graph sizes with and without subsampling over five repetitions. The graph is a GIRG with default configuration except for its size. The RJ*-subsampling uses a fixed subgraph size of 100 nodes and $\alpha = 0.15$. We see how both the runtime and average precision increases considerably when we subsample with RJ* (`rjs`) compared to no subsampling (`none`). Note that in the plot for the average precision the y-axis does not start at zero.

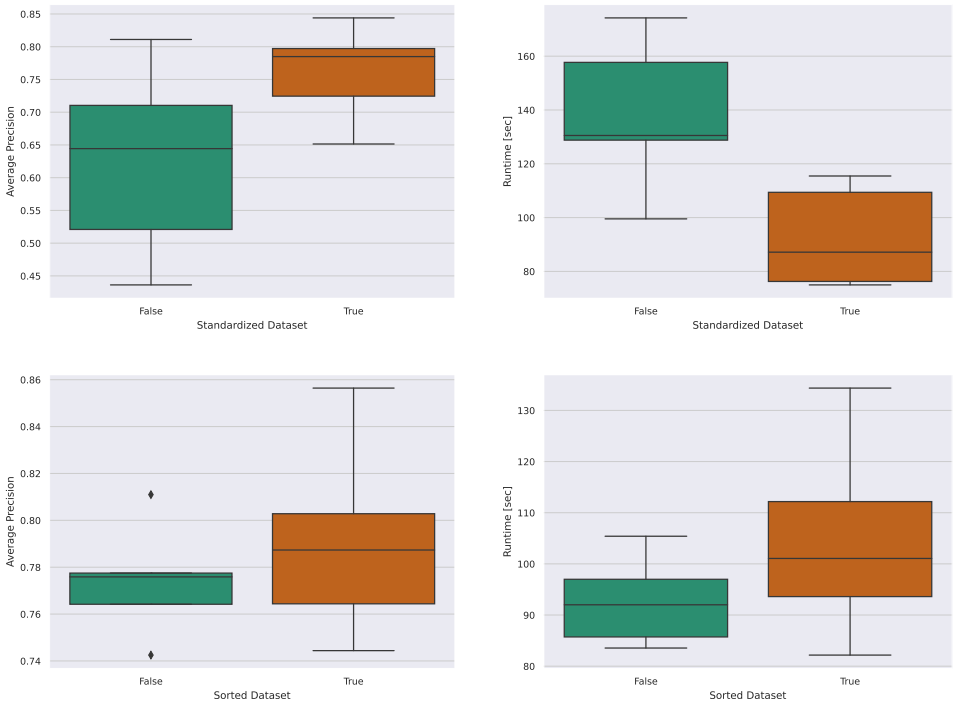

Figure 4.6: The plot shows the average precision (left) and runtime (right) on a GIRG with and without dataset standardization (top) and dataset sorting (bottom) over five runs. The GIRG is created with 1 000 nodes and a degree of 10. The subgraph is generated with RJ* subsampling with $\alpha = 0.15$ and a subgraph size of 100 nodes. Note that the plot y-axis does not start at 0.

4.4 Dataset Standardization

In Section 3.1 we explained how normalization through feature standardization should improve the framework performance. Normalization can improve framework performance for embeddings with features that follow different ranges, as normalization removes the need to generalize for varying ranges. One such example are GIRG embeddings. GIRG embeddings assign each node its position $p \in [0, 1]^d$ and an arbitrarily large weight $w \in \mathbb{R}$, making it hard for the NN to use both together effectively. In Figure 4.6 (top), we can see how the decoder average precision increases for the standardized dataset, while reducing its runtime.

4.5 Dataset Sorting

In Section 3.1 we also explained how dataset sorting could improve the framework performance. We see in Figure 4.6 (bottom) how sorting the dataset impacts the performance compared to randomly choosing, slightly increasing precision at the cost of also slightly increased runtime. It therefore seems that sorting the dataset presents no justifiable reason to pursue. We therefore do not sort the dataset[1] and instead pick one of the two f_{uv} or f_{vu} at random for the dataset.

[1]Dataset sorting is disabled for all other framework tests.

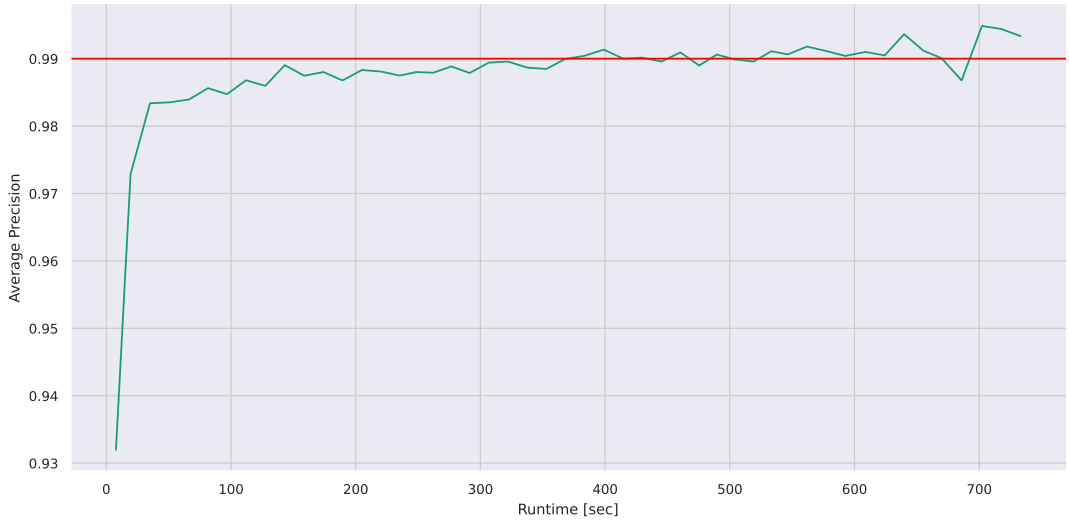

Figure 4.7: The plot shows the mean average precision per runtime in seconds, combined for 20 repetitions and averaged over 50 bins on the validation set, while training (threshold stopping disabled). The line in red shows the threshold 99%. The test is performed on RGGs with 1 000 nodes using RJ* subsampling, with subgraph size 500 and $\alpha = 0.15$. The plot increases in roughness with increasing runtime, as traditional early stopping is still in effect and fewer repetitions reached that runtime. Note that the plot y-axis does not start at 0.

4.6 Early Stopping

In Section 3.7 we showed that early stopping by threshold should speed up the framework train time while only slightly diminishing the average precision. We see in Figure 4.7 how the framework performs without the threshold at 99% in place. The mean average precision reaches the aforementioned 99% after around 380 seconds, whereas the NN continues training for up to 700 seconds while only improving slightly.

5 Conclusion

In this thesis we explored how deep decoders allow us to measure the quality of graph embeddings without making any assumptions about underlying models, distances, or applications. Additionally, we looked into how epoch subsampling improves the performance of such decoders.

We have implemented a Python framework that encapsulates our research questions and allows us to evaluate and compare different configurations. We tested how different graph types, subgraph types, dataset preprocessors and decoder structures affect the framework performance in both speed and precision.

In our first set of experiments, we looked at the scores achieved by different graphs and subgraphs. We tested both known good (RGG and GIRG) and bad (random) embeddings with different configurations of our RJ*-subsampling algorithm. Our results indicate that the framework and its decoder are able to effectively distinguish good embeddings from bad embeddings. Good embeddings achieved high scores, while bad embeddings scored poorly. Additionally, we determined that our subsampling algorithm RJ* performs best on average with a low jump probability. Feature preprocessing with normalization also shows considerable performance improvements for the framework. We also visualized what the decoder actually learns by plotting the decoder prediction for different node pair distances on RGGs. We see that the decoder is able to learn the threshold characteristic of RGGs.

The framework also includes utilities to visualize the framework predictions into graph reconstructions. We see how the framework is able to reconstruct graphs accurately.

Our results present a proof-of-concept showing that the quality of graph embeddings can be measured with deep decoders without making assumptions. In conjunction with the Python framework it is a foundation for future work to further explore how DNNs can be used in evaluating graph embeddings.

Future work could alter the NN by testing combinations of loss and activation functions, the composition of layers, and feature preprocessors. Other decision thresholds could also be used to achieve different desired results. Additional graphs and embeddings of known quality should be tested (e.g., node2vec [GL16]) to provide a better understanding of the framework itself. Furthermore, it would be interesting to see how the framework performs on much bigger graphs with millions of nodes. We could introduce weights to nodes that are used by the subsampling algorithm to decide which nodes should be explored first, like using the node degree or even the inverse of the node degree. Lastly, real-world data is not perfect. It would be helpful to know how the framework performs on incomplete and inaccurate data.

Bibliography

[BFK21] Thomas Bläsius, Tobias Friedrich, and Maximilian Katzmann. "Force-Directed Embedding of Scale-Free Networks in the Hyperbolic Plane". In: *19th International Symposium on Experimental Algorithms (SEA 2021)*. Edited by David Coudert and Emanuele Natale. Vol. 190. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 22:1–22:18. ISBN: 978-3-95977-185-6. DOI: *10.4230/LIPIcs.SEA.2021.22*.

[BFKL16] Thomas Bläsius, Tobias Friedrich, Anton Krohmer, and Sören Laue. "Efficient Embedding of Scale-Free Graphs in the Hyperbolic Plane". In: *24th Annual European Symposium on Algorithms (ESA 2016)*. Edited by Piotr Sankowski and Christos Zaroliagis. Vol. 57. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 16:1–16:18. ISBN: 978-3-95977-015-6. DOI: *10.4230/LIPIcs.ESA.2016.16*.

[BKL19] Karl Bringmann, Ralph Keusch, and Johannes Lengler. "Geometric inhomogeneous random graphs". In: *Theoretical Computer Science* Volume 760 (2019), pp. 35–54. ISSN: 0304-3975. DOI: *https://doi.org/10.1016/j.tcs.2018.08.014*.

[Blä+19] Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. "Efficiently Generating Geometric Inhomogeneous and Hyperbolic Random Graphs". In: *27th Annual European Symposium on Algorithms (ESA 2019)*. Edited by Michael A. Bender, Ola Svensson, and Grzegorz Herman. Vol. 144. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 21:1–21:14. ISBN: 978-3-95977-124-5. DOI: *10.4230/LIPIcs.ESA.2019.21*.

[BNRF21] Kamal Berahmand, Elahe Nasiri, Mehrdad Rostami, and Saman Forouzandeh. "A modified DeepWalk method for link prediction in attributed social network". In: *Computing* Volume 103 (Oct. 2021), pp. 2227–2249. ISSN: 1436-5057. DOI: *10.1007/s00607-021-00982-2*.

[CZC18] HongYun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. "A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications". In: *IEEE Transactions on Knowledge and Data Engineering* Volume 30 (2018), pp. 1616–1637. DOI: *10.1109/TKDE.2018.2807452*.

[Deg23] J. Degenhard. "Number of Twitter users worldwide from 2018 to 2027". MathWorld. Mar. 12, 2023. URL: *https://www.statista.com/forecasts/1146722/twitter-users-in-the-world* (visited on 04/19/2023).

[Dev22] Google Developers. "Embeddings". Google. July 18, 2022. URL: *https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture* (visited on 12/05/2023).

[Gér19] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow*. O'Reilly Media, Inc., 2019. ISBN: 9781492032649.

[GL16] Aditya Grover and Jure Leskovec. "node2vec: Scalable feature learning for networks". In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 855–864.

[KB17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: *1412.6980*.

[Nek07] Maziar Nekovee. "Worm epidemics in wireless ad hoc networks". In: *New Journal of Physics* Volume 9 (June 2007), p. 189. DOI: *10.1088/1367-2630/9/6/189*.

[Pen03] Mathew Penrose. *Random Geometric Graphs*. Oxford University Press, May 2003. ISBN: 9780198506263. DOI: *10.1093/acprof:oso/9780198506263.001.0001*.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* Volume 323 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: *10.1038/323533a0*.

[Sal+21] Guillaume Salha, Romain Hennequin, Jean-Baptiste Remy, Manuel Moussallam, and Michalis Vazirgiannis. "FastGAE: Scalable graph autoencoders with stochastic subgraph decoding". In: *Neural Networks* Volume 142 (2021), pp. 1–19.

[Slu14] David J Slutsky. "The effective use of graphs". en. In: *J. Wrist Surg.* Volume 3 (May 2014), pp. 67–68.

[Sri+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* Volume 15 (2014), pp. 1929–1958.

[SS20] Dalwinder Singh and Birmohan Singh. "Investigating the impact of data normalization on classification performance". In: *Applied Soft Computing* Volume 97 (2020), p. 105524. ISSN: 1568-4946. DOI: *https://doi.org/10.1016/j.asoc.2019.105524*.

[Yue+19] Xiang Yue, Zhen Wang, Jingong Huang, Srinivasan Parthasarathy, Soheil Moosavinasab, Yungui Huang, Simon M Lin, Wen Zhang, Ping Zhang, and Huan Sun. "Graph embedding on biomedical networks: methods, applications and evaluations". In: *Bioinformatics* Volume 36 (Oct. 2019), pp. 1241–1251. ISSN: 1367-4803. eprint: *https://academic.oup.com/bioinformatics/article-pdf/36/4/1241/48983324/bioinformatics_36_4_1241.pdf*.

[ZZP20] Zhiqiang Zhong, Yang Zhang, and Jun Pang. "NeuLP: An End-to-End Deep-Learning Model for Link Prediction". In: *Web Information Systems Engineering – WISE 2020*. Edited by Zhisheng Huang, Wouter Beek, Hua Wang, Rui Zhou, and Yanchun Zhang. Cham: Springer International Publishing, 2020, pp. 96–108. ISBN: 978-3-030-62005-9.