APPLICATION OF GRAPH NEURAL NETWORKS ON SOFTWARE MODELING

A Thesis Submitted to
the Graduate School of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Onur Yusuf LEBLEBİCİ

ACKNOWLEDGEMENTS

First, I would like to thank my supervisors, Assoc. Prof. Dr. Tuğkan Tuğlular and Prof. Dr. Fevzi Belli, who advised, motivated and shared their experiences with me. Their guidance helped me in all the time of research and writing of this thesis.

I am grateful to my sister-in-law Assoc. Prof. Dr. Gonca Özçelik Kayseri for her support.

Finally, I would like to thank my wife Sinem Leblebici for her unconditional and constant support in this process, and my dear daughters Bade and Öykü, whom I deprived of the time we spent together in that period, for their patience and support. This thesis is dedicated to my beloved family

ABSTRACT

APPLICATION OF GRAPH NEURAL NETWORKS ON SOFTWARE MODELING

Deficiencies and inconsistencies introduced during the modeling of software systems can cause undesirable consequences that may result in high costs and negatively affect the quality of all developments made using these models. Therefore, creating better models will help the software engineers to build better software systems that meet expectations. One of the software modelling methods used for analysis of graphical user interfaces is Event Sequence Graphs (ESG). The goal of this thesis is to propose a method that predicts missing or forgotten links between events defined in an ESG via Graph Neural Networks (GNN). A five-step process consisting of the following steps is proposed: (i) data collection from ESG model, (ii) dataset transformation, (iii) GNN model training, (iv) validation of trained model and (v) testing the model on unseen data. Three performance metrics, namely cross entropy loss, area under curve and accuracy, were used to measure the performance of the GNN models. Examining the results of the experiments performed on different datasets and different variations of GNN, shows that even with relatively small datasets prepared from ESG models, predicts missing or forgotten links between events defined in an ESG can be achieved.

ÖZET

GRAFİK YAPAY SİNİR AĞLARININ YAZILIM MODELLEMESİNE UYGULANMASI

Yazılım sistemlerinin modellemeleri sırasında yapılan eksiklikler ve oluşan tutarsızlıklar, bu modeller kullanılarak yapılan tüm geliştirmelerde de yüksek maliyetlerle sonuçlanan istenmeyen sonuçlara sebep olabilmektedir. Yazılım modellemesi sırasında yazılım mühendislerine verilebilecek öneriler ile daha iyi modeller oluşturulabilir ve bu sayede kullanıcı beklentilerini daha iyi karşılayan sistemler oluşturulabilir. Yazılım modellemede kullanılan yöntemlerden bir taneside, grafik kullanıcı arayüzlerinin analizinde kullanılan olay akış grafikleridir. Bu tezin hedefi olay akış grafikleri üzerinde yer alan bileşenler arasında unutulmuş veya eksik bağlantıları grafik yapay sinir ağları kullanarak tahminleyecek bir yöntem öndermektir. Bu yöntem beş basamaktan oluşan bir süreçten oluşmaktadır: (i) veri toplama, (ii) grafik yapay sinir ağ modelini eğitmek, (iii) eğitilen modeli doğrulamak ve (v) modeli daha önce görmediği veriler ile test etmek. Eğitilen grafik yapay sinir ağ modellerinin performansını ölçmek için çapraş entropi kaybı, eğri altında kalan alan ve doğruluk performans metrikleri kullanılmıştır. Farklı veri kümeleri ve farklı grafik yapay sinir ağı varyasyonları ile yapılan deneylerin incelenmesi sonucunda, nispeten küçük ölçekli veri kümelerinde dahi başarı elde edilebildiği gözlemlenmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

Figure **Page**

Figure **Page**

LIST OF TABLES

Table **Page**

CHAPTER 1
INTRODUCTION

Software applications are generally sophisticated. Those systems may require many sub-systems and components in it. In order to design a software system, details of the system are required to be understood at varying levels. Typical software modeling approaches may not be able to reduce this complexity for engineers. Predicting and giving recommendations on the connections between software components such as classes, events, UI elements, user interactions etc. has great importance in modelling. From the software engineering perspective, deciding and connecting components with each other require significant effort. It is also error prone. Instead of putting all the workload on the software engineer, providing some recommendation can help engineers with modeling the composition and interaction of components in a software system. There are many different models and tools used in software modeling. Most of these models are graph-based models and there is a well-established theory of graph transformations [1], which has a number of system modelling and software engineering applications. The graph-based modeling technique taken under consideration in this thesis is Event Sequence Graphs (ESGs) [2].

GUIs can be modeled as sequences of events of the objects defined in GUI. The operations on components of the GUI, such as buttons, lists, and checkboxes, are controlled and/or observed by input/output devices. Thus, an event can be a user input or a system response; both of them are elements of event set V and lead to a sequence of user inputs and expected desirable system outputs. An ESG is a tuple (V, E, Ξ, Γ), where $V \neq \emptyset$ is a finite set of nodes (vertices or events) and $E \subseteq V \times V$ is a finite set of arcs (edges), and $\Xi, \Gamma \subseteq V$ finite sets of distinguished vertices with $\xi \in \Xi$, $\gamma \in \Gamma$, called entry nodes (start events) and exit nodes (finish events), respectively [2]. The semantics of an ESG is as follows. Any $v \in V$ represents an event. For two events $v_1, v_2 \in V$, the event v_2 must be enabled after the execution of v_1 if and only if $(v_1, v_2) \in E$ [3]. ESG is chosen in this thesis as modeling technique because of its simple semantics.

There are many applications of graph-structured data, such as finding friends on social network [4], [5], molecule interactions in medicine [6], highlighting the product which a customer is interested in e-commerce [7], relation learning for knowledge graphs [8], extrapolating paths [9], metabolic network reconstructions [10]. Graph-structured

data are different from linear-structured data. Graph-structured data cannot be given to neural networks by traditional methods as in linear-structured data. The applications of neural networks on graph-structured data, is getting attraction in recent years. In 2009, Scarselli et al. proposed a method [11], which makes it possible to work neural networks on graphs. After this proposal, the studies published in this field has come infrequent until 2016, and then the studies started to gain momentum again. With many researches, graph classification, node classification, edge classification studies have been carried out using different types of neural network architectures [12][13][14][15]. Some of the architectures are as follows, recurrent neural network (RNN), convolutional neural network (CNN), autoencoders (AE), attention mechanism.

Another important application of graph neural network is link prediction. Zhang and Chen proposed a next generation approach for link prediction called Weisfeiler-Lehman Neural Machine [16], which learns the patterns on graphs to predict links. That was a game changing approach. They used fully connected neural network for learning the link existence patterns on graphs. After that Zhang and Chen improved their method and proposed deep graph convolutional neural network (DGCNN) for link prediction [17]. They extracted local enclosing subgraphs for each vertex and applied DGCNN to predict edges. Another approach is based on the representation of connections between nodes is due to the relationship between the features of the nodes, these relationships can be learned through graph neural networks (GNN). Gu et al. proposed a method [18] based on that idea an used graph attention network [15] (GAT) as GNN variation.

One of the methods used for link prediction is heuristic methods. This method, which is based on assumptions such as having common neighbors, can be used to suggest friends in social networks, but does not show any success in predicting molecule connections[6] or extrapolating paths [9]. ESGs are considered analogous to molecules, where events are like atoms and their different combination means a new model.

In this thesis, an application of graph neural network (GNN), which predicts missing or unnecessary links between events defined in an ESG, is introduced. For an ESG, a link means a transition between two events. Experiments were performed on four different datasets with two different GNN variations to predict links that have not been seen before. The following steps were followed in the experiments: data collection, data transformation, model training, validation of the trained model, and measurement of the performance.

The motivation in this thesis is to help software engineers during system modeling. With the developed approach, errors that may arise from models will be prevented or reduced and quality will be increased. Since these models are used for coding, testing and design in the software development processes later, and any deficiencies and errors may occur in this process can cause very high costs. Modeling quality directly affects the quality of the system.

The outline of the thesis is as follows. Chapter 2 provides fundamental information about terms and terminologies used in this thesis. Chapter 3 gives an overview of preliminary research on link prediction using GNN. Chapter 4 introduces steps of the developed approach with, Chapter 5 presenting the evaluations over different datasets and different GNN models using proposed approach. The last Chapter provides conclusions and possible future works.

CHAPTER 2
FUNDAMENTALS

2.1. Neural Networks

Machine learning is used to find patterns in data represented by numbers. Neural networks are models that learn the nonlinear relationship between input and output. The basic methods used to train neural networks can be listed as follows.

Supervised learning: It is a learning method in which the expected output based on the given input is predefined and these definitions are used to train the model. In other words, this method is used to generate a function that produces desired outputs with respect to the given inputs.

Unsupervised learning: This learning method tries to learn groups according to the characteristics of the given data. There is no labeling process for data, only data. They are used to solve problems such as clustering, dimensionality reduction.

Semi-Supervised learning: This learning method is used in cases where only a certain amount of data is labeled in the existing data set. Labeled data used through the supervised learning, while the un-labeled data used through the unsupervised learning. The purpose here is to guess the labels of un-labeled data.

Reinforcement learning: The main purpose in this learning method is to win the game. A method called policy is used, in which the agent reacts according to the environment and receives feedback from the environment. The agent tries different actions each time and must learn to make the best choice according to the reward-penalty system.

Problem domains handled by neural networks can be listed as follows.

Classification: The classification problem occurs when one or more labels needed to be generated as output. Neural network model makes predictions based on previous observations. The purpose of a neural network model specialized for classification is to approximate the function that describes the discrete outputs based on the given input values. Predicting the plant species can be given as an example of such problems.

Regression: The regression problem occurs when continuous value needed to be generated as output. Neural network model makes predictions based on previous

observations. The purpose of a neural network model specialized for regression is to approximate the function that describes the continues numeric outputs based on the given input values. Calculating the risk ratio for insurance can be given as an example for such problems.

Clustering: It is the grouping of data with similar characteristics in a data set. There are many similarities among clusters created by unsupervised learning, but less similarities between different clusters.

2.2. Neural Network Models

Feed Forward Neural Networks (FFNN) and Multilayer Perceptron [19] [20]**:** This kind of neural network models consist of layers and these layers are named as input, hidden and output. Models consist of 1 input, 1 output layer and 0 or more hidden layers. Generally, each layer fully connected to the next. They feed information from input to output. The simplest model is logic gate and it has two input cells and one output cell.

Convolutional Neural Networks (CNN) [21]**:** They are different from other artificial neural networks. Although they are generally used for image classification purposes, there are many different usage areas. CNN processes a given input by passing it through the following steps; there are convolution layer, non-linear function and pooling layer those are arranged one after the other. After passing through those layers, data pass through the final fully connected layer and generates numbers on output layer that explain the possibility of being a certain class. The Convolution layer is the basic building component of CNNs where the most calculations are made. In the conversion of the convolution layer, each learnable filter with size $n \times n \times k$ (where n is height-width and k is depth) is slide (with a given factor) on the input data. The filter and the input data segment at the current position and size of the filter are subjected to matrix operation and an activation map is created as output. The point is that learnable filters are activated when they see certain patterns. ReLU is widely used as a non-linear function. In the pooling layer, the method of max-pooling is also widely used.

Recurrent Neural Networks (RNN) [22] [23]**:** They are neural network models that uses outputs of the previous layers as the input of the next layers, recurrently. Although RNN models are generally used in natural language processing, there are many other application areas. RNNs have different types according to the number of inputs and

outputs. For example, one-to-many (one input-multiple output) source code generation, many-to-one (multiple input-one output) sentiment classification, many-to-many (multiple input-multiple output) language translation. RNNs are difficult to train due to vanishing and exploding gradient problems. The difficulties arising from these problems have been reduced with the Long Short Term Memory (LSTM) [24].

Autoencoders (AE) [25]**:** Autoencoders represents compressed knowledge of the original input data and used for representation learning tasks. They learn the hidden representation of the input data in an unsupervised manner. The architecture of autoencoders consists of hidden layers that are symmetrically structured between input and output layers.

Generative Adversarial Networks (GAN) [26]**:** These are models in which two different networks, one generative and the other discriminative, work together. The generative network in the model generates data, the discriminative network consumes this generated data. Training process of this models as follows, while the purpose of the generative network is to fool the discriminative network, the purpose of the discriminative network is to ensure if the given input real or fake. The problem encountered in this model is that the tuning process for both networks is handled separately. Because of that the performance of one of the two models is lower than the other, affects the overall performance of the system.

Graph Neural Networks (GNN) [11]**:** Graph structured data are different from data in matrix or vector structures. When each data kept in matrix or vector structure considered as a cell, changing the order of these cells breaks the integrity of the data. In contrast, graph structured data is isomorphic. The nodes defined on the graph and the links between these nodes contain very valuable data. GNNs are neural networks specialized for learning over graph structed data.

2.3. Neural Network Hyper Parameters

Epoch: Giving the entire training set to the neural network model once is called one epoch. The number of epochs expresses how many times the training set will pass through to the neural network model.

Batch Size: It represents the amount of data that will be used in each iteration in neural network trainings. If batch size is equal to all dataset, it is called batch-mode, if it

is greater than one and less than the whole dataset, it is called mini-batch, and if it is one, it is called stochastic-mode.

Dropout: It is the random removal of the connections of neurons in MLP layers at a given rate. The main purpose of the parameter is to prevent the overfitting (model learns the training data but cannot makes predictions on unseen data).

Number of Hidden Units: It expresses, how many neurons will be defined in a hidden layer.

Learning Rate: On neural networks, parameter values are updated via backpropagation process. Parameter value modification of a perceptron is performed by finding the difference by taking derivatives and multiplying the difference with the given learning rate. This parameter is a tuning parameter of optimization algorithms.

2.4. Metrics to Evaluate Neural Networks

One of the most crucial part working with neural network is evaluation of the model performance. Most of the time accuracy metric is used to measure the performance of the model, but it is not enough just using one metric. Some of the metrics used to evaluate neural network models can be listed as follows.

Confusion Matrix: It is a table that shows the total number of inputs, actual outputs and predicted inputs to visualize complete performance. An example is given in Table 2.1. True positive (45), true negative (45), false positive (55) and false negative (55) values can be seen in the given table below. this metric visualizes prediction and expected value made by model.

Table 2.1. A Sample Confusion Matrix

# Input: 100	**Predicted # Class A**	**Predicted # Class B**
Actual # Class A	**30**	**45**
Actual # Class B	**10**	**15**

Classification Accuracy: It is the ratio of the number of inputs given for estimation to the neural network and the number of outputs correctly predicted by the model. In order to provide reliable results, an equal or close to the number of inputs for

each class should be given. In this way, how accurately the trained model can predict is measured. Since GNNs have not been used to predict a software model before, random baseline is used for comparison.

$$Accuracy = \frac{\text{Number Of Correct Predictions}}{\text{Total Number Of Input}}$$

F1 Score: When the distribution of classes in dataset is not balanced, classification accuracy is not a good performance measurement choice. To overcome this issue there are two sub metrics used in F1 score, one is recall which is the ratio of the sum of true positive and false negative to true positive, and the other one is precision which is the ratio of the sum of true positive and false positive to true positive. F1 Score tries to find a balance between precision and recall and combines them into a single metric.

$$Recall = \frac{\text{True Possitive}}{\text{True Possitive} + \text{False Negative}}$$

$$Precision = \frac{\text{True Possitive}}{\text{True Possitive} + \text{False Possitive}}$$

$$F1\ Score = 2 \text{ x } \frac{\text{Pression x Recall}}{\text{Pression} + \text{Recall}}$$

Area Under Curve (AUC): To understand AUC, it is necessary to explain the ROC Curve first. Neural networks assign a certain probability to each class in their outputs, while making class selections, either the class with the highest probability is selected or the classes above a predetermined threshold value are specified as output. As the threshold value changes (increases or decreases) the outputs of the model to be used as predicted classes will naturally change. ROC shows the ratio between true positive rate

and false positive rate for different threshold values as shown in Figure 2.1. Since GNNs have not been used to predict a software model before, random baseline is used for comparison.

$$True\ Positive\ Rate = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

$$False\ Positive\ Rate = \frac{\text{False Positive}}{\text{False Positive} + \text{True Negative}}$$

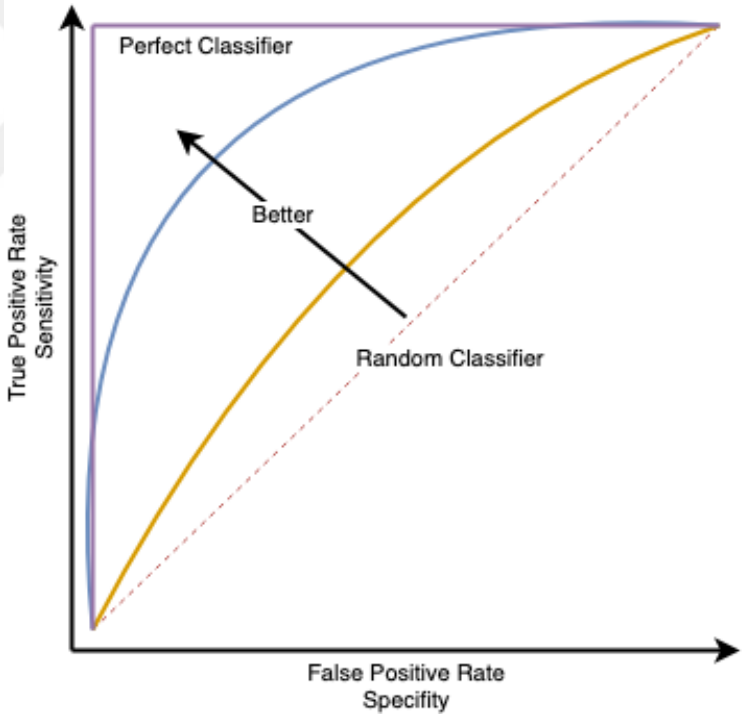


Figure 2.1. ROC Curve

AUC is a metric of performance of a binary classifier on any threshold values, so the metric does not change by threshold. AUC specifies the area under ROC Curve shown in Figure 2.2. The value of AUC is between 0 and 1, and higher values are better.

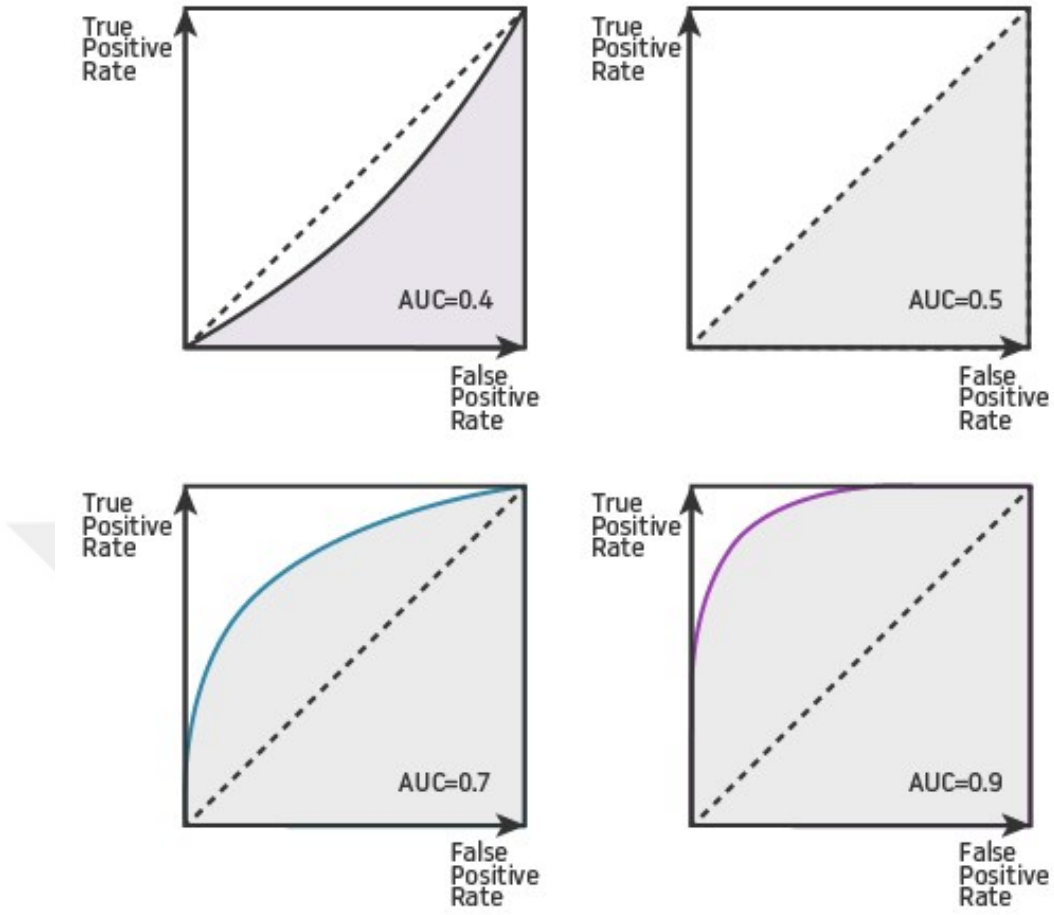


Figure 2.2. Area Under Curve (AUC)

Cross Entropy Loss (Log Loss): It is the measurement of the distance between the values (which are between 0 and 1) produced by the output layer of a neural network, from the expected values. For example, suppose that a neural network model has two units in the output layer, the expected output for this network is 1 for the first unit and 0 for the second unit. If the predicted output is 0.01 for the first unit, which is said to be a high loss value. For the best results, the loss should converge to zero.

Mean Absolute and Squared Error: It is the average distance between predictions and actual outputs only difference between absolute and squared one is, mean squared error takes the square of the distance. This metrics are used to measure accuracy for continuous variables. However, it does not talk about whether the predictions are overestimating or underestimating.

$$Mean\ Absolute\ Error = \frac{1}{N}\sum_{j=1}^{N} | yj - y'j |$$

$$Mean\ Squared\ Error = \frac{1}{N}\sum_{j=1}^{N} (yj - y'j)^2$$

2.5. Graph Types

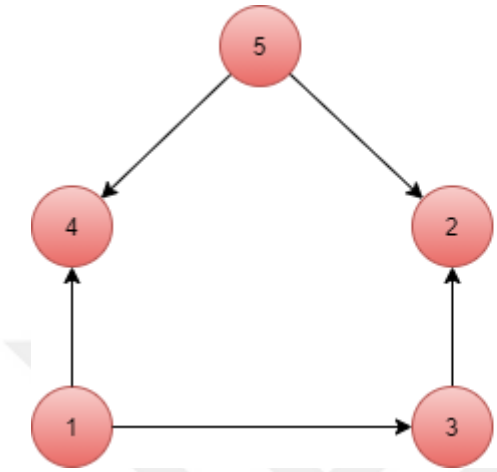
A graph (*G*) is a pair of sets of vertices (*V*) and set of edges (*E*). A vector *n(v)* represents neighbors of vertex *v*. In addition, vertices and edges may have features and/or labels stored in a feature vector. Graphs can be divided into two groups, namely homogeneous and heterogeneous. Moreover, graphs can also be separated as dynamic, static, directed, undirected, weighted, vertex labeled, and edge labeled graphs, given in Table 2.2.

Table 2.2. Graph Types

	Homogeneous	**Heterogenous**
Dynamic Graph		
Static Graph		
Directed		
Undirected		
Weighted		
Vertex Labeled		
Edge Labeled		

In the following, all the items given in Table 2.2. are explained.

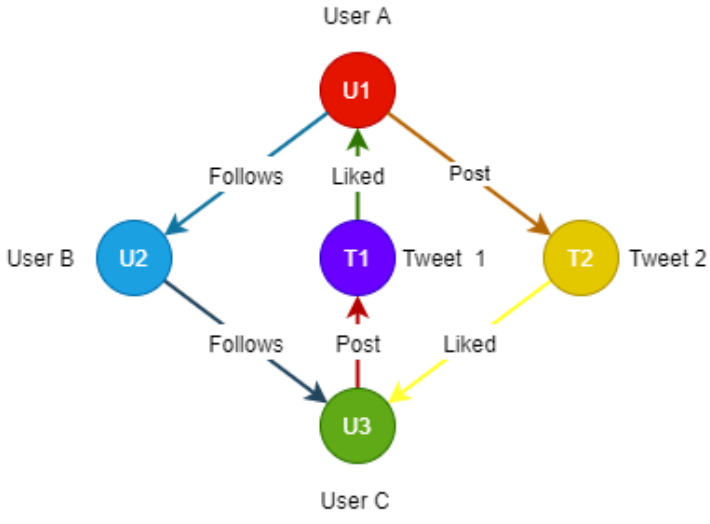
Homogeneous Graphs: All the vertices and all the edges of a homogeneous graph have the same types. Vertices share the same identity space and feature vector as seen in Figure 2.3.



Id	EventType	Duration
1	Save	3
2	Cancel	1
3	Delete	2
4	Read	4
5	Options	1

Figure 2.3. The structure of homogeneous graphs

Heterogeneous Graphs: A heterogeneous graph can have vertices and edges of different types. Vertices/Edges of different types have independent identity space and feature vectors. For example, as illustrated in the Figure 2.4., the user and tweet vertices have different identifiers, and both have different features.



Id	Gender	Age
U1	Female	30
U2	Female	23
U3	Male	27

Id	Length	Category	Hits
T1	100	Movies	7
T2	150	Education	10

Figure 2.4. The structure of heterogeneous graphs

Static Graphs: They are the graphs that do not change over time. In the literature, most of the studies on graph theory is based on static graph structure. A wealth of such literature has been developed for static graph theory [27].

Dynamic Graphs: The authors [27] proposed dynamic graphs, which changes over time. In many science disciplines, dynamic graphs are considered. This is especially true for computer science, where almost always the data structures (modeled as graphs) change as the program runs as shown in Figure 2.5.

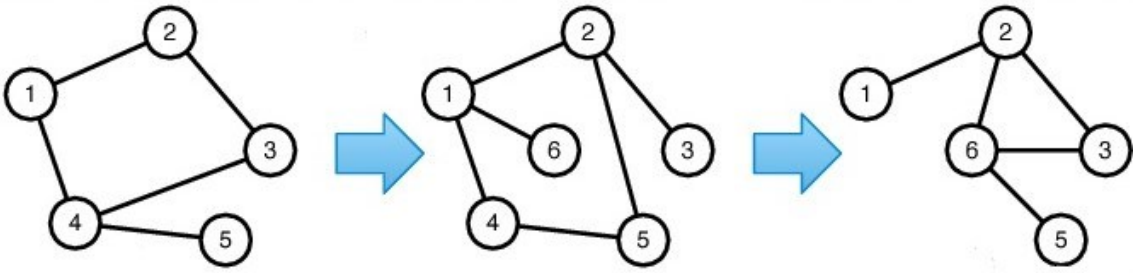


Figure 2.5. The structure of dynamic graph

Edge/Vertex Labeled Graph: Graphs those edges and/or vertices have labels shown in Figure 2.6.

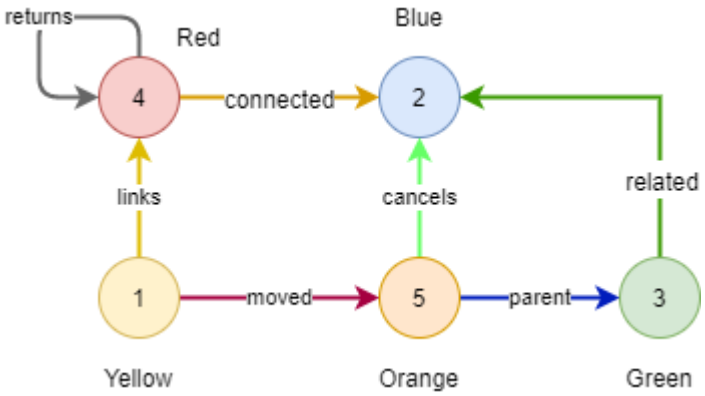


Figure 2.6. The structure of edge/vertex labeled graph

Multigraph: As shown in Figure 2.7, in case that there is more than one type of edge defined between two vertices those kinds of graphs are called multigraph. Each color on edges represents a different type of edge.

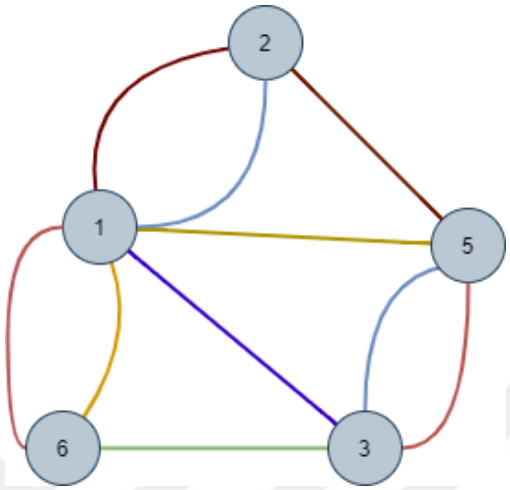


Figure 2.7. The structure of multigraph

Directed Graph: The arrangement of the node pairs is significant in a directed graph. Therefore, *u* is adjacent to *v* only if the pair <*u, v*> is in the edge set. The vertices are usually linked with each other by arrows. An arrow from *u* to *v* is drawn only if <*u, v*> is in the edge set. An example directed graph is shown in Figure 2.8.

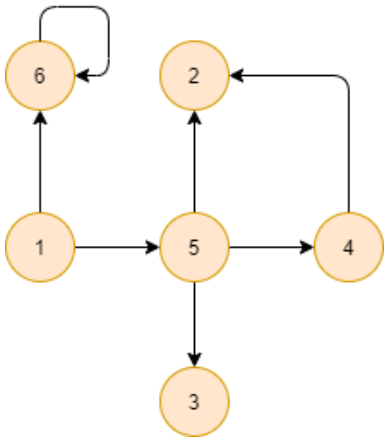


Figure 2.8. The structure of directed graph

Undirected Graph: In undirected graph structure, the arrangement of the vertex pairs in the edge set does not cause any problem. Given graph in Figure 2.9. can be written [{4, 6}, {4, 5}, {3, 4}, {3, 2}, {2,5}, {1,2}, {1, 5}] or [{6, 4}, {5, 4}, {4, 3}, {2, 3}, {5,2}, {2,1}, {5, 1}]. The vertices are usually linked with each other by straight lines. The adjacency matrix is symmetric, so if u ~ v then it is also the case that v ~ u where ~ represents being connected.

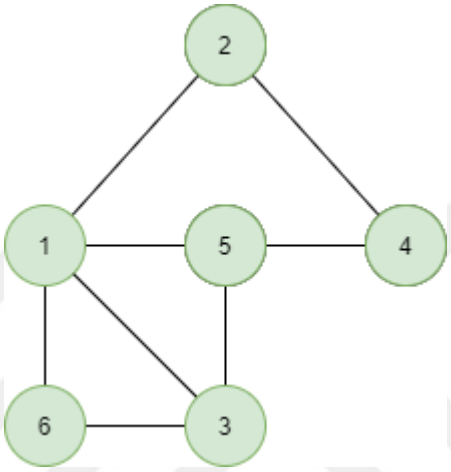


Figure 2.9. The structure of undirected graph

Graphs are matched in many real-life problems. For example, if we consider the objects we want to model as vertex and the relationship between those objects as edges, graphs are very suitable data structures used to meet such needs. The questions within this scope can be grouped under 2 items, (i) "How similar are the graphs given to each other?" and (ii) "How similar are the vertices in a given graph?".

The needs such as finding similarity between graphs, classifying vertices, suggesting new connections have influenced the studies in this field. These studies historically started with graph kernels and evolved towards graph neural networks.

2.6. Graph Kernels

Many different approaches are proposed to find the similarity of the given graphs. Most valid method is to check if the topologies are identical or not, in other words they

are isomorphic. Graph isomorphism problem is in NP, but there is no efficient algorithm for it except heuristic ones. Graph kernels bridge the gap between graph-structured data and a large spectrum of machine learning algorithms called kernel methods. Informally, a kernel is a function of two objects that quantifies their similarity.

Random walk [28] and Weisfeiler-Leman graph kernel [29] can be given as examples of a kernel between graphs.

All the graph kernels suffer computational complexity and cost inefficiency. Another drawback of these approaches is that they cannot generate repetitively usable models and all the computations must be repeated for each graph. To overcome these issues researches working on this topic have begun to evolve into neural networks.

CHAPTER 3
RELATED WORKS

3.1. Graph Neural Networks

Graphs are isomorphic data structures and there is no fixed ordering in them. For example, when the pixel information of a picture is mixed, that picture does not give the same information as before, but graphs contain the same information in every way due to their isomorphic structure. Due to the isomorphic structures of graphs, it is very unlikely that they will be fed directly to neural networks.

The general purpose of graph neural networks is to solve classification and regression problems of a graph that has not been encountered before with a pre-trained model. Previous studies attempt to add learning capability over statistical methods, which assume that the dataset contains patterns and relationships between those patterns.

Some of the significant research studies and variations of GNNs are given in Figure 3.1.

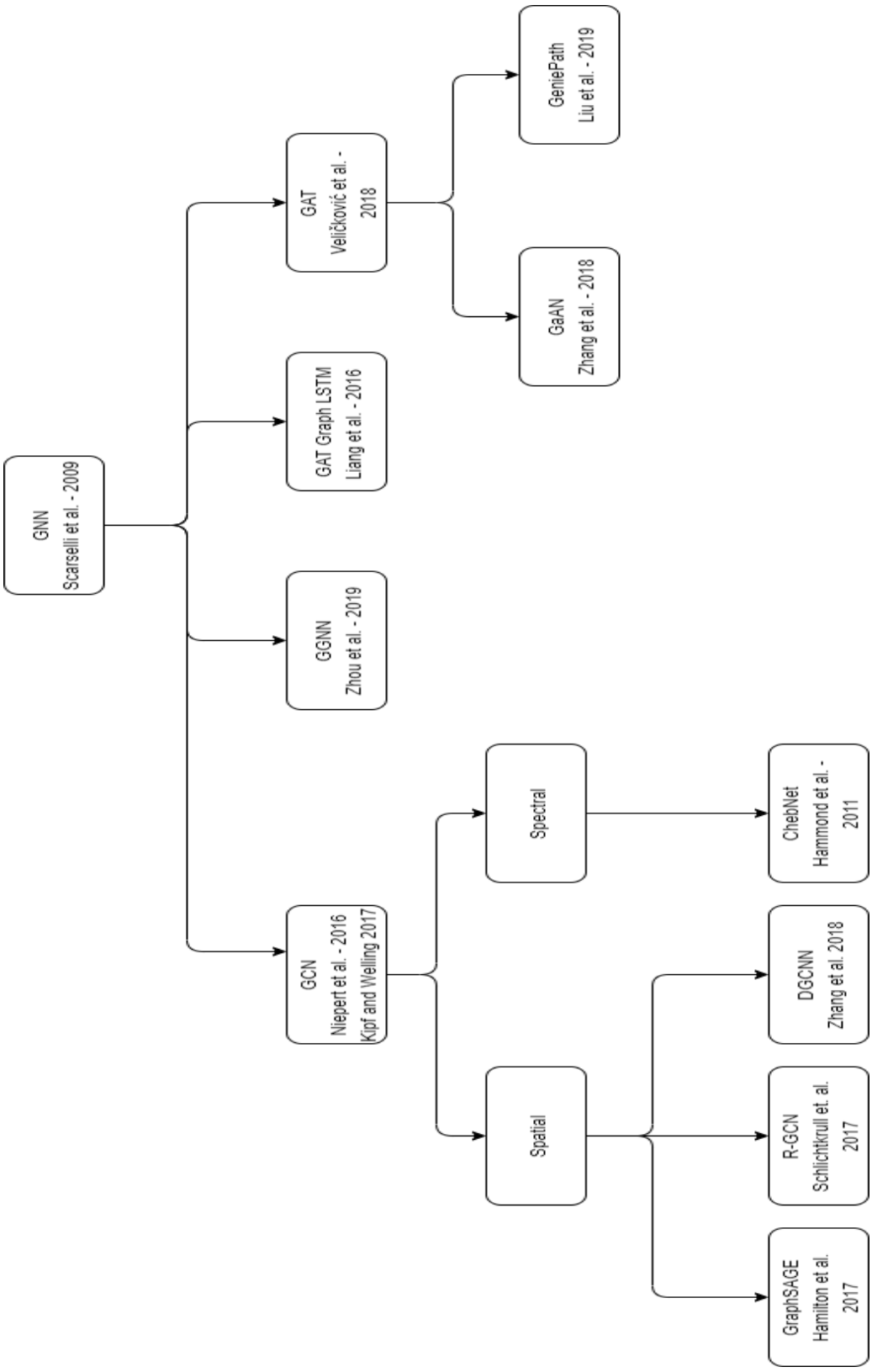


Figure 3.1. Variants Of Graph Neural Networks

Scarselli et al., 2009 introduced Graph Neural Network (GNN) in [11], which applies neural networks to graphs, non-Euclidean data. They extended and applied RNN so it could be applicable to variants of graphs, directed or undirected, cyclic or acyclic. However, the proposed method only works for static graphs and cannot be used for dynamic ones. The proposed method needs to be run separately for all vertices, feeding the information of neighboring vertices to the recurrent neural network consecutively, and to continue this process until the model is stable. The localized functions for GNNs are described in the equation given in Figure 3.2.

$$\mathbf{x}_n = f_{\mathbf{w}}(\mathbf{l}_n, \mathbf{l}_{co[n]}, \mathbf{x}_{ne[n]}, \mathbf{l}_{ne[n]})$$

$$\mathbf{o}_n = g_{\mathbf{w}}(\mathbf{x}_n, \mathbf{l}_n)$$

where $f_{\mathbf{w}} = \sum_{u \in ne[n]} h_{\mathbf{w}}(\mathbf{l}_n, \mathbf{l}_{(n,u)}, \mathbf{x}_u, \mathbf{l}_u)$

x - Set of node states (the state of an arbitrary node, n, is defined as x_n)
l - Set of node features (the features of an arbitrary node, n, is defined as l_n)
$l_{ne[n]}$ - Set of node features for the neighbors of an arbitrary node n
$l_{n1,n2}$ - Set of edge features between two arbitrary nodes n_1 and n_2
f - The local transition function to determine a node's state
g - The local output function to determine a node's output
o_n - The output of an arbitrary node n

Figure 3.2. GNN Localized Functions *[11]*

Another important concept in graph neural networks is message passing. As shown in Figure 3.3., each vertex transmits its own state information to its neighbor vertices. In each iteration, state information from neighbors is given to a function and the hidden state information of the vertex is updated. This function can be a sum, mean etc. The number of steps a vertex collects information from its neighbors is defined as a parameter. This parameter is called a hop.

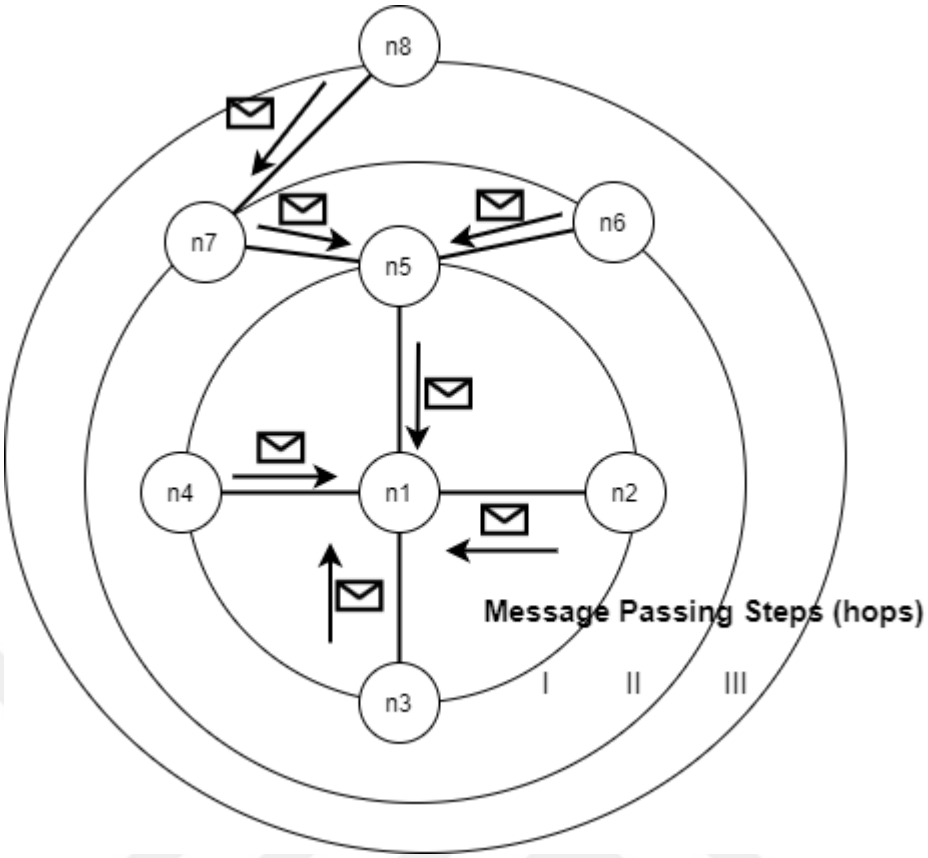


Figure 3.3. Message Passing

The first studies were based on working with a method in which all vertices iteratively publish the information of all their neighbors and this process is repeated until the highest possible stability is reached. However, this process causes very high calculation costs, especially in large graphs. At the same time, it couldn't reach expected accuracy. Although recent studies manage to overcome some of the issues, it is still an open research area.

3.2. Graph Convolutional Networks

Since convolutional neural networks [21] are more effective and highly successful methods compared to other neural network methods, their popularity has increased exponentially in recent years. Convolutional neural networks apply filters to given input data and subsamples them as shown in Figure 3.4. These sampling can be done over a function like average, min, max etc.

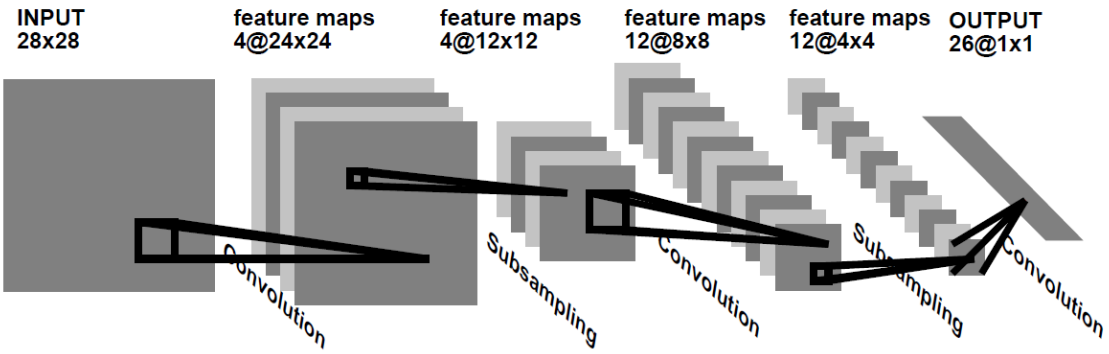


Figure 3.4. Convolutional Neural Network *[21]*

The usage of convolutions on graphs can be compared to the usage on image data. If pixels on the image are considered as vertices, all the vertices are fully connected with the vertices around them. The application of how to apply a filter to such data can be seen in Figure 3.5.

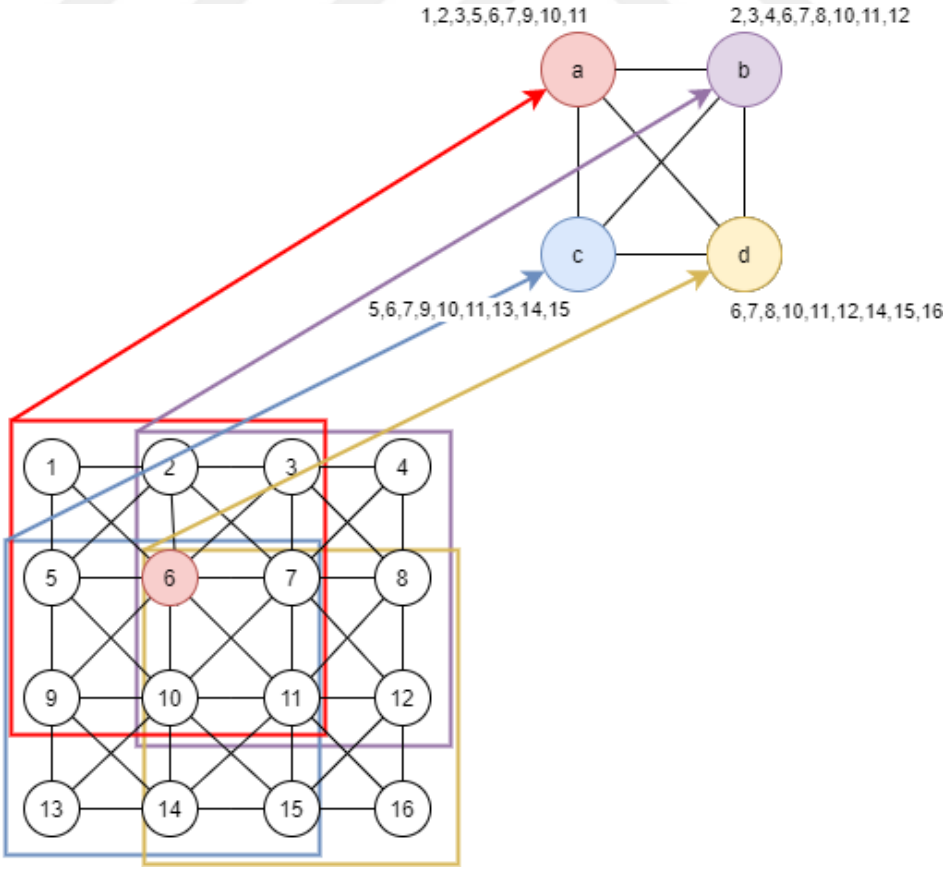


Figure 3.5. 2D-Convolution. Subsampling over 3x3 filter on 4x4 data

Due to its isomorphic structure, the same method cannot be applied on graphs in the same way as it is applied in other data types. However, if the filter to be applied is defined as a hop instead of defining it in (a x a) size and if the neighbor vertices for each vertex are included into a given filter of a hop, then this convolution method is applied to graphs. As shown in Figure 3.6., 1 hop filter applied to vertex no 1, 1 hop neighbors 3, 4, 6, 7 will be included. If a 2-hop filter to vertex no 1 was applied, then 3, 4, 6, 7 vertices from the first-degree neighbor, 2, 8, 12 vertices from the second degree neighbor will be included into the filter. Applying these filters to vertices generates sub-graphs for each vertex.

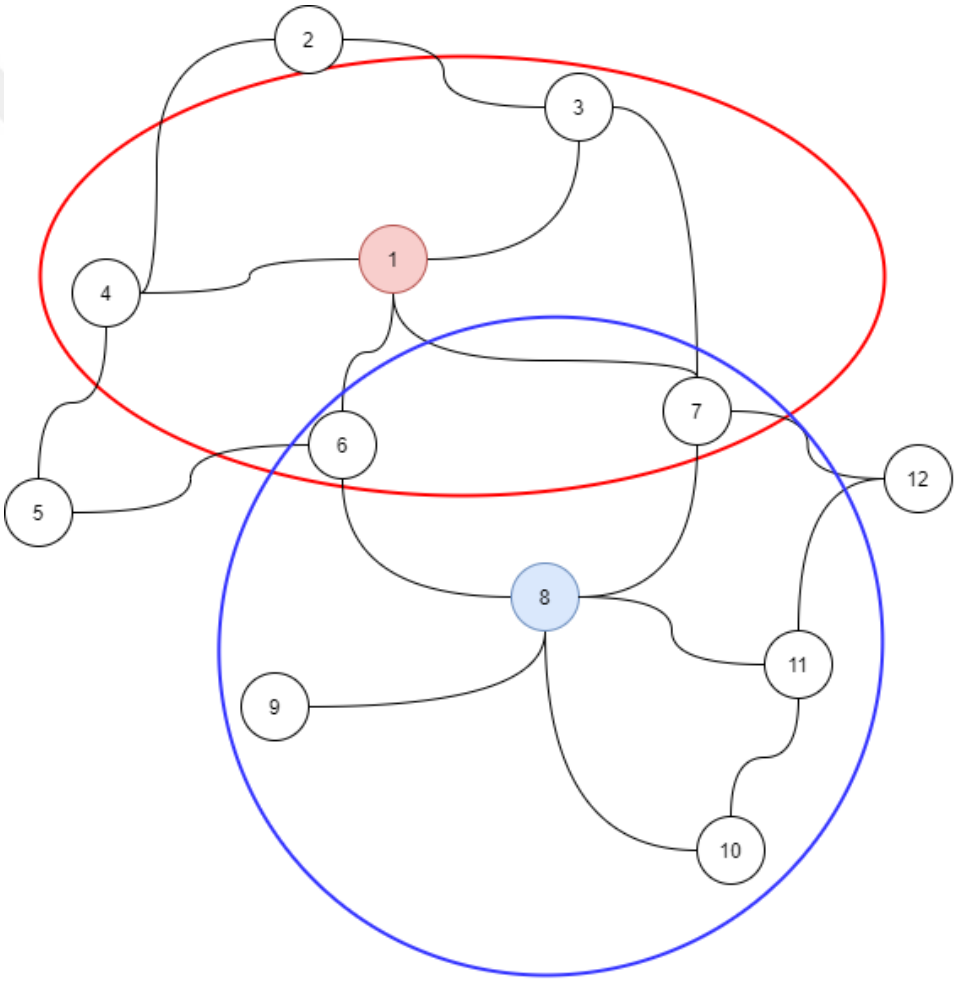


Figure 3.6. Graph Convolution. 1 hop filter applied on vertex 1 and 8.

Convolutional Graph Neural Networks can be divided into two main categories, spectral methods and spatial methods. Spectral models have a theoretical foundation in

graph signal processing and rely on graph Fourier base generalization. Hammond et al. proposed a spectral model called ChebNet [30], they defined a wavelet transform on feature vectors of vertices of a weighted graph. Spatial-based models combine the messaging passing method used by RecGNN [11] with convolution. These models, on the other hand, perform graph convolutions locally on each vertex, where weights can be easily shared across different locations and structures. Spectral models show very low performance on new graphs, which is completely contrary to the philosophy of neural networks. Unlike spectral methods, spatial methods are generalizable, efficient and flexible models.

Zhang et al. proposed Deep Graph Convolutional Neural Network DGCNN [31]. They innovated the Sort Pooling mechanism which makes it possible to use classical CNN on graph structured data. Defferrard et al. combined spectral method with CNN [13]. They used Graph Fourier Transform to find localized convolution filters and cluster similar vertices.

GraphSAGE [32] has made very serious improvements over the original GCN. With these improvements, it has provided solutions to problems such as the high computation cost of applying GCN to large graphs due to its structure, and the necessity of applying it to static graphs. While the original GCN uses the features of all neighbors belonging to a vertex, GraphSAGE has passed the features of the predefined number of neighbors into the aggregator function to identify general representation of a vertex. GraphSAGE has suggested 3 different aggregate functions such as, mean, LSTM and pooling.

FastGCN [33] made improvements on the sampling algorithm. Instead of randomly choosing a fixed number of vertices, it includes the important vertices into the sample set. To do this, instead of sampling directly on the neighbors of the vertex, an importance function for the receptive field in each layer is used.

3.3. Graph Attention Networks

Attention mechanisms are a de facto when we need to deal with sequential data. One of the most important features of the attention mechanism is that it can process variable-size inputs and make decisions using the most relevant pieces of the input. When

attention mechanism is used in conjunction with RNN or CNN, great success has been achieved in learning sentence structures [34] and language translation [35] .

Inspired by the success of the attention mechanism on sequential data Veličković et al. proposed an graph attention network (GAT) [15] for vertex classification tasks. The main idea in their proposed work is that they calculate the information of each vertex using their neighbors and use the principles of self-attention strategy while doing this.

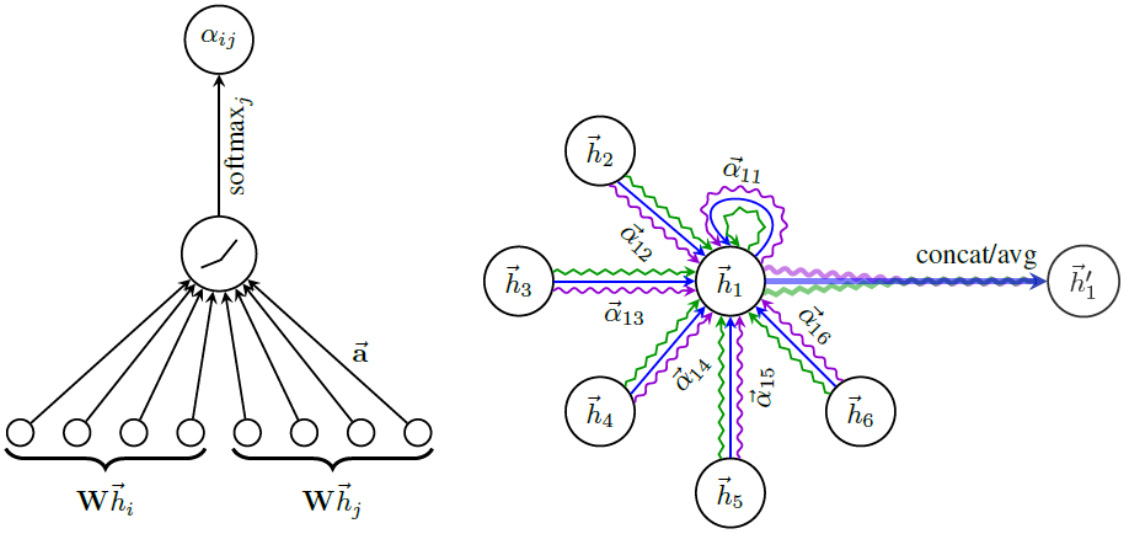


Figure 3.7. Single Attention Layer *[15]*

Left part of Figure 3.7. describes the single-attention layer, used in GAT architecture. Right part of Figure 3.7. shows the multi (3) head attention which increases the expressive capability of the model. Each color represents independent attention. Calculated values from each attention are averaged or concatenated to compute the result. GAT considers that each attention head is of equal weighted. Zhang et al. proposed The Gated Attention Networks (GaAN) [14] model and took this one step further by giving a score to each attention head to improving performance.

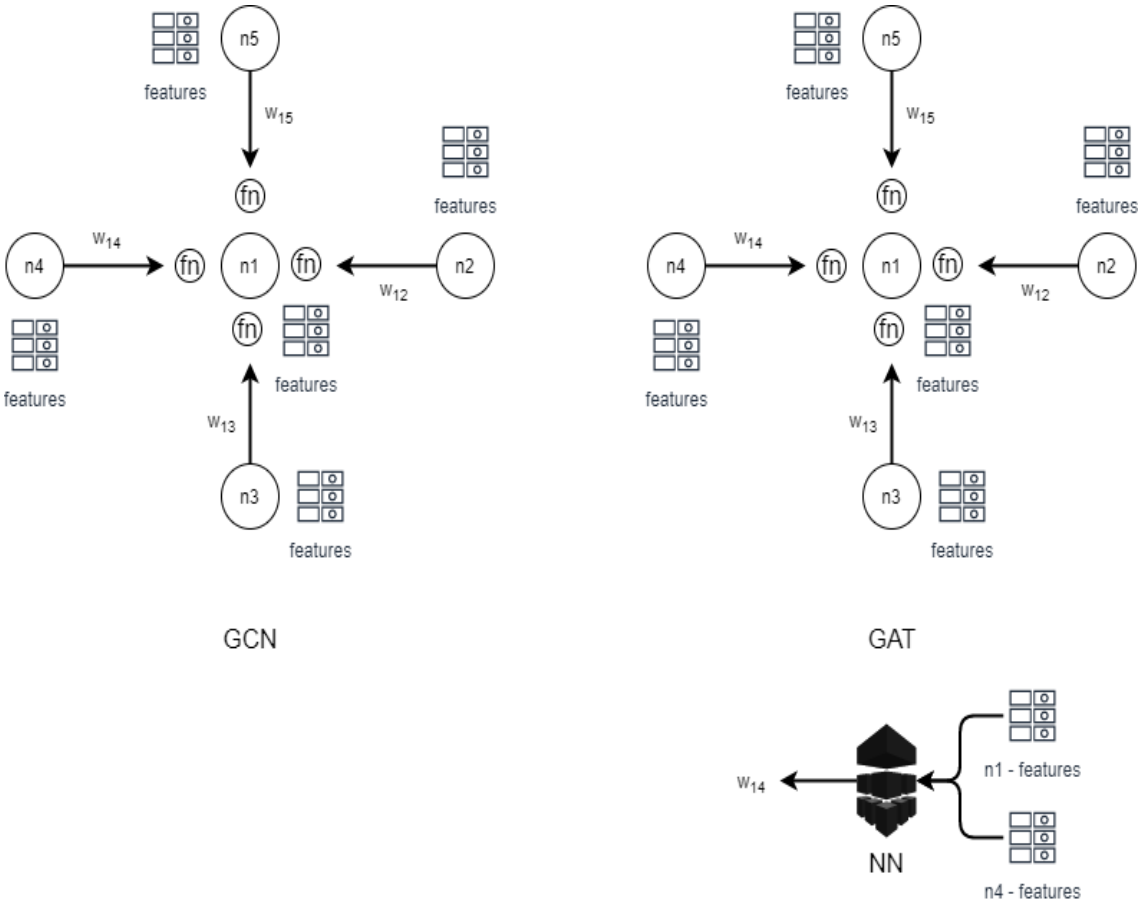


Figure 3.8. Difference of GCN and GAT

Main difference between graph convolutional network and graph attention network is that GCN assigns explicit nonparametric weight to the nearby vertices during the aggregation process, whereas GAT implicitly learns the weights via neural network, in that way more imported vertices have more effects on the learning process.

3.4. Edge and Node Classification on Graphs

Node and Edge classification is a task where the model evaluates the class by the features of the neighbor nodes and/or edges.

The DeepWalk [36] method proposed in 2014 was the first significant deep learning method proposed for node classification. DeepWalk takes samples using random walks (lengths are given as parameters) for each node, to generate embeddings to learn nodes hidden features which represent it. LINE [37] proposed by Tang et al. and node2vec

[38] proposed by Grover and Leskovec extended DeepWalk by changing the embedding generation strategy. DeepWalk [36], GCN [39] , GAT [15] achieved success rates of 67.2%, 81.5%, and 83.1%, respectively, on the CoRA dataset.

In 2017 Muhan Zhang and Yixin Chen proposed Weisfeiler-Lehman Neural Machine (WLNM) [11] for link prediction. WLNM extracts subgraphs from edges' neighbors to train neural networks and uses this model for link prediction.

Kim et al. - 2019 [40] and Gong and Cheng - 2019 [41] proposed multi-dimensional edge feature prediction models. Both proposed works consolidate graph convolutional network and graph attention network models to prediction edge labels and/or features.

3.5. Link Prediction

Link prediction is the prediction of whether there is a link between the nodes defined on the graph [4]. One of the existing approaches for link prediction is heuristic methods. Although this method works for some specific scenarios, it generally performs poorly. For example, whether there is a possible link between two nodes is estimated by looking at the number of shared neighbors. Although this method is successful in social networks, it does not show any success in methods such as predicting intramolecular bonds. [42].

Zhang and Chen proposed SEAL framework [17] for link prediction task which uses sub-graphs, attributes and embedding (uses node2vec for this purpose) features of the graph. SEAL Framework extracts sub-graphs of related nodes and learns the features of these sub-graphs via DGCNN and uses the learned model for link prediction. As seen in figure – 16, this process is achieved in 3 steps.

1. Sub-graph extraction, positive and negative link sampling for training data.
2. Nodes feature vector construction for each sub-graph.
3. DGCNN learning.

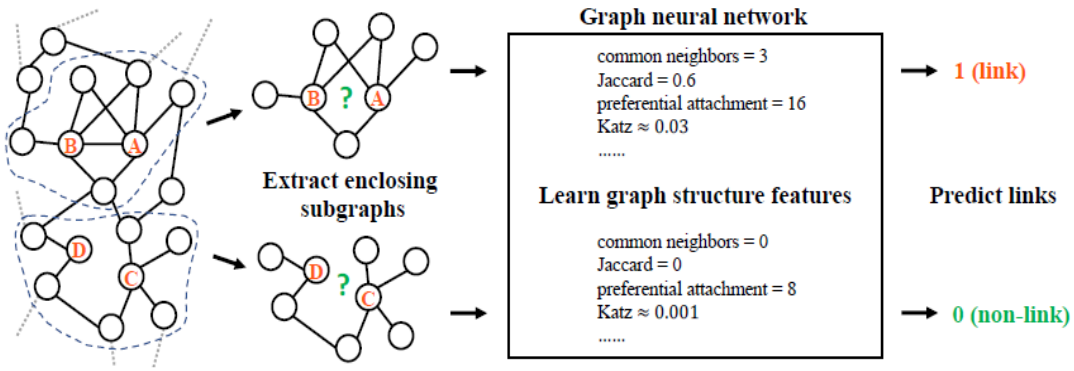


Figure 3.9. Architecture of SEAL Framework *[17]*

Veličković et al. proposed GAT model for graphs and in 2019 Gu et al. specialized this model for link prediction tasks. When the total number of nodes in a graph is defined as N, then node classification has an O(N) complexity, while link prediction has $O(N^2)$ [18]. Original GAT model needs entire graph data at once on node classification task. Those limitations cause memory bottlenecks. It can be thought that this problem can be overcome by using small mini batches, but this strategy reduces link prediction accuracy very much [18]. Gu et al. proposed DeepLinker [18] which uses fixed neighborhoods on mini batch sampling strategy. DeepLinker shares similar architecture with GraphSAGE. Differences between them are in sampling strategy and using GAT instead of GCN. The proposed DeepLinker model is used to create a hidden representation of each node, using the attention mechanism that is shared by the node's neighbors.

Zhan and Chen proposed Weisfeiler-Lehman Neural Machine [16] which combines neural networks with Palette WL which is a variation of Weisfeiler-Lehman algorithm to extract encoded sub-graph patterns. As seen in Figure 3.10., this process is achieved in 3 steps:

1. K-hope neighboring sub-graph extraction.
2. Sub-graph pattern encoding via Palette-WL.
3. Neural network training for classification.

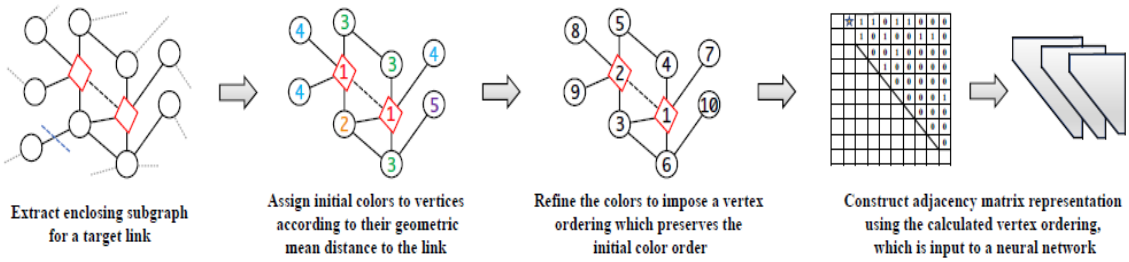


Figure 3.10. Schema showing how the steps of WLNM work *[16]*

CHAPTER 4
PROPOSED METHOD

In software development, the analysis and design phase are very important, since mistakes made in this phase are very costly. One of the most important factors in the analysis and design phase is to understand user needs very well. In this context, the software development team, which may be composed of requirements, software, and quality engineers, defines the usage characteristics and scenarios with different user profile types that will use the software system. Depending on the project size, there may be more than one software development team. This kind of variety should be managed to create complete and consistent specifications. Deficiencies and inconsistencies that may occur in the modeling step can cause serious errors in the entire software system. Many sub-software development processes, from software developments to graphical interface designs, workflows, security scenarios, software testing processes, are affected through the models created at the analysis and design phase.

When developing a software system, the process usually starts with the creation of the model of the system to be developed [43]. In this way, it makes that possible to look at the general view of the designed system and it becomes clear whether the user requirements are met as expected. System modeling requires the ability to distinguish between important and necessary information from others.

There are many different models and tools used in software modeling. Some of these models can be listed as follows: Business Process Modeling and Notation, UML Diagrams, Flowcharts, Event Sequence Graphs. All of these models are graph-based models and there is a well-established theory of graph transformations [1] which has a number of applications to system modelling and software engineering based on concrete specification languages and supporting tools. It is possible to transform software models to their graph representations.

One of the modelling methods used for analysis of graphical user interfaces is Event Sequence Graphs (ESG) proposed by Belli in 2001 [2], which is also used in this thesis. GUIs can be modeled as sequences of events of the objects defined in GUI as given in Figure 4.1. and Figure 4.2.

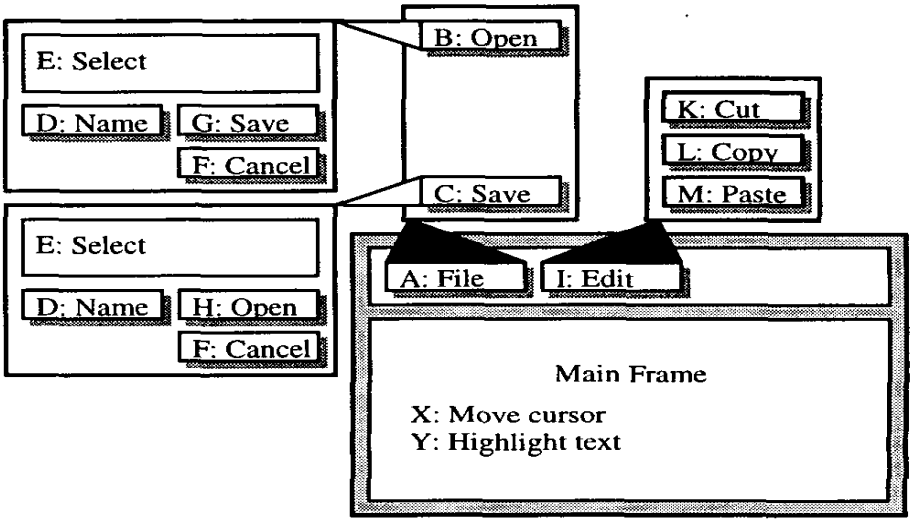


Figure 4.1. A GUI Example [2]

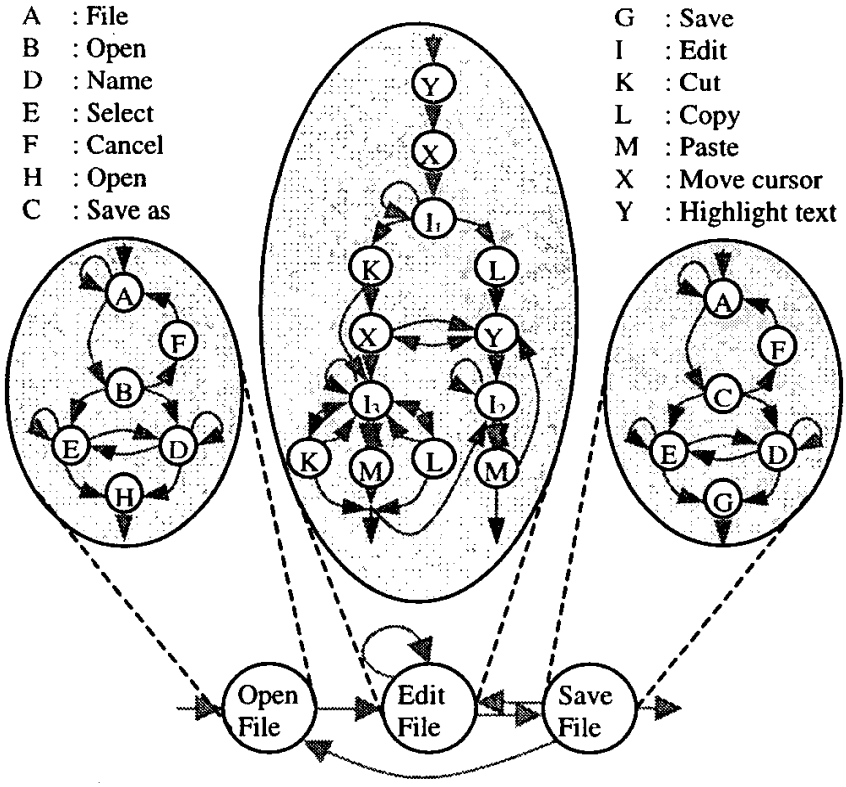


Figure 4.2. ESG model of the GUI [2]

In this thesis, models are considered as the designs of software systems and systems are developed based on these designs. Creating better models will help the software engineers to build better software systems that meet user expectations. The goal of this thesis is to propose a method that finds missing connections between nodes defined in ESG. As in other modeling methods, also in ESG, the missing or forgotten relationships between the components on the model naturally affect all processes to be made through this model.

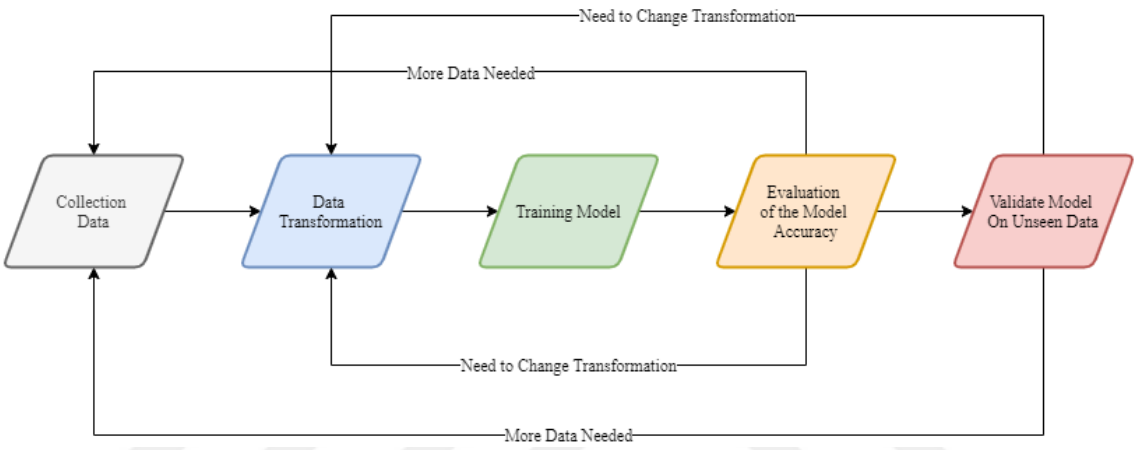


Figure 4.3. Process Used in this Thesis

The process used in this thesis is described in Figure 4.3. On the data collection stage, a bank account [44] (Öztürk - 2020), email [44] (Öztürk - 2020), student attendance [44] (Öztürk - 2020) and reservation system models [45] (Tuglular et al. - 2016) are used. These models are drawn by “Test Suite Designer” (TSD) [45] tool. This tool generates a xml file with mxe extension. The proposed data transformation method reads a mxe file and transforms it to desired graph data that graph neural network models need. At the training stage, GAT and GCN neural network models are used. Three performance metrics, cross entropy loss, area under curve and accuracy are used to measure the performance of trained models.

In the following sections, details of the collected data, how they transformed into required data format and GNN variations and the parameters used are mentioned.

4.1 Data Collection

One of the most challenging processes when working on neural networks is to find or create the data sets. For this purpose, previously prepared data sets using the ESG method were used. The data sets used in thesis are listed as follows.

Bank Account: Transactions such as withdrawal, viewing balance, depositing money, withdrawing money and requesting interest rates are modeled.

Email: Commands such as preparing new messages, viewing the mailbox, answering and forwarding messages, creating an address book, and creating an auto response messages are modeled.

Student Attendance: An attendance/nonattendance tracking application is modeled. In this model, there are two different roles as student and teacher. Students can enter and follow attendance information, and teachers can organize and monitor classes and calendar.

Iselta: It is a model of an application that you can edit and view your profile, list hotels and make reservations.

4.2 Data Transformation

Important part of the data transformation is embeddings [46]. Neural network embeddings are useful because they can reduce the dimensionality of categorical variables and meaningfully represent categories in the transformed space. Embeddings have 3 main purposes; (i) making a recommendation based on categories, finding closest neighbors in embedding vector, (ii) for supervised learning task, converting data to feed to a neural network model, (iii) for the visualization of relations between categories. In this thesis embeddings are used for neural network inputs. As shown in Figure 4.4., each node transformed into its low-dimensional representations.

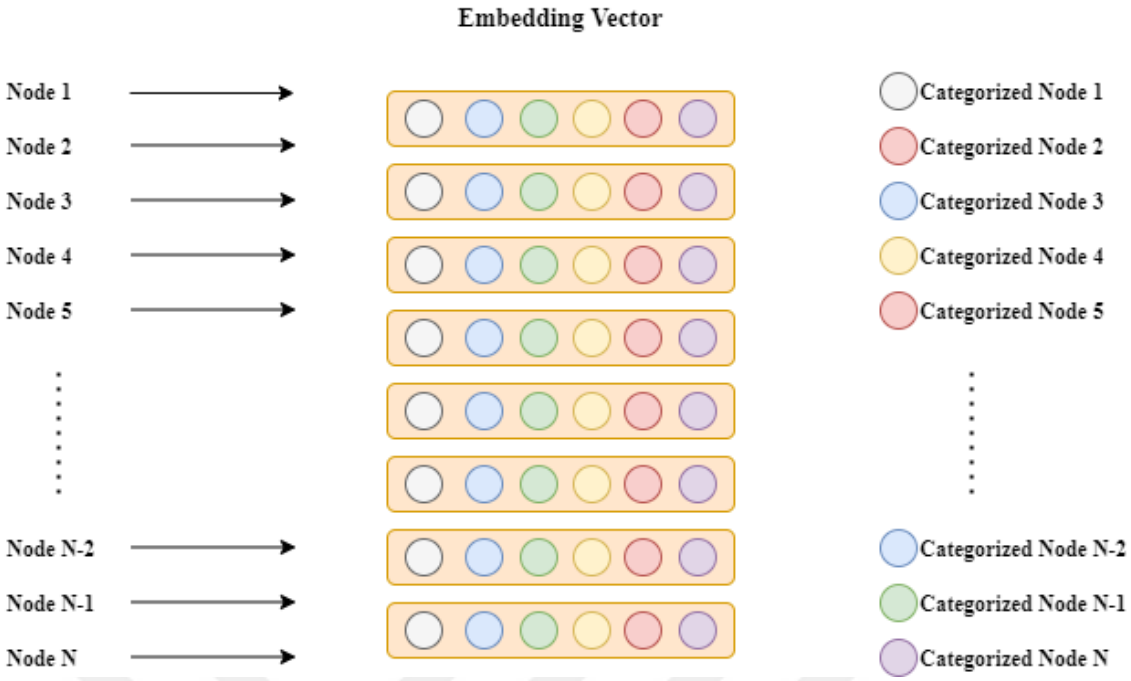


Figure 4.4. Node Embedding

Many software systems that we encounter in real life are complex structures with many details. Regardless of their level of experience, people have an upper limit on their ability to analyze. Since the complexity of software systems makes it impossible to handle all aspects of the system by one person at once, such systems must be designed in parts. Each sub-part to be designed is handled and prepared separately by domain experts and software engineers. The models created as a result of these designs are relatively small. The graphs transformed from these models are naturally small. As the number of embedding vectors and the number of elements within the embedding vectors are increased, it naturally grows the representation space of a node. Therefore, the number of embedding vectors should be selected carefully for the proper representation of small graphs.

According to the number of embedding vectors used and the number of elements defined in each embedding vector, the number of elements in the representation space that will be represent a node can be calculated with the following formula.

$$E = \prod_{v=1}^{n} \sum_{i=0}^{k} 1$$

Equation 4.1.

Where k is the number of elements in embedding vector v, n is the number of embedding vectors defined.

As the number of nodes in graphs decreases, it is necessary to keep the representation space smaller in order to infer the patterns of connections between nodes. In this context, embedding vectors and their number of elements should be selected carefully. If the representation space to represent nodes on small graphs becomes larger, the representation will not be able to switch from high dimensionality to low dimensionality, although embedding is applied. One of the main purposes of the embeddings is the transition from high dimensionality representation to low dimensionality one. The size of this representation space (E) must be much less than the number of nodes (G (n)) in the graph.

$$E << G(n)$$

As the number of nodes in graphs decreases, it is necessary to keep the representation space smaller in order to infer the patterns of connections between nodes. In this context, embedding vectors and their number of elements should be selected carefully. If the representation space to represent nodes on small graphs becomes larger, the representation will not be able to switch from high dimensionality to low dimensionality, although embedding is applied. One of the main purposes of the embeddings is the transition from high dimensionality representation to low dimensionality one. The size of this representation space (E) must be much less than the number of nodes (G (n)) in the graph.

Node Type		Entry-Exit		Value Type	
1	Error	1	None	1	None
2	Info	2	Entry	2	Numeric
3	Action	3	Exit	3	Bool
4	Input			4	Text
5	Success			5	Date/Time
				6	Enum
				7	File
Event Type		**Is A Form Element**		**Annotation**	
1	Data Input	1	Yes	1	{}
2	Help/Info/Message	2	No	2	a
3	Save			3	b
4	Cancel			4	a,b
5	Process			5	c
6	Calculate			6	a,c
7	Validate			7	b,c
8	Navigate			8	a,b,c
9	Delete			9	d
10	Get			10	a,d
11	Load			11	b,d
12	Select			12	a,b,d
				13	c,d
				14	a,c,d
				15	b,c,d
				16	a,b,c,d

Figure 4.5. Sample Node Embedding Vectors.

Annotations a: Is Required, b: Has Min-Max Value c: Has Min-Max Length

d: Has Condition or Regex

ESG - Node	Node Type	Entry-Exit	Value Type	Event Type	Is A Form Elem.	Annotation
Save Data	3	1	1	3	1	1
Name Input	4	1	4	1	1	6
Age Input	4	1	2	1	1	3

Figure 4:6. Sample Application of Node Embedding Vectors for nodes of an ESG

Embedding vectors that can be used in an ESG model are defined in Figure 4.5. In Figure 4.6, these embedding vectors are applied on sample nodes. When all the embedding vectors defined in the Figure 4.5. are used for an ESG model, the size of the representation space can be calculated with Equation 4.1. The number of embedding vectors and the number of elements in each embedding vector is given in the Figure 4.5. are as follows: Node Type: 5, Entry-Exit: 2, Value Type: 7, Action Type: 12, Is A Form Element: 2, Annotation: 16. In this case, the representation space size is 5x2x7x12x2x16=26.880. There will be a relatively large graph needed to learn the edge patterns of graph nodes represented by elements in a representation space of this size. However, as mentioned earlier, models in a software system are designed smaller in nature. To prevent this undesirable situation, it would be more appropriate to represent the nodes belonging to Event Sequence Graphs with a single embedding vector. In this context, the "Event Type" embedding vector is chosen as the most suitable embedding vector for learning since it expresses the patterns between nodes best by the neural network. Moreover, transformation from node names defined in an ESG to "Event Types" embedding elements can be performed with a simple mapping operation. For these reasons "Event Type" embedding is used in this thesis. Of course, embeddings can be learned and reused in different models. But in this thesis, embedding vectors generated manually.

Files generated by TSD have mxe extension and they are formed in xml notation. Key elements used in this xml are <mxGraphModel> which represents a graph, and <mxCell>, which represents a vertex if <vertex=1> attribute is defined or an edge if <edge=1> attribute is defined. TSD allows to define sub-graph via <mxCell> elements, which contains child <mxGraphModel>. <mxCell> elements have an <id> attribute, which is unique in container <mxGraphModel>, but it is not globally unique. When a child graph is defined in a graph, <mxCell> defined in that graph have their own id sequences. Sample snapshot of a mxe file shown in Figure 4.7.

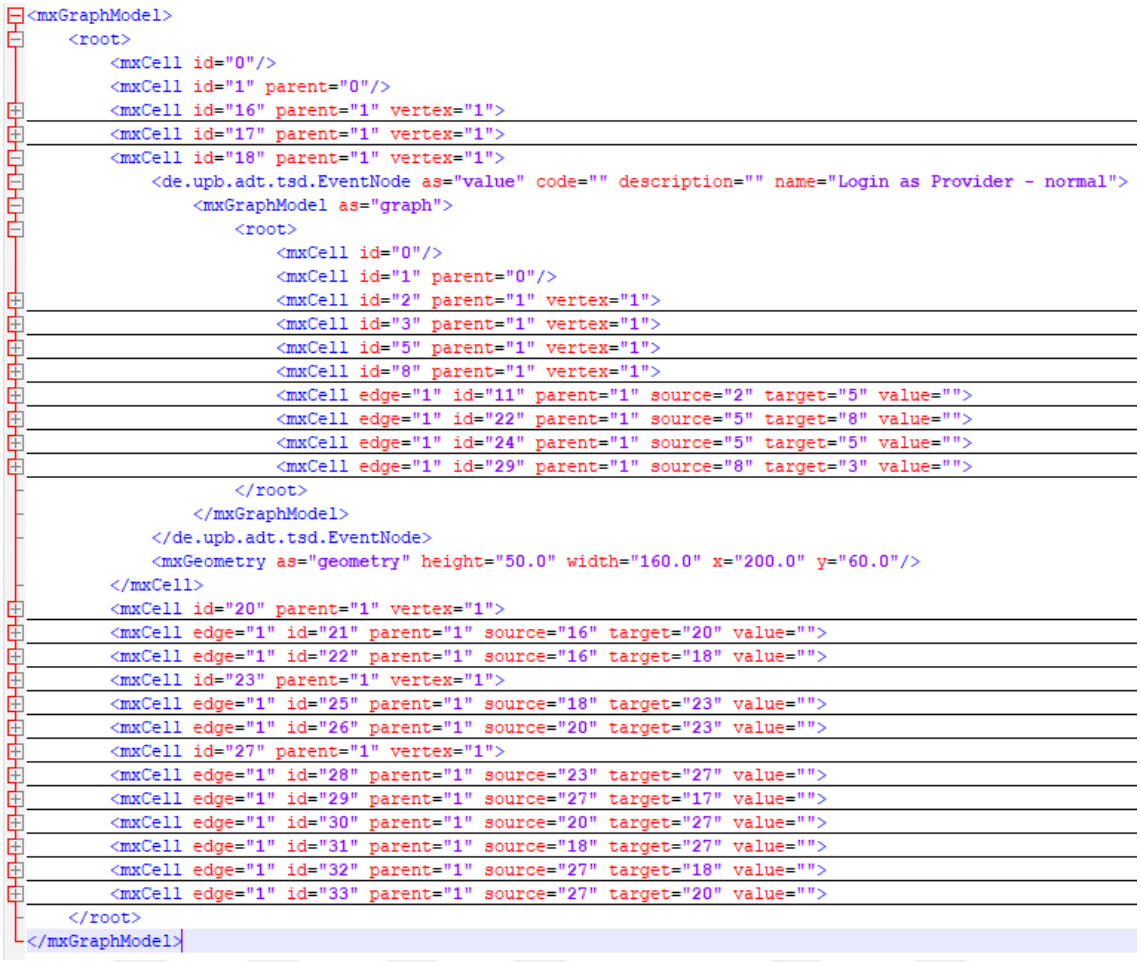


Figure 4.7. Content of a mxe file

Child graphs in a mxe file represented as grouping vertices in their parents. ESGs should be flattened to be analyzed properly. To accomplish this task, a mxe model parser tool is implemented. This tool has two main part; one parses mxe and extracts graphs' edges and vertices and second one flattens the cascade graphs. The algorithm for the tool is shown in Figure 4.8. and Figure 4.9.

```
def parse(xmlroot,parentMxCellId):
        nodes = []
        edges = []
        #node identifiers of the sub graph grouping nodes
        subGraphGroupingNodes = []
        entryMxCell = null
        exitMxCell = null
        #extract and parse nodes only to find all nodes and generate node identifiers
        #sometimes edge elements defined before the node definition
        for mxCell in xmlroot:
                if mxCell is vertex:
                        nodes.append(mxCell)
                        if mxCell is entryCell:
                                entryMxCell = mxCell
                        elseif mxCell is exitCell:
                                exitMxCell = mxCell
                        elseif mxCell has subGraph:
                                subNodes, subEdges, childSubGraphGroupingNodes,childEntryMxCell,childExitMxCell = parse(mxCell.subGraph,mxCell.Id)
                                nodes += subNodes
                                edges += subEdges
                                subGraphGroupingNodes += childSubGraphGroupingNodes
                                subGraphGroupingNodes.append([mxCell,entryMxCell,exitMxCell])
        for mxCell in xmlroot:
                if mxCell is edge:
                        sourceNode = findNode(nodes, mxCell.source)
                        targetNode = findNode(nodes, mxCell.target)
                        edges.append([sourceNode,targetNode])
        return nodes,edges,subGraphGroupingNodes,entryMxCell,exitMxCell
```

Figure 4.8. Mxe Parser

```
 def parseAndFlattenGraph(xmlroot):
         nodes = []
         edges = []
         nodes, edges,subGraphGroupingNodes = parse(mxroot,"")
         #remove links to sub-graph grouping nodes
         flattenEdges = removeGroupingNodeEdges(edges,subGraphGroupingNodes)
         #add links from parent graph to sub-graph
         for groupingNode in subGraphGroupingNodes:
                 #find the edges which target is given grouping node,
                 edgesByTargetCell = findEdgesByTargetCell(edges,groupingNode.MxCell)
                 #find the edges which source is given entry node
                 edgesBySourceCell = findEdgesBySourceCell(edges,groupingNode.EntryCell)
                 for edgeByTargetCell in edgesByTargetCell:
                         for edgeBySourceCell in edgesBySourceCell:
                                 flattenEdges.append([edgeByTargetCell.source,edgeBySourceCell.target])
                 #find the edges which target is given exit node,
                 edgesByTargetCell = findEdgesByTargetCell(edges,groupingNode.ExitMxCell)
                 #find the edges which source is given grouping node
                 edgesBySourceCell = findEdgesBySourceCell(edges,groupingNode.MxCell)
                 for edgeByTargetCell in edgesByTargetCell:
                         for edgeBySourceCell in edgesBySourceCell:
                                 flattenEdges.append([edgeByTargetCell.source,edgeBySourceCell.target])
                 if groupingNode has selfLoopEdge:
                         #find the edges which target is given exit node,
                         edgesByTargetCell = findEdgesByTargetCell(edges,groupingNode.ExitMxCell)
                         #find the edges which source is given grouping node
                         edgesBySourceCell = findEdgesBySourceCell(edges,groupingNode.EntryCell)
                         for edgeByTargetCell in edgesByTargetCell:
                                 for edgeBySourceCell in edgesBySourceCell:
                                         flattenEdges.append([edgeByTargetCell.source,edgeBySourceCell.target])
         #remove links to sub-graph grouping nodes
         flattenNodes = removeGroupingNodes(nodes,subGraphGroupingNodes)
         return flattenNodes, flattenEdges
```

Figure 4.9. Flatten Graph Generator

4.3 Training Model

4.3.1 SEAL Framework

SEAL [17] is a specialized framework for link prediction. With an innovative approach, the authors transform the link prediction problem into a sub-graph classification problem. For each edge, a surrounding sub-graph extracted at n-hop distance. In addition, negative samples containing wrong connections were created. These generated sub-graphs and node feature matrix (which contains k features for each node) feed to a GNN for classification. In this way, both node features and graph structure were used during the learning process.

SEAL implemented by Zhang [47]. This implementation consists of the following steps; read graph data and node attributes (if use_attribute argument has been given) from file, load them into compressed sparse column matrix, sampling both positive and negative train/test links from loaded matrix, if embedding learning enabled, node2vec [38] is used to create node information. If the library runs on training mode (is default behavior) then for each target link, SEAL extracts its n-hop (via hop argument) enclosing subgraph and creates its node information matrix. SEAL uses deep graph convolutional neural network (DGCNN) [31] model for classification. SEAL transforms the link prediction problem into graph a classification problem and each subgraph (positive and negative samples) generated by SEAL passes through DGCNN for classification task. The architecture of DGCNN is given in Figure 4.10.

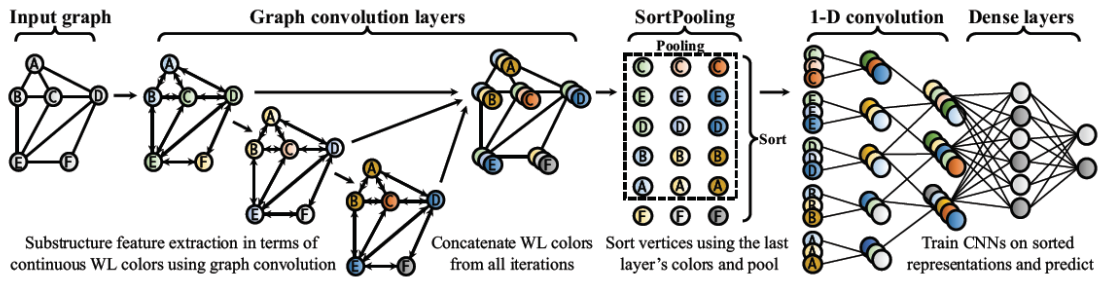


Figure 4.10. Architecture of DGCNN *[31]*

In DGCNN [31] architecture, Sort Pooling layer is the key innovation, which differentiates it from other GCNs. On traditional GCN, feature values of neighboring

nodes are summed up before passing them to CNN, but in DGCNN, Sort Pooling layer organizes node features in a solid order. In this way, it makes it possible to keep more information about different node features. Input of this layer is node features and feature channels and the output are sorted node features and output channels of each feature. Details of the CNN used in DGCNN is shown in Figure 4.11.

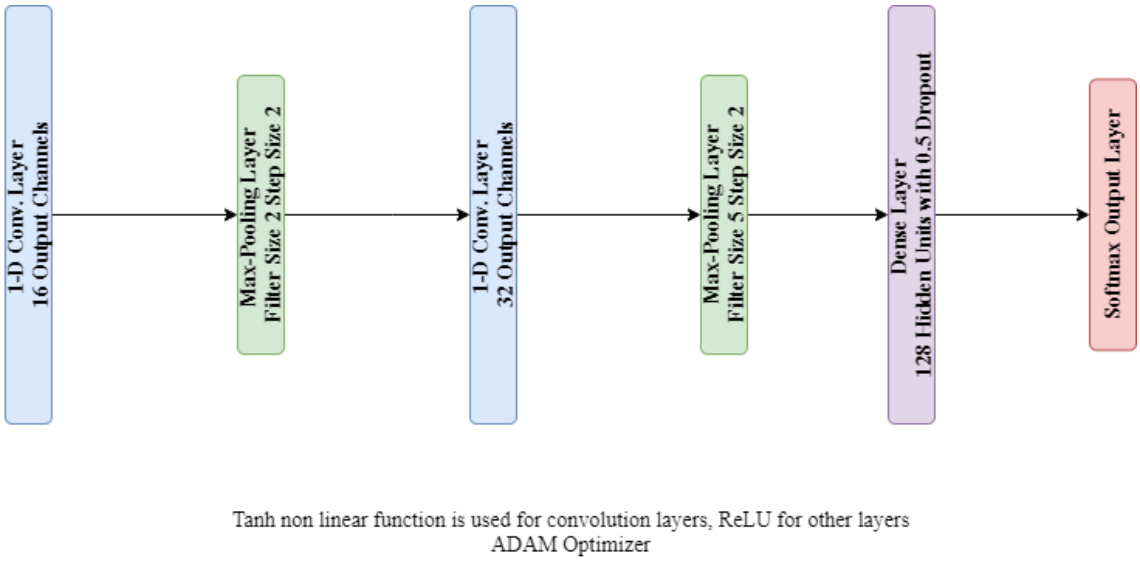


Figure 4.11. Details of the CNN configuration used in DGCNN

Original SEAL implementation is extended in some aspects as follows. Parameters of DGCNN model are hidden and couldn't be tuned externally. The ability to tune hyperparameters of neural network is crucial. Some minor bugs are fixed which prevents the application to run on training data format except mat file format. Training, validation and test results were printed on the screen by the application. Working in this way was challenging to evaluate results between iterations. For this reason, all the results are written in a csv formatted file at the end of the each iteration. Extended version of the SEAL is published to github as SEAL-ESG and can be accessible publicly from https://github.com/onurleblebici/SEAL-ESG. Available parameters of the SEAL-ESG implementation and their explanations are given in Table 4.1. If embeddings are enabled, then node2vec software is needed to run the application.

Table 4.1. Arguments of the SEAL-ESG Framework

Argument Name	**Explanation**	**Default Value**
data-name	Data filename which has mat extension	
train-name	Training data file name which is formatted as plain text	
test-name	Test data file name which is formatted as plain text. This is an optional parameter, it is also possible to use some part of training data as test data.	
only-predict	If True, will load the saved model and output predictions for links in test-name; you still need to specify train-name in order to build the observed network and extract subgraphs	
batch-size	Number of data feed to model on each iteration	50
max-train-num	Set maximum number of train links (to fit into memory)	100000
no-cuda	Disables CUDA training	False
seed	Random seed to initialize the pseudo-random number generator.	1
test-ratio	Ratio of test data to be used in training data	0.2
no-parallel	If True, use single thread for subgraph extraction, by default use all CPU cores to extract subgraphs in parallel	False
all-unknown-as-negative	If True, regard all unknown links as negative test data; sample a portion from them as negative training data. Otherwise train negative and test negative data are both sampled from unknown links without overlap	False
hop	Enclosing subgraph hop number	1
max-nodes-per-hop	If > 0, upper bound the number of nodes per hop by subsampling	None
use-embedding	Whether to use node2vec node embeddings	False
use-attribute	Whether to use node attributes	False
save-model	Save the final model	
sortpooling_k	Specifies how many percent of the output of sort pooling layer will be fed to CNN, number of nodes kept after SortPooling	0.6
latent_dim	Dimensions of latent layer. Linear transformation for SortPooling layer output	[32, 32, 32, 1]
hidden	Number of hidden units in dense layer	128
out_dim	Auto calculate input size for dense layer, graph embedding output size	0

(cont. on next page)

Table 4.1. (cont.)		
dropout	Dropout enabled for dense layer	True
num_class	Binary classification- link exists or not	2
num_epochs	Number of times training data will feed to model	50
learning_rate	Update weight coefficient	1e-4
validation-size	Validation dataset size (percentage of training dataset)	0.1

4.3.2. DeepLinker

DeepLinker [18] is an extension of GAT [15], which is specialized for link prediction. The input of a GAT is features of each node and the output is learned features of each node produced by GAT. A shared linear transformation with a weight matrix which is applied to each node is required to transform the input node features into a learned output feature. A single layer feed forward neural network (FFNN) with weight vector called attention mechanism is used to find out which neighbors of a node are more important (softmax function is used for ranking). Importance factor calculation between node *i* and node *j* is shown in Equation 4.2. where *W* is weight matrix, *h* is set of node features, *T* is transposition and || is the concatenation operation. Schema of the attention mechanism, which is formulated in Equation 4.2. is shown in Figure 4.12.

$$e_{ij} = a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j)$$

$$\alpha_{ij} = \mathrm{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k\in\mathcal{N}_i}\exp(e_{ik})}.$$

$$\alpha_{ij} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\mathrm{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)}$$

Equation 4.2. Attention mechanism *[15]*

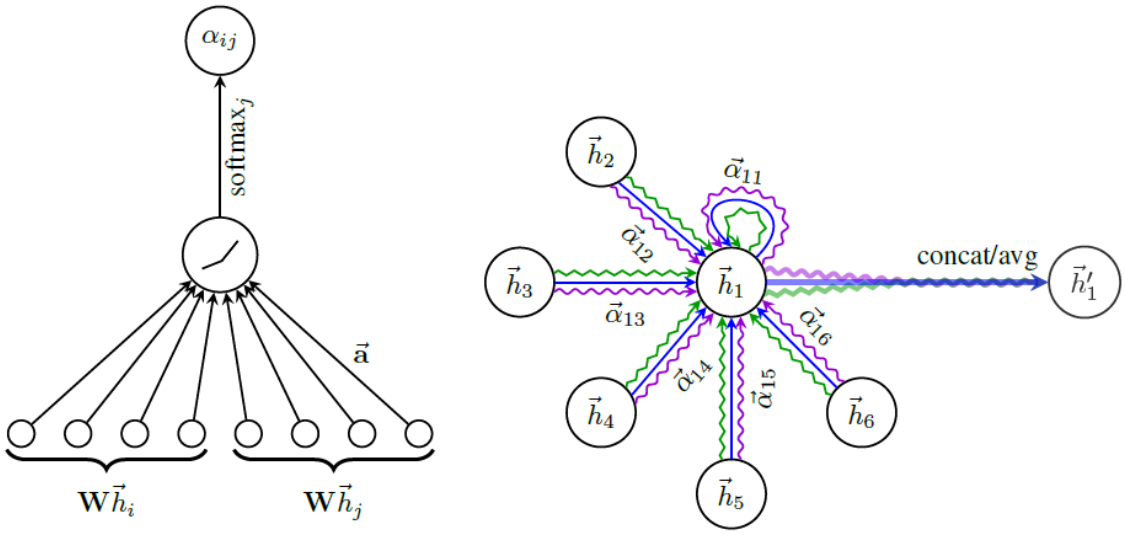


Figure 4.12. Single Attention Mechanism between two nodes is shown in left, and the Multiple Head Attention Mechanism between a node an its neighbors is shown in right. *[15]*

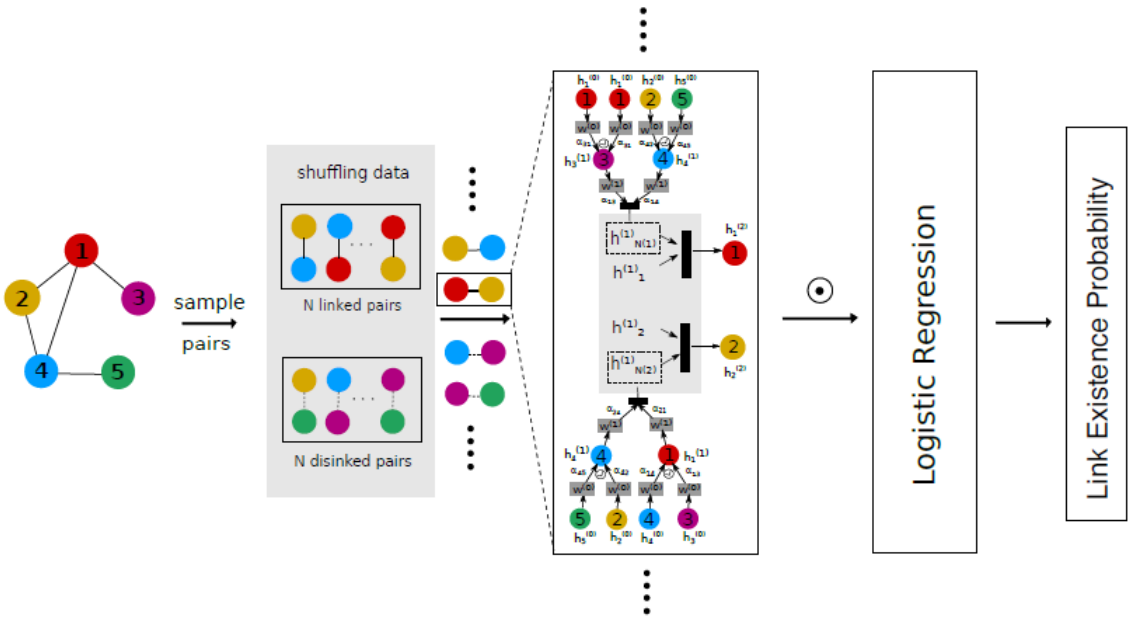


Figure 4.13. Architecture of DeepLinker *[18]*

The architecture of DeepLinker is shown in Figure 4.13. DeepLinker creates a data set for a given graph by creating positive (nodes those have connections) and negative (nodes those have no connection) edge samples. The following operations are

performed for each of the node pairs in the data set; for current node pair (for example 1 and 2) find the first (3, 4) and second level (1, 2, 5) neighbors of each node. DeepLinker uses fixed sized neighborhood sampling for optimum memory usage, and then calculates the edge vector representation of the node pair over their and their neighbor's initial features using GAT. Following that, DeepLinker calculates the Hadamard distance of the output of the GAT, which is an edge vector representation of the node pair and makes link predictions via training a logistic regression function.

Original DeepLinker implementation is extended in some aspects as follows. There was and no parametric data input support to work with other training data. Along with this, a feature that can load the outputs of the mxe_parser application has been added. Only gpu support was available, cpu support is added. Test evaluation metrics are calculated at the end of the each epoch. Training, validation and test results were printed on the screen by the application, all the results are written in a csv formatted file at the end of each iteration. Extended version of the DeepLinker is published to github as DeepLinker-ESG and can be accessible publicly from https://github.com/onurleblebici/DeepLinker-ESG. Available parameters of the DeepLinker-ESG (also used for GAT) implementation are given in Table 4.3.

Table 4.2. Parameters of DeepLinker

Parameter	**Explanation**	**Value**
epochs	Number of epochs to train.	100
lr	Initial learning rate-Adam Optimizer	5e-4
weight_decay	Weight decay, L2 loss on parameters-Adam Optimizer	5e-4
hidden	Number of hidden units	32
K	Number of attention-Multiheaded Attention	8
dropout	Dropout rate	0.5
batchSize	32	
trainAttention	Train attention weight or not	True
dataset-name	Name of the dataset	
validation-size	Validation dataset size (percentage of training dataset)	0.1

CHAPTER 5
Evaluation

SEAL-ESG and DeepLinker-ESG link prediction approaches are performed on Software models, Bank Account, Student Attendance, Email and ISELTA drawn by ESG Tool. The studies performed to predict possible missing links on a given ESG. In addition, results and discussion, threats to validity is explained in this section.

5.1. Experiments

The experiment steps can be listed as follows: preparing the environment, determining node features and creating an embedding file to find node feature, parsing the files with mxe extension, transforming them into files containing the edge and node information of the graph, and training the model using these output files.

First step is to prepare the environment. The hardware configuration of the computer used in experiments is presented in Table 5.1. Required installations is listed in Table 5.2. Installed Python libraries are given in Table 5.3. Git repositories to be cloned are listed in Table 5.4.

Table 5.1. Computer Hardware Specifications

CPU	Intel(R) Core (TM) i7-9750H CPU @ 2.60GHz
RAM	16 GB, 2667 Mhz
Disk	PM981 NVMe Samsung 512 GB
GPU	NVIDIA GeForce GTX 1650, Dedicated GPU memory 4.0 GB

Table 5.2. Required Installations

Ubuntu 20	Operating System
Python 3.8/2.7	Python is an interpreted, high level and general purpose programming language.
CUDA Toolkit 11	CUDA is a parallel computing platform and programming model that makes using a GPU for general purpose computing
Conda 4.8.4	Package management and environment management system

Table 5.3. Required Python Libraries

Library Name	SEAL-ESG Version	DeepLinker-ESG Version
python	3.8.x	2.7.x
pytorch	1.6	-
torch	-	1.4
numpy	1.18.1	1.16.6
scipy	1.4.1	1.2.3
networkx	2.4	2.2
tqdm	4.42.1	-
sklearn	0.0	0.0
gensim	3.8.3	-
tensorboardX	2.1	2.1
future	-	0.18.2

Table 5.4. Required Git Repositories

Repository Name	Repository URL
SEAL-ESG	https://github.com/onurleblebici/SEAL-ESG
DeepLinker-ESG	https://github.com/onurleblebici/DeepLinker-ESG
mxeParser-ESG	https://github.com/onurleblebici/mxeParser-ESG

Second step is determining the node features, which defines the node best. This is a manual process depending on ESGs under consideration because different domain may require different kinds of node features. The chosen features for the evaluation are listed in Table 5.5. and they are determined as generic as possible for a regular application. Mapping table used for the conversion of node names to node features is listed in Table 5.6. This is a manual process. Table 5.6 should be stored row by row in a file with txt extension, separated by commas. The first column specifies the numeric identifier of the feature, the second column contains the friendly name of the feature, and the other columns hold regular expressions to be used for node name matching. A sample line belonging to an embeddings.txt file is defined as: "2, Info, confirm*, prompt*, receive *". Node feature distribution of all dataset is given in Table 5.7.

Table 5.5. Node Features

Numeric Identifier of The Node Feature	Friendly Name of the Node Feature
0	[,]
1	Error
2	Info
3	Input
4	Help
5	Save
6	Edit
7	Add
8	Ok
9	Cancel
10	Process
11	Calculate
12	Validate
13	Navigate
14	Delete
15	Get
16	Load
17	Select
18	Print
19	Access
20	View

Table 5.6. Node Name to Node Feature Mappings

Id	Name	Regular Expressions (Matching Node Names)
0	**[,]**	[,]
1	**Error**	error
2	**Info**	info, confirm, prompt, receive
3	**Input**	input, data, characteristics, contingents, prices, special, change, enter, pay*, ~free, read
4	**Help**	help
5	**Save**	save, send, put, take, submit
6	**Edit**	edit, update
7	**Add**	add, new, compose, create

(cont. on next page)

Table 5.6. (cont.)		
8	**Ok**	ok
9	**Cancel**	cancel
10	**Process**	process, encrypt, sign, server*, returnMoney
11	**Calculate**	calculate
12	**Validate**	validation, verify
13	**Navigate**	navigate, link, overview, continue, pause, finish, release
14	**Delete**	delete
15	**Get**	get, open, request
16	**Load**	load
17	**Select**	select
18	**Print**	print
19	**Access**	log*in, log, sing*in, access
20	**View**	view, trace, monitor

Table 5.7. Node Feature Distribution of Dataset

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
	[OR]	Error	Info	Input	Help	Save	Edit	Add	Ok	Cancel	Process	Calculate	Validate	Navigate	Delete	Get	Load	Select	Print	Login	View	Total Nodes
ISELTA	2	5	0	19	0	5	2	1	3	6	0	0	3	20	0	0	0	0	0	2	0	68
Student Attendance System	26	0	47	48	0	43	81	17	0	0	5	0	0	0	8	43	0	51	0	30	48	447
Bank Account	16	8	18	28	0	16	0	0	0	10	0	0	0	0	0	16	0	16	0	0	0	128
Email	26	0	54	40	0	13	0	13	0	0	2	0	0	0	0	13	0	13	0	0	0	174
Total Labels	70	13	119	135	0	77	83	31	3	16	7	0	3	20	8	72	0	80	0	32	48	

Third step is generation of dataset. mxe_parser application and mxe files can be found in mxeParser-ESG git repository, given in Table 5.4. Available arguments of the mxe_parser application is listed in Table 5.8. Details of graphical models used in this thesis are given in Table 5.9.

Table 5.8. Arguments of mxe parser

Argument	**Explanation**
--input	Path of the mxe file
--output	Prefix name of the output files.

(cont. on next page)

Table 5.8. (cont.)	
--number-of-node-features	Number of node feature will be added to output
--number-of-edge-features	Number of edge feature will be added to output
--as-undirected	Converts directed graph to undirected graph
--generate-edge-symmetry	Generates edge symmetry on adjacency matrix
--embeddings	Path of the embeddings file
--tab-to-eol	Adds extra tab to end of each line
--add-info-firstline	Adds number of edges, nodes and features of them as first line into relevant output files.
--add-node-labels	Adds the node labels to the last column of the nodes output file. (This is required for DeepLinker-ESG)
--duplicate-node-features	If arg > 1 then clones same feature given n times. (This is required for DeepLinker-ESG to use in attentions)

Table 5.9. Graph Data Details of Dataset Models

Dataset	**Number of Nodes**	**Number of Edges**
ISELTA	68	249
Student Attendance System	50	95
Bank Account	21	38
Email	19	35

Output graphs generated by mxe_parser is created as directed (via --as-undirected False argument). For each mxe input file, application generates three output files, which are nodes, edges and mappings. Node output file is a tab separated file and each line represents a node and features of the node except the first line, first line represents the number of nodes and number of node features that this graph contains. Edge output file is also a tab separated file, first line represents the number of edges and number of edge features defined for this graph. Other lines are structured like, first column is the source node identifier and the second column are the target node identifier for the edge, other columns represent the features of the edge if exists. Node mapping output file shows the mappings of nodes defined in ESG and node identifier generated by mxe_parser application. An example input is given in Figure 5.1 and the output files of this input are given in Figure 5.2.

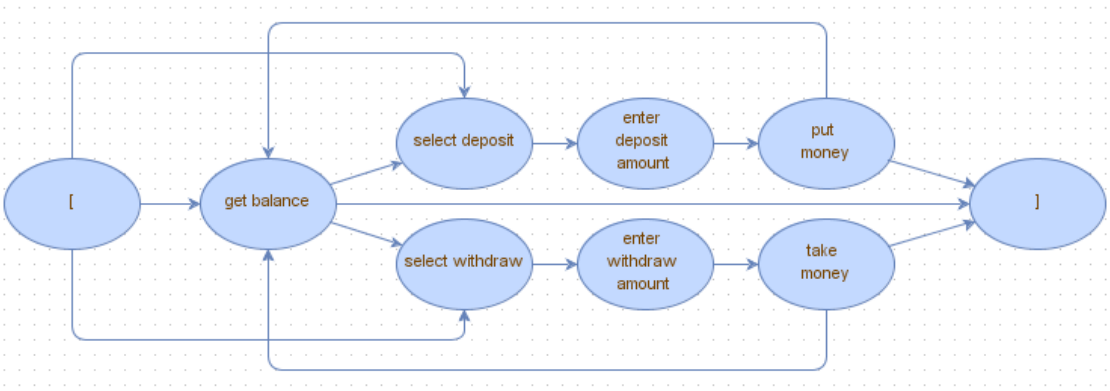


Figure 5.1. Bank Account-Base Product (A Sample ESG)

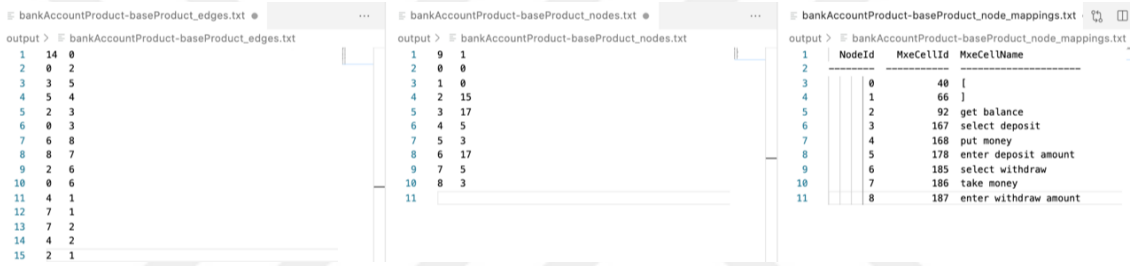


Figure 5.2. Generated Output Files of Sample Bank Account-Base Product

Fourth step is to train models. Two different model used in this step; one is SEAL Framework that uses GCN as neural network model and the other is DeepLinker that uses GAT as neural network model. To run on isolated environments, conda virtual environment (detailed information can be reached from https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/environments.html) for each workspace should be created. Before start using the virtual environments for SEAL-ESG python version should be set to 3.8 and 2.7 for the DeepLinker.

In the original SEAL and DeepLinker projects, extra tuning arguments were added, add-ons were made to use mxe_parser outputs, performance metrics were saved in csv files and some bug fixes were made. As such, they were published on github as SEAL-ESG and DeepLinker-ESG. Tuned parameters of SEAL and DeepLinker in experiments is given in Table 5.10.

Table 5.10. Parameters used on Experiments

	SEAL-ESG	DeepLinker-ESG
Batch size	X	X
Dropout Ratio	X	X
Number of Hidden Units	X	X
Learning Rate	X	X
Number of Epochs	X	X
Test Ratio	X	X
Hop	X	
SortPooling K	X	
Number of Attention		X
Weight Decay		X

5.2. Results

Parameter value table for SEAL and DeepLinker is listed in Table 5.11 and Table 5.12, respectively. In case of using combinations of all values entered in the parameter value table, it is necessary to run too many iterations. These combinations result in 576 iteration for SEAL and 648 iteration for DeepLinker. First five parameters (batch size, dropout, hidden units, learning rate, number of epochs) of each model are same, and those parameters are common for many neural networks.

Therefore, default values defined by the authors (SEAL by Zuhan, DeepLinker by Villafly) are used to decrease to total number of iterations to be run. For the remaining parameters (hop, max nodes per hop, sortpooling K parameters for SEAL and number of attentions and weight decay for DeepLinker), combinations of given values are considered. The values of parameters used in each iteration for the training of SEAL and DeepLinker models are listed in the Table 5.11. and Table 5.12. respectively.

Table 5.11. SEAL Parameters on Each Iteration (*batch-size 40 is for email dataset)

SEAL-ESG								
Iteration	Batch Size	Dropout	Hidden Units	Learning Rate	Epochs	Hops	Sortpooling k	Test Ratio
1	*50**	*0.5*	*128*	*0.001*	*50*	*1*	*0.6*	*0.2*
2	*25*	*0.5*	*128*	*0.001*	*50*	*1*	*0.6*	*0.2*
3	*10*	*0.5*	*128*	*0.001*	*50*	*1*	*0.6*	*0.2*
4	*1*	*0.5*	*128*	*0.001*	*50*	*1*	*0.6*	*0.2*
5	*50**	*0.5*	*128*	*0.0005*	*50*	*1*	*0.6*	*0.2*
6	*25*	*0.5*	*128*	*0.0005*	*50*	*1*	*0.6*	*0.2*
7	*10*	*0.5*	*128*	*0.0005*	*50*	*1*	*0.6*	*0.2*
8	*1*	*0.5*	*128*	*0.0005*	*50*	*1*	*0.6*	*0.2*
9	*50**	*0.5*	*128*	*0.0001*	*50*	*1*	*0.6*	*0.2*
10	*25*	*0.5*	*128*	*0.0001*	*50*	*1*	*0.6*	*0.2*
11	*10*	*0.5*	*128*	*0.0001*	*50*	*1*	*0.6*	*0.2*
12	*1*	*0.5*	*128*	*0.0001*	*50*	*1*	*0.6*	*0.2*
13	*50**	*0.5*	*128*	*0.001*	*50*	*1*	*0.6*	*0.2*
14	*25*	*0.5*	*128*	*0.001*	*50*	*1*	*0.6*	*0.2*
15	*10*	*0.5*	*128*	*0.001*	*50*	*1*	*0.6*	*0.2*
16	*1*	*0.5*	*128*	*0.001*	*50*	*1*	*0.6*	*0.2*
17	*50**	*0.5*	*64*	*0.0005*	*50*	*2*	*0.6*	*0.2*
18	*25*	*0.5*	*64*	*0.0005*	*50*	*2*	*0.6*	*0.2*
19	*10*	*0.5*	*64*	*0.0005*	*50*	*2*	*0.6*	*0.2*
20	*1*	*0.5*	*64*	*0.0005*	*50*	*2*	*0.6*	*0.2*
21	*50**	*0.5*	*64*	*0.0001*	*50*	*2*	*0.6*	*0.2*
22	*25*	*0.5*	*64*	*0.0001*	*50*	*2*	*0.6*	*0.2*
23	*10*	*0.5*	*64*	*0.0001*	*50*	*2*	*0.6*	*0.2*
24	*1*	*0.5*	*64*	*0.0001*	*50*	*2*	*0.6*	*0.2*
25	*50**	*0.5*	*64*	*0.0005*	*50*	*2*	*0.6*	*0.2*
26	*25*	*0.5*	*64*	*0.0005*	*50*	*2*	*0.6*	*0.2*
27	*10*	*0.5*	*64*	*0.0005*	*50*	*2*	*0.6*	*0.2*
28	*1*	*0.5*	*64*	*0.0005*	*50*	*2*	*0.6*	*0.2*
29	*50**	*0.5*	*64*	*0.0001*	*50*	*2*	*0.6*	*0.2*
30	*25*	*0.5*	*64*	*0.0001*	*50*	*2*	*0.6*	*0.2*
31	*10*	*0.5*	*64*	*0.0001*	*50*	*2*	*0.6*	*0.2*
32	*1*	*0.5*	*64*	*0.0001*	*50*	*2*	*0.6*	*0.2*

Table 5.12. DeepLinker Parameters on Each Iteration

DeepLinker-ESG								
Iteration	Batch Size	Dropout	Hidden Units	Learning Rate	Epochs	Number of Attention (K)	Test Ratio	Weight Decay
1	*16*	*0.5*	*32*	*0.001*	*50*	*2*	*0.2*	*0.001*
2	*16*	*0.5*	*32*	*0.001*	*50*	*2*	*0.2*	*0.0001*
3	*16*	*0.5*	*32*	*0.001*	*50*	*8*	*0.2*	*0.001*
4	*16*	*0.5*	*32*	*0.001*	*50*	*8*	*0.2*	*0.0001*
5	*16*	*0.5*	*32*	*0.0001*	*50*	*2*	*0.2*	*0.001*
6	*16*	*0.5*	*32*	*0.0001*	*50*	*2*	*0.2*	*0.0001*
7	*16*	*0.5*	*32*	*0.0001*	*50*	*8*	*0.2*	*0.001*
8	*16*	*0.5*	*32*	*0.0001*	*50*	*8*	*0.2*	*0.0001*
9	*16*	*0.5*	*32*	*0.0005*	*50*	*2*	*0.2*	*0.001*
10	*16*	*0.5*	*32*	*0.0005*	*50*	*2*	*0.2*	*0.0001*
11	*16*	*0.5*	*32*	*0.0005*	*50*	*8*	*0.2*	*0.001*
12	*16*	*0.5*	*32*	*0.0005*	*50*	*8*	*0.2*	*0.0001*
13	*32*	*0.5*	*32*	*0.001*	*50*	*2*	*0.2*	*0.001*
14	*32*	*0.5*	*32*	*0.001*	*50*	*2*	*0.2*	*0.0001*
15	*32*	*0.5*	*32*	*0.001*	*50*	*8*	*0.2*	*0.001*
16	*32*	*0.5*	*32*	*0.001*	*50*	*8*	*0.2*	*0.0001*
17	*32*	*0.5*	*32*	*0.0001*	*50*	*2*	*0.2*	*0.001*
18	*32*	*0.5*	*32*	*0.0001*	*50*	*2*	*0.2*	*0.0001*
19	*32*	*0.5*	*32*	*0.0001*	*50*	*8*	*0.2*	*0.001*
20	*32*	*0.5*	*32*	*0.0001*	*50*	*8*	*0.2*	*0.0001*
21	*32*	*0.5*	*32*	*0.0005*	*50*	*2*	*0.2*	*0.001*
22	*32*	*0.5*	*32*	*0.0005*	*50*	*2*	*0.2*	*0.0001*
23	*32*	*0.5*	*32*	*0.0005*	*50*	*8*	*0.2*	*0.001*
24	*32*	*0.5*	*32*	*0.0005*	*50*	*8*	*0.2*	*0.0001*

Best performed results of SEAL-ESG iterations of the all datasets are listed in Table 5.13. Performance of the SEAL-ESG trainings for each iteration measured by loss, accuracy and auc. Graphical representation of best performed iteration metrics are also listed in Table 5.13. The rest of the iteration results are given in APPENDIX A.

Table 5.13. SEAL Best Performed Iteration Results

Dataset	Iteration	SEAL-ESG –Best Iteration Results on Datasets								
		Training			Validation			Test		
		loss	acc	auc	loss	acc	auc	loss	acc	auc
Iselta	***5***	*0.360*	*0.856*	*0.921*	*0.306*	*0.875*	*0.957*	*0.462*	*0.800*	*0.884*
Student	***2***	*0.467*	*0.840*	*0.862*	*0.721*	*0.667*	*0.500*	*0.613*	*0.719*	*0.740*
Bank	***4***	*0.331*	*0.868*	*0.932*	*0.451*	*0.800*	*NaN*	*0.617*	*0.786*	*0.755*
Email	***4***	*0.085*	*0.977*	*NaN*	*0.085*	*NaN*	*NaN*	*0.378*	*0.900*	*0.920*

Results of DeepLinker -ESG iterations of the all datasets are listed in below tables. Performance of the DeepLinker-ESG trainings for each iteration measured by loss, accuracy and auc (only available for test). Graphical representation of best performed iteration metrics are also listed in Table 5.14. The rest of the iteration results are given in APPENDIX B.

Table 5.14. DeepLinker Best Performed Iteration Results

Dataset	Iteration	DeepLinker-ESG –Best Iteration Results on Datasets								
		Training			Validation			Test		
		loss	acc	auc	loss	acc	auc	loss	acc	auc
Iselta	**8**	*0.693*	*0.552*	*NaN*	*0.652*	*0.875*	*NaN*	*0.688*	*0.594*	*0.587*
Student	**8**	*0.684*	*0.574*	*0.000*	*0.646*	*0.571*	*0.000*	*0.683*	*0.625*	*0.625*
Bank	**8**	*0.675*	*0.643*	*NaN*	*0.671*	*0.667*	*NaN*	*0.681*	*0.781*	*0.703*
Email	**8**	*0.676*	*0.594*	*0.000*	*0.668*	*1.000*	*0.000*	*0.687*	*0.607*	*0.577*

5.3. Discussion

In this thesis, the experiments are performed on the datasets explained above. Each of these software models have their own specific domain and the components of these software models are observed to contain certain patterns. It is considered that these

patterns can be revealed through graph neural networks, which are specialized for the graph structured data. The experiments are performed under these considerations.

First impressions of the experimental results are as follows: when examining the results of the experiments performed on different datasets using SEAL and DeepLinker applications which are using two different variations of GNN and they use completely different architectural designs. SEAL uses DGCNN as a GNN model under the hood. It converts the link existence problem into a sub-graph classification problem by dividing a given graph into its sub-graphs (with samples created with negative and positive neighbors for each node). It performed much better than DeepLinker (uses GAT as a GNN model), which tries to solve the link existence problem by learning the hidden representations of nodes' relations with their neighbors.

Before evaluating the experimental results, it is necessary to briefly mention how the metrics are used in evaluating the results. *Auc* (area under curve) can be considered as summary of the model performance and gives the distribution of classes within the dataset for all classification thresholds. The wider the area under the *roc* (receiver operating characteristic) curve, the higher the model's ability to distinguish classes. *Auc* value of 0.5 means random estimation, the closer this value is to 1, the higher the model's ability to distinguish between classes. *Acc* (accuracy) is the basic performance metric that expresses how many of the observations made as a result of the model are correct, but most of the cases it is not sufficient to measure the performance of the model alone (for example, where the distribution of the dataset between classes is not balanced). *Loss* (cross entropy) gives the difference between the estimation made by the model and the actual value. Classification results generated by a neural network falls into [0,1] interval for each class. Based on the given input values to the neural network model, it assigns a value between [0,1] for each class. Among these assigned values, the class with the highest value or above the specified threshold is taken as the result. While the accuracy metric evaluates the results as true or false, the loss metric measures how far the value assigned by the model for the correct class is from 1.

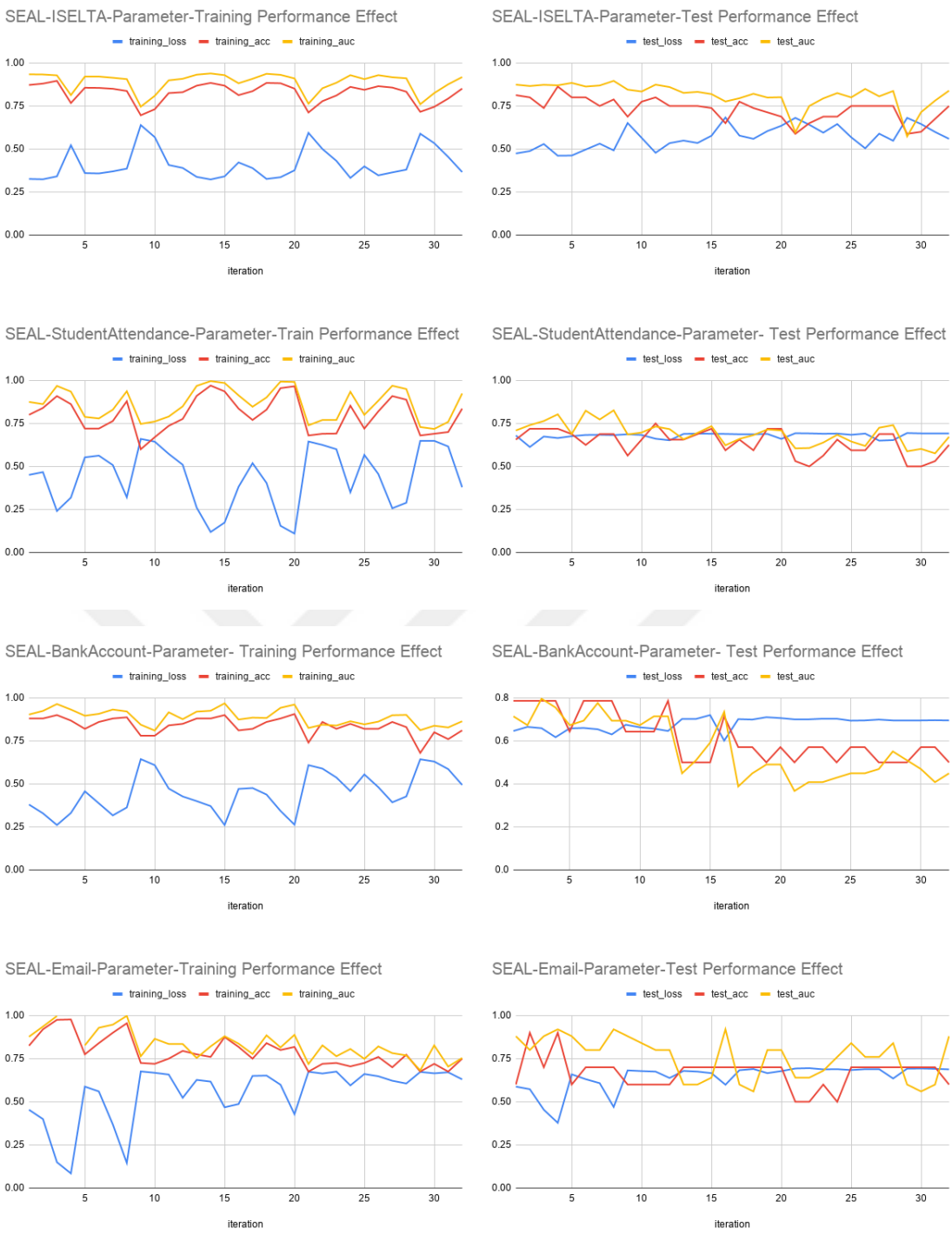


Figure 5.3. SEAL Performance Effects of Parameter Changes on Each Iteration

The performance outputs of the parameters used in SEAL iterations is given in Figure 5.3. Regardless of the size of the datasets, iterations 9, 21 and 29 show the worst performance. As the dataset size getting smaller, performance began to be negatively affected in all iterations between 17-32. When looking at the effects of the hop parameter

changes on performance, it can be said that all the first-order neighbors of a component belonging to a software model, the representation is learned best by DGCNN. A special case occurs in the 16th iteration, setting the batch size to minimum value 1 and learning rate to 0.001 (which is the biggest learning rate used in experiments) even the model overfits in large datasets, small positive effect was observed on performance in small datasets. When the iterations with the best results are examined, the performance is higher in the iterations 1, 2, 5, 8, 11 and 26 in large datasets, while the iterations 3, 4, 7, 12 and 16 showed higher performance in smaller iterations. When the results are examined in general, it has been observed that giving the batch size value as 1 increases the possibility of overfitting. It has been observed that changing the batch size value and the learning rate values inversely increases the performance. As the size of the dataset grows, using a larger value batch size and a smaller learning rate affects the performance positively.

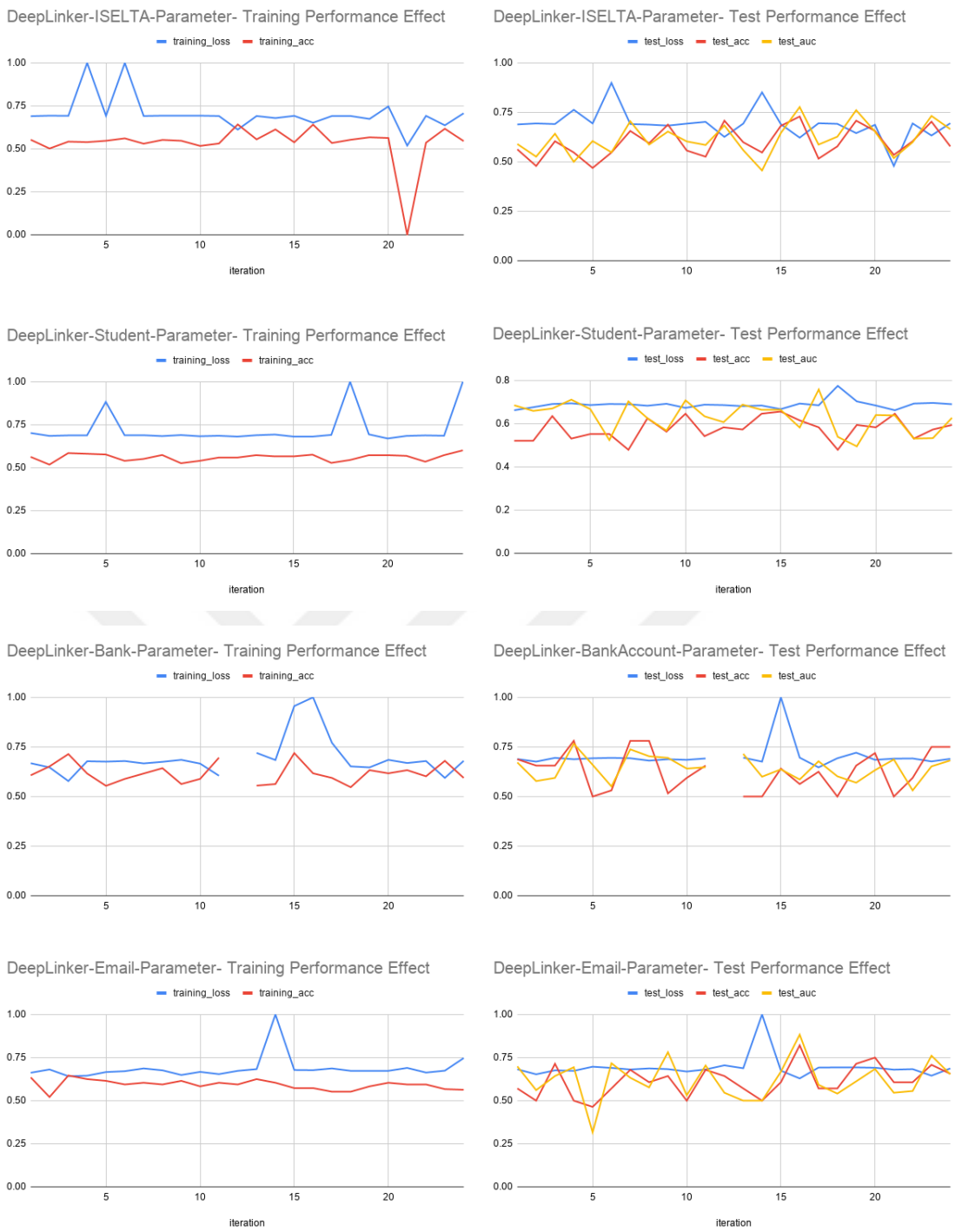


Figure 5.4. DeepLinker Performance Effects of Parameter Changes on Each Iteration

When the parameters used in DeepLinker iterations and the performance outputs of these parameters are compared in Figure 5.4, performance is distributed around 0.5, which is closed to random estimation, even if the tunings performed by changing the parameters, it makes a +-10% performance changes. While the model was being trained,

the distribution of the dataset between negative and positive classes (negative meaning no link and positive meaning there is a link between nodes) was made equally and at the same time, the distribution within batches was adjusted to be equal. In such a result, it can be thought that software models are relatively small models and there is not enough data for GAT to learn the relationships between nodes. When the datasets used in the article where GAT model is used for link prediction are examined, it is seen that large scale graphs are used. For example, the cora dataset used in the article [18] consists of 2708 nodes, 5429 edge and 1433 node features. On the other hand, ISELTA, which is the largest dataset used in this thesis, has 68 nodes, 249 edge and 1 node feature.

Figure 5.5. SEAL – Dataset size Performance Effect

As expected, as the size of the dataset increases, the learning and prediction performance of the model also increases as shown in Figure 5.5. On average, when the size of the dataset is increased from 100 to 250, the loss value decreases by 30% and the auc value increases by 12%. Dataset sizes have been listed in Table 5.9.

Experiments on two different machine learning models with 4 different datasets have shown that one of the best ways to understand how components used in software models is to form a pattern with neighboring components through the sub-graphs (in other words, micro-models) they create with the neighboring components, but not through the attributes of the component and the attributes of its neighboring components. In this way, even with relatively small datasets, success can be achieved.

One of the disadvantages of SEAL is that when a disconnected graph is given, it is not possible to make edge prediction from scratch (without any edge definition) since it cannot generate sub-graphs for this graph. However, this is possible with GNN model, which learns the feature representation of these nodes, and performs link prediction by learning the relationships between the hidden representation of the nodes, not the edges.

As a result, although GAT model showed higher performance than GCN models in large datasets, realizing graph similarity estimation approach using deep graph convolutional neural networks results in higher performance in predicting connections between components of software models.

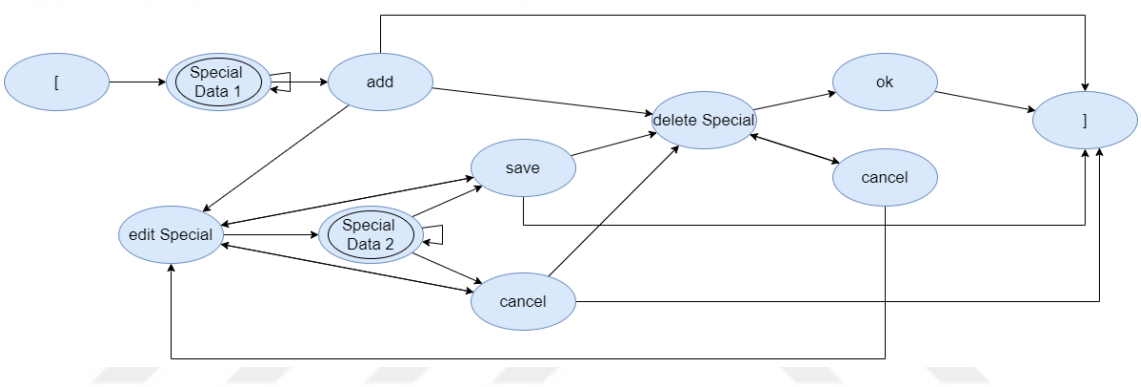


Figure 5.6. Original Specials ESG

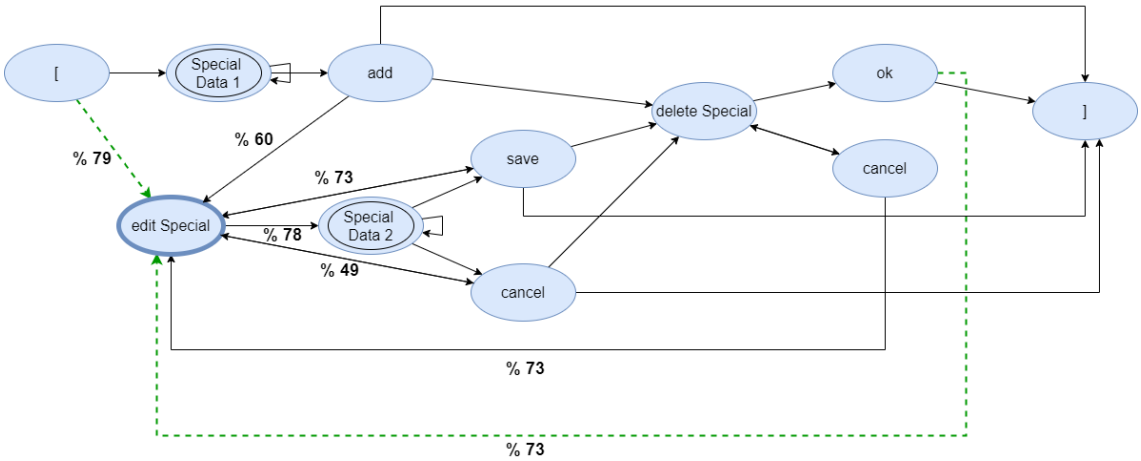


Figure 5.7. Specials ESG “edit Special” Node Qualitative Link Predictions

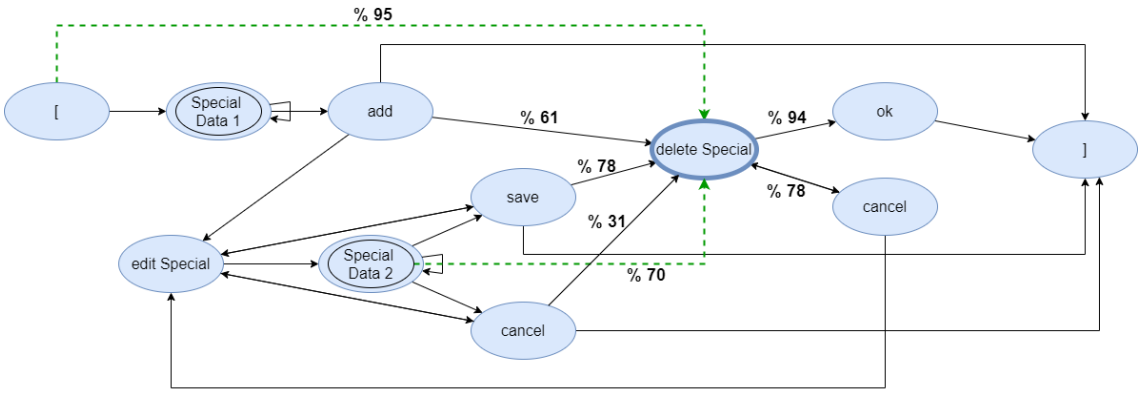


Figure 5-8 Specials ESG “delete Special” Node Qualitative Link Predictions.

Figure 5.7 and Figure 5.8 shows the qualitative link prediction results for specials ESG given in Figure 5.6. A SEAL-ESG model is trained using ISELTA dataset. Using this trained model two link prediction scenarios are executed for the nodes "edit Special" and "delete Special". Green dotted arrows are the new links predicted by the trained model those are not defined in original ESG. For the “edit Special” node, link predictions and the probabilities generated by trained model are listed in Table 5.15 given in the Figure 5.7. For the “delete Special” node, link predictions and the probabilities generated by trained model are also listed in Table 5.15 given in the Figure 5.8. The results suggested by the model can be evaluated as follows. There are two new possible connection suggestions for the ‘edit Special’ node with probabilities %79 and %73. These suggestions should be taken into consideration by the modeler. In addition, a suggestion with probability value %49 is presented for the connection between ‘edit Special’ and ‘cancel’. This suggestion may be thought as “don’t care” and the connection can be left as it is or removed by the discretion of the modeler. For the ‘delete Special’ node, it is seen that two new connections and one low-probability connection are offered. The connection from 'cancel' to 'delete Special' has a probability value %31, it may be considered as to break this existing connection entirely.

Table 5.15. Link Predictions Made by the Trained Model

Node	Link	Probability Of Existence
Edit Special	{‘[‘, ’edit Special’}	%79
	{ ‘Ok’, ‘edit Special’}	%73
	{‘add’, ‘edit Special’}	%60
	{‘edit Special’, ‘save’}	%73
	{‘edit Special’, ‘SpecialData2’}	%78
	{‘edit Special’, ‘cancel’}	%49
	{‘cancel’, ‘edit Special’}	%73
Delete Special	{‘[‘, ’delete Special’}	%95
	{‘delete Special’, ‘Ok’}	%94
	{‘delete Special’, ‘Cancel’}	%78
	{‘save’, ‘delete Special’}	%78
	{‘SpecialData2’, ‘delete Special’}	%70
	{‘add’, ‘delete Special’}	%61
	{‘cancel’, ‘delete Special’}	%31

5.4. Threats to Validity

Internal Validity: To make the studies in this thesis trustworthy, all the datasets are selected from different domains and different size of software applications. Two different GNN models which are applicable to link prediction problem, are selected for experiments. The performance of these models are measured with different set of parameter values. All the software applications modeled by ESG and drawn by TSD.

External Validity: It is unlikely to say that proposed method in this thesis will work on different software modelling tools and methods, even if it's possible. Considering a class diagram which is modeled with UML notation, they are heterogenous directed multi graphs but ESGs are homogenous directed graphs. Connection between classes has completely different meanings in comparison with ESGs.

CHAPTER 6
CONCLUSION AND FUTURE WORK

Enterprise software applications are generally sophisticated. Such systems can have many sub-systems and components in them. In order to design, configure, update or implement a software system, the details of the system are required to be understood at varying levels of management, design, implementation of the system. Typical software modeling systems may not be able to reduce this complexity for engineers. Predicting the connections between software components has great importance in modelling. From the software engineering perspective, deciding and connecting components with each other require significant effort. It is also error prone. Instead of putting all the workload on the software engineer, giving some recommendation can help engineers with modeling the composition and interaction of components in a software system.

The method proposed in this thesis is to help software engineers on software modelling. The modeling technique used in this thesis is ESG, which is used to model transition between GUI components. The goal of this thesis is to propose a method that finds missing or forgotten transitions between components defined in ESG. Graph neural network models (GNN) are used to solve this problem. Selected GNN variations are graph convolutional neural networks (GCN) and graph attention neural networks (GAT). Steps of the process to find missing links between components are listed as follows: (i) find the ESG models, (ii) transform ESG models into graph structured data and extract features of the components, (iii) train the GNN model, (iv) evaluate the performance of the trained model. Experiments are performed on different datasets with different GNN models. The results show that there are hidden patterns between ESG components. It is possible to extract them over machine learning algorithms, which are specialized for graphs, and make recommendations on missing links or edges of the graph-based system model.

This thesis is focused on ESG models to find missing links between components. Four datasets are used in this study. Diversifying datasets and evaluating their results in larger datasets could be the of the subject to future studies. There are many software modelling tools and methods in literature. Other methods used for software modeling could be worked on in the future. As another application area, it can be used to increase the accuracy of models created automatically with the ripping method.

REFERENCES

[1] G. ROZENBERG, *Handbook of graph grammars and computing by graph transformation*. World Scientific, 1997.

[2] F. Belli, "Finite state testing and analysis of graphical user interfaces," in *Proceedings 12th International Symposium on Software Reliability Engineering*, Hong Kong, China, 2001, pp. 34–43, doi: 10.1109/ISSRE.2001.989456.

[3] F. Belli, C. J. Budnik, and L. White, "Event-based modelling, analysis and testing of user interactions: approach and case study," *Softw. Test. Verification Reliab.*, vol. 16, no. 1, pp. 3–32, Mar. 2006, doi: 10.1002/stvr.335.

[4] D. Liben-Nowell and J. Kleinberg, "The Link-Prediction Problem for Social Networks," p. 23.

[5] L. A. Adamic and E. Adar, "Friends and neighbors on the Web," *Soc. Netw.*, vol. 25, no. 3, pp. 211–230, Jul. 2003, doi: 10.1016/S0378-8733(03)00009-1.

[6] E. M. Airoldi, D. M. Blei, S. E. Fienberg, and E. P. Xing, "Mixed Membership Stochastic Blockmodels," p. 34.

[7] Y. Koren, R. Bell, and C. Volinsky, "Matrix Factorization Techniques for Recommender Systems," *Computer*, vol. 42, no. 8, pp. 30–37, Aug. 2009, doi: 10.1109/MC.2009.263.

[8] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich, "A Review of Relational Machine Learning for Knowledge Graphs," *ArXiv150300759 Cs Stat*, Sep. 2015, doi: 10.1109/JPROC.2015.2483592.

[9] J.-B. Cordonnier and A. Loukas, "Extrapolating paths with graph neural networks," *ArXiv190307518 Cs Stat*, Mar. 2019, Accessed: Oct. 13, 2020. [Online]. Available: http://arxiv.org/abs/1903.07518.

[10] T. Oyetunde, M. Zhang, Y. Chen, Y. Tang, and C. Lo, "BoostGAPFILL: improving the fidelity of metabolic network reconstructions through integrated constraint and pattern-based methods," *Bioinformatics*, p. btw684, Oct. 2016, doi: 10.1093/bioinformatics/btw684.

[11] F. Scarselli, M. Gori, Ah Chung Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009, doi: 10.1109/TNN.2008.2005605.

[12] M. Niepert, M. Ahmed, and K. Kutzkov, “Learning Convolutional Neural Networks for Graphs,” *ArXiv160505273 Cs Stat*, Jun. 2016, Accessed: Oct. 22, 2020. [Online]. Available: http://arxiv.org/abs/1605.05273.

[13] M. Defferrard, X. Bresson, and P. Vandergheynst, “Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering,” *ArXiv160609375 Cs Stat*, Feb. 2017, Accessed: Oct. 15, 2020. [Online]. Available: http://arxiv.org/abs/1606.09375.

[14] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D.-Y. Yeung, “GaAN: Gated Attention Networks for Learning on Large and Spatiotemporal Graphs,” *ArXiv180307294 Cs*, Mar. 2018, Accessed: Oct. 22, 2020. [Online]. Available: http://arxiv.org/abs/1803.07294.

[15] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph Attention Networks,” *ArXiv171010903 Cs Stat*, Feb. 2018, Accessed: Oct. 15, 2020. [Online]. Available: http://arxiv.org/abs/1710.10903.

[16] M. Zhang and Y. Chen, “Weisfeiler-Lehman Neural Machine for Link Prediction,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Halifax NS Canada, Aug. 2017, pp. 575–583, doi: 10.1145/3097983.3097996.

[17] M. Zhang and Y. Chen, “Link Prediction Based on Graph Neural Networks,” p. 18.

[18] W. Gu, F. Gao, X. Lou, and J. Zhang, “Link Prediction via Graph Attention Network,” *ArXiv191004807 Cs*, Oct. 2019, Accessed: Oct. 13, 2020. [Online]. Available: http://arxiv.org/abs/1910.04807.

[19] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain.,” *Psychol. Rev.*, vol. 65, no. 6, pp. 386–408, 1958, doi: 10.1037/h0042519.

[20] S. K. Pal and S. Mitra, “Multilayer perceptron, fuzzy sets, and classification,” *IEEE Trans. Neural Netw.*, vol. 3, no. 5, pp. 683–697, Sep. 1992, doi: 10.1109/72.159058.

[21] Y. LeCun, Y. Bengio, and T. B. Laboratories, “Convolutional Networks for Images, Speech, and Time-Series,” p. 15.

[22] S. Kombrink, T. Mikolov, M. Karafiat, and L. Burget, “Recurrent Neural Network Based Language Modeling in Meeting Recognition,” p. 4.

[23] J. L. ELMAN, “Finding structure in time,” 1990.

[24] S. Hochreiter and J. Schmidhuber, “LONG SHORT-TERM MEMORY,” 1997.

[25] H. Bourlard, “Auto-association by multilayer perceptrons and singular value decomposition,” 2000.

[26] I. Goodfellow *et al.*, “Generative Adversarial Nets,” p. 9.

[27] F. Harary and G. Gupta, “Dynamic graph models,” *Math. Comput. Model.*, vol. 25, no. 7, pp. 79–87, Apr. 1997, doi: 10.1016/S0895-7177(97)00050-2.

[28] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, “Graph Kernels,” 2010.

[29] N. Shervashidze, “Weisfeiler-Lehman Graph Kernels,” p. 23.

[30] D. K. Hammond, P. Vandergheynst, and R. Gribonval, “Wavelets on graphs via spectral graph theory,” *Appl. Comput. Harmon. Anal.*, vol. 30, no. 2, pp. 129–150, Mar. 2011, doi: 10.1016/j.acha.2010.04.005.

[31] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, “An End-to-End Deep Learning Architecture for Graph Classification,” p. 8, 2018.

[32] W. L. Hamilton, R. Ying, and J. Leskovec, “Inductive Representation Learning on Large Graphs,” p. 19.

[33] J. Chen, T. Ma, and C. Xiao, “FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling,” *ArXiv180110247 Cs*, Jan. 2018, Accessed: Oct. 25, 2020. [Online]. Available: http://arxiv.org/abs/1801.10247.

[34] Z. Lin *et al.*, “A Structured Self-attentive Sentence Embedding,” *ArXiv170303130 Cs*, Mar. 2017, Accessed: Oct. 25, 2020. [Online]. Available: http://arxiv.org/abs/1703.03130.

[35] A. Vaswani *et al.*, “Attention is All you Need,” p. 11.

[36] B. Perozzi, R. Al-Rfou, and S. Skiena, “DeepWalk: Online Learning of Social Representations,” *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min. - KDD 14*, pp. 701–710, 2014, doi: 10.1145/2623330.2623732.

[37] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, “LINE: Large-scale Information Network Embedding,” *ArXiv150303578 Cs*, Mar. 2015, doi: 10.1145/2736277.2741093.

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

[39] T. N. Kipf and M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” *ArXiv160902907 Cs Stat*, Feb. 2017, Accessed: Oct. 15, 2020. [Online]. Available: http://arxiv.org/abs/1609.02907.

[40] J. Kim, T. Kim, S. Kim, and C. D. Yoo, “Edge-Labeling Graph Neural Network for Few-Shot Learning,” p. 10.

[41] L. Gong and Q. Cheng, “Exploiting Edge Features for Graph Neural Networks,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Long Beach, CA, USA, Jun. 2019, pp. 9203–9211, doi: 10.1109/CVPR.2019.00943.

[42] István A. Kovács *et al.*, “Network-based prediction of protein interactions,” 2019, doi: https://doi.org/10.1038/s41467-019-09177-y.

[43] B. Shneiderman, *Designing the User Interface*. 1998.

[44] D. Öztürk, “A MODEL-BASED TEST GENERATION APPROACH FOR AGILE SOFTWARE PRODUCT LINES,” M.Sc. Thesis, İzmir Institute of Technology, 2020.

[45] T. Tuglular, F. Belli, and M. Linschulte, “Input Contract Testing of Graphical User Interfaces,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 26, no. 02, pp. 183–215, Mar. 2016, doi: 10.1142/S0218194016500091.

[46] Y. Bengio, R. Ducharme, and P. Vincent, “A Neural Probabilistic Language Model,” p. 7.

[47] M. Zhang, *SEAL*. 2018.

APPENDIX A
SEAL – ESG DATASET RESULTS

Table A.1. SEAL-ESG ISELTA Dataset Iteration Best Results

Iteration	SEAL-ESG – ISELTA Dataset								
	Training			Validation			Test		
	loss	acc	auc	loss	acc	auc	loss	acc	auc
1	***0.326***	***0.872***	***0.934***	***0.267***	***0.938***	***0.969***	***0.474***	***0.813***	***0.874***
2	*0.324*	*0.880*	*0.933*	*0.317*	*0.875*	*0.949*	*0.488*	*0.800*	*0.866*
3	*0.341*	*0.896*	*0.928*	*0.351*	*0.906*	*0.953*	*0.529*	*0.738*	*0.873*
4	*0.522*	*0.767*	*0.813*	*0.482*	*0.813*	*0.859*	*0.461*	*0.863*	*0.870*
5	***0.360***	***0.856***	***0.921***	***0.306***	***0.875***	***0.957***	***0.462***	***0.800***	***0.884***
6	*0.358*	*0.855*	*0.921*	*0.298*	*0.875*	*0.953*	*0.497*	*0.800*	*0.863*
7	*0.370*	*0.850*	*0.914*	*0.324*	*0.875*	*0.965*	*0.531*	*0.750*	*0.869*
8	*0.386*	*0.837*	*0.906*	*0.339*	*0.906*	*0.961*	*0.492*	*0.788*	*0.896*
9	*0.639*	*0.696*	*0.746*	*0.627*	*0.719*	*0.805*	*0.651*	*0.688*	*0.845*
10	*0.569*	*0.731*	*0.809*	*0.545*	*0.750*	*0.836*	*0.565*	*0.775*	*0.834*
11	*0.407*	*0.825*	*0.899*	*0.336*	*0.844*	*0.945*	*0.478*	*0.800*	*0.874*
12	*0.390*	*0.830*	*0.908*	*0.299*	*0.906*	*0.965*	*0.534*	*0.750*	*0.860*
13	*0.338*	*0.868*	*0.932*	*0.369*	*0.844*	*0.918*	*0.549*	*0.750*	*0.826*
14	*0.323*	*0.884*	*0.939*	*0.384*	*0.844*	*0.922*	*0.535*	*0.750*	*0.832*
15	*0.341*	*0.868*	*0.929*	*0.392*	*0.844*	*0.918*	*0.577*	*0.738*	*0.819*
16	*0.422*	*0.813*	*0.882*	*0.453*	*0.781*	*0.879*	*0.683*	*0.650*	*0.776*
17	*0.389*	*0.836*	*0.908*	*0.429*	*0.750*	*0.871*	*0.578*	*0.775*	*0.795*
18	*0.326*	*0.884*	*0.937*	*0.397*	*0.844*	*0.922*	*0.559*	*0.738*	*0.821*
19	*0.336*	*0.882*	*0.931*	*0.400*	*0.844*	*0.898*	*0.604*	*0.713*	*0.799*
20	*0.376*	*0.851*	*0.910*	*0.410*	*0.844*	*0.922*	*0.635*	*0.688*	*0.801*
21	*0.594*	*0.712*	*0.763*	*0.689*	*0.688*	*0.660*	*0.681*	*0.588*	*0.597*
22	*0.500*	*0.778*	*0.853*	*0.606*	*0.688*	*0.754*	*0.641*	*0.650*	*0.750*
23	*0.431*	*0.811*	*0.885*	*0.492*	*0.688*	*0.848*	*0.595*	*0.688*	*0.794*
24	*0.332*	*0.861*	*0.929*	*0.449*	*0.813*	*0.902*	*0.645*	*0.688*	*0.825*
25	*0.399*	*0.844*	*0.906*	*0.420*	*0.813*	*0.879*	*0.568*	*0.750*	*0.800*
26	*0.347*	*0.865*	*0.929*	*0.377*	*0.844*	*0.918*	*0.504*	*0.750*	*0.849*
27	*0.364*	*0.857*	*0.917*	*0.411*	*0.844*	*0.922*	*0.589*	*0.750*	*0.806*
28	*0.380*	*0.833*	*0.911*	*0.358*	*0.875*	*0.945*	*0.548*	*0.750*	*0.837*

(cont. on next page)

Table A.1. (cont.)									
29	*0.589*	*0.716*	*0.760*	*0.691*	*0.656*	*0.637*	*0.681*	*0.588*	*0.573*
30	*0.534*	*0.745*	*0.825*	*0.625*	*0.625*	*0.758*	*0.645*	*0.600*	*0.714*
31	*0.454*	*0.793*	*0.876*	*0.501*	*0.750*	*0.848*	*0.599*	*0.675*	*0.781*
32	*0.366*	*0.851*	*0.918*	*0.397*	*0.844*	*0.918*	*0.559*	*0.750*	*0.839*

Figure A.1. SEAL - ISELTA Dataset – Training – Iteration.1

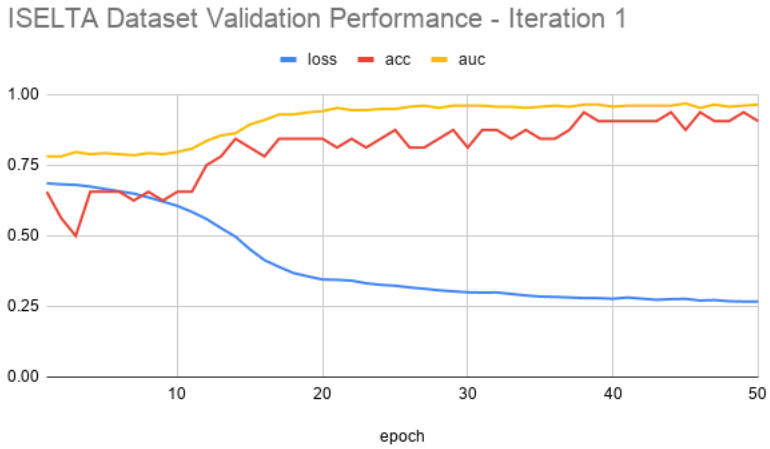


Figure A.2. SEAL - ISELTA Dataset – Validation – Iteration.1

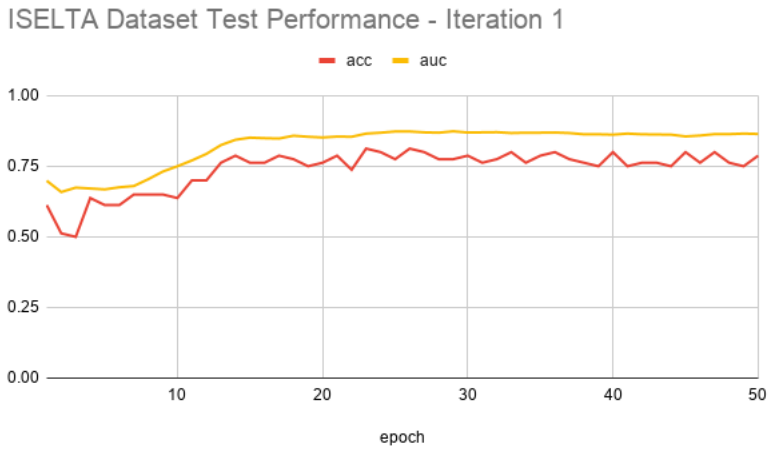


Figure A.3. SEAL - ISELTA Dataset – Test – Iteration.1

Table A.2. SEAL-ESG Bank Account Dataset Iteration Best Results

Iteration	SEAL-ESG – Bank Account Dataset								
	Training			Validation			Test		
	loss	acc	auc	loss	acc	auc	loss	acc	auc
1	*0.380*	*0.880*	*0.903*	*0.695*	*0.400*	*NaN*	*0.646*	*0.786*	*0.714*
2	***0.329***	***0.880***	***0.924***	***0.696***	***0.400***	***NaN***	***0.665***	***0.786***	***0.673***
3	*0.261*	*0.900*	*0.964*	*0.745*	*0.600*	*NaN*	*0.659*	*0.786*	*0.796*
4	*0.331*	*0.868*	*0.932*	*0.451*	*0.800*	*NaN*	*0.617*	*0.786*	*0.755*
5	*0.457*	*0.820*	*0.896*	*0.686*	*0.400*	*NaN*	*0.658*	*0.643*	*0.673*
6	*0.387*	*0.860*	*0.907*	*0.687*	*0.400*	*NaN*	*0.660*	*0.786*	*0.694*
7	*0.317*	*0.880*	*0.932*	*0.711*	*0.400*	*NaN*	*0.654*	*0.786*	*0.776*
8	*0.363*	*0.887*	*0.921*	*0.734*	*0.400*	*NaN*	*0.630*	*0.786*	*0.694*
9	*0.644*	*0.780*	*0.844*	*0.678*	*NaN*	*NaN*	*0.675*	*0.643*	*0.694*
10	*0.609*	*0.780*	*0.810*	*0.679*	*NaN*	*NaN*	*0.663*	*0.643*	*0.673*
11	*0.473*	*0.840*	*0.916*	*0.683*	*0.600*	*NaN*	*0.657*	*0.643*	*0.714*
12	*0.427*	*0.849*	*0.876*	*0.701*	*0.400*	*NaN*	*0.646*	*0.786*	*0.714*
13	*0.400*	*0.880*	*0.920*	*0.616*	*0.600*	*NaN*	*0.702*	*0.500*	*0.449*
14	*0.371*	*0.880*	*0.925*	*0.519*	*0.800*	*NaN*	*0.702*	*0.500*	*0.510*
15	*0.261*	*0.900*	*0.968*	*0.465*	*0.600*	*NaN*	*0.720*	*0.500*	*0.592*
16	*0.471*	*0.811*	*0.874*	*0.330*	*0.800*	*NaN*	*0.601*	*0.714*	*0.735*
17	*0.476*	*0.820*	*0.885*	*0.653*	*0.600*	*NaN*	*0.701*	*0.571*	*0.388*

(cont. on next page)

Table A.2. (cont.)									
18	*0.438*	*0.860*	*0.883*	*0.605*	*0.600*	*NaN*	*0.699*	*0.571*	*0.449*
19	*0.343*	*0.880*	*0.942*	*0.533*	*0.600*	*NaN*	*0.710*	*0.500*	*0.490*
20	*0.263*	*0.906*	*0.961*	*0.362*	*NaN*	*NaN*	*0.706*	*0.571*	*0.490*
21	*0.609*	*0.740*	*0.825*	*0.728*	*0.000*	*NaN*	*0.700*	*0.500*	*0.367*
22	*0.589*	*0.860*	*0.844*	*0.724*	*0.400*	*NaN*	*0.700*	*0.571*	*0.408*
23	*0.537*	*0.820*	*0.839*	*0.718*	*0.600*	*NaN*	*0.703*	*0.571*	*0.408*
24	*0.458*	*0.849*	*0.864*	*0.534*	*0.600*	*NaN*	*0.703*	*0.500*	*0.429*
25	*0.555*	*0.820*	*0.846*	*0.671*	*0.600*	*NaN*	*0.694*	*0.571*	*0.449*
26	*0.481*	*0.820*	*0.862*	*0.623*	*0.800*	*NaN*	*0.695*	*0.571*	*0.449*
27	*0.393*	*0.860*	*0.899*	*0.532*	*0.800*	*NaN*	*0.699*	*0.500*	*0.469*
28	*0.427*	*0.830*	*0.901*	*0.418*	*0.800*	*NaN*	*0.695*	*0.500*	*0.551*
29	*0.643*	*0.680*	*0.812*	*0.722*	*0.000*	*NaN*	*0.695*	*0.500*	*0.510*
30	*0.630*	*0.800*	*0.838*	*0.722*	*0.000*	*NaN*	*0.695*	*0.571*	*0.469*
31	*0.586*	*0.760*	*0.828*	*0.729*	*0.400*	*NaN*	*0.696*	*0.571*	*0.408*
32	*0.494*	*0.811*	*0.864*	*0.586*	*0.800*	*NaN*	*0.695*	*0.500*	*0.449*

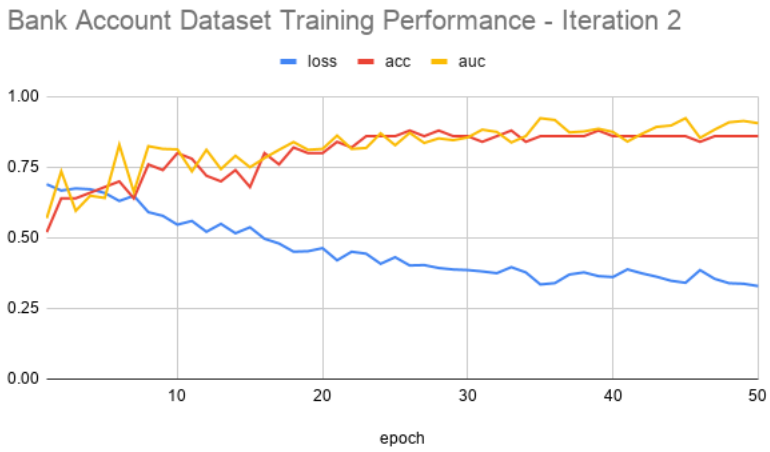


Figure A.4. SEAL – Bank Account Dataset – Training – Iteration.2

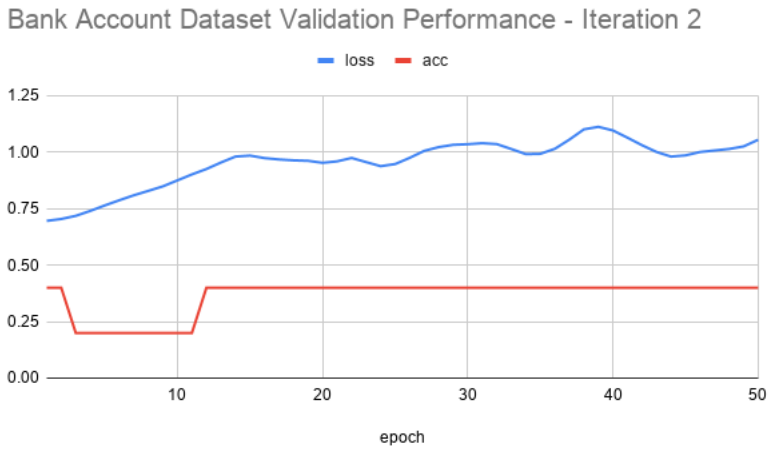


Figure A.5. SEAL – Bank Account Dataset – Validation – Iteration.2

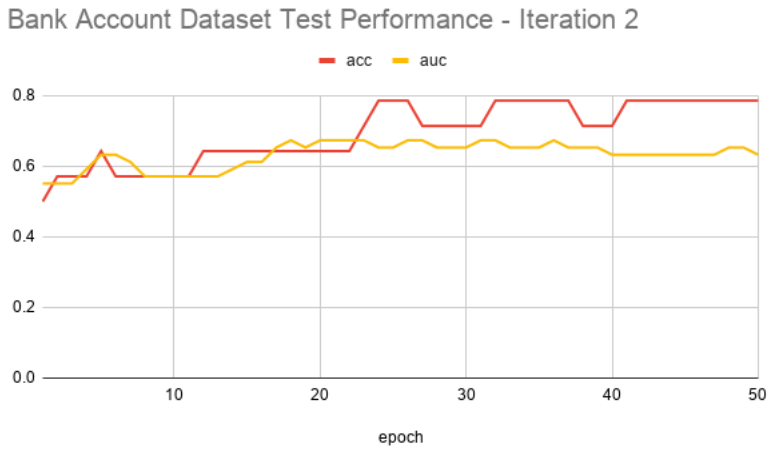


Figure A.6. SEAL – Bank Account Dataset – Test – Iteration.2

Table A.3. SEAL-ESG Email Dataset Iteration Best Results

Iteration	SEAL-ESG – Email Dataset								
	Training			Validation			Test		
	loss	acc	auc	loss	acc	auc	loss	acc	auc
1	*0.454*	*0.825*	*0.877*	*0.326*	*NaN*	*NaN*	*0.588*	*0.600*	*0.880*
2	***0.400***	***0.920***	***0.936***	***0.371***	***NaN***	***NaN***	***0.573***	***0.900***	***0.800***
3	*0.150*	*0.975*	*0.997*	*0.074*	*NaN*	*NaN*	*0.454*	*0.700*	*0.880*
4	*0.085*	*0.977*	*NaN*	*0.085*	*NaN*	*NaN*	*0.378*	*0.900*	*0.920*
5	*0.588*	*0.775*	*0.827*	*0.541*	*0.750*	*NaN*	*0.659*	*0.600*	*0.880*
6	*0.560*	*0.840*	*0.929*	*0.565*	*0.750*	*NaN*	*0.631*	*0.700*	*0.800*
7	*0.369*	*0.900*	*0.947*	*0.270*	*NaN*	*NaN*	*0.608*	*0.700*	*0.800*
8	*0.146*	*0.955*	*0.998*	*0.022*	*NaN*	*NaN*	*0.470*	*0.700*	*0.920*
9	*0.676*	*0.725*	*0.764*	*0.674*	*0.500*	*NaN*	*0.682*	*0.600*	*0.880*
10	*0.668*	*0.720*	*0.865*	*0.671*	*0.500*	*NaN*	*0.678*	*0.600*	*0.840*
11	*0.658*	*0.750*	*0.835*	*0.647*	*0.750*	*NaN*	*0.675*	*0.600*	*0.800*
12	*0.523*	*0.795*	*0.835*	*0.391*	*NaN*	*NaN*	*0.638*	*0.600*	*0.800*
13	*0.627*	*0.775*	*0.754*	*0.690*	*0.750*	*0.750*	*0.679*	*0.700*	*0.600*
14	*0.617*	*0.760*	*0.821*	*0.673*	*0.750*	*0.750*	*0.675*	*0.700*	*0.600*
15	*0.468*	*0.875*	*0.880*	*0.371*	*0.750*	*NaN*	*0.666*	*0.700*	*0.640*
16	*0.487*	*0.818*	*0.837*	*0.559*	*0.750*	*NaN*	*0.599*	*0.700*	*0.920*
17	*0.650*	*0.750*	*0.777*	*0.688*	*0.750*	*0.750*	*0.683*	*0.700*	*0.600*
18	*0.652*	*0.840*	*0.885*	*0.680*	*0.750*	*0.750*	*0.690*	*0.700*	*0.560*
19	*0.599*	*0.800*	*0.815*	*0.650*	*0.750*	*0.750*	*0.666*	*0.700*	*0.800*
20	*0.428*	*0.818*	*0.888*	*0.522*	*0.750*	*NaN*	*0.678*	*0.700*	*0.800*
21	*0.674*	*0.675*	*0.719*	*0.693*	*0.500*	*0.500*	*0.693*	*0.500*	*0.640*
22	*0.664*	*0.720*	*0.827*	*0.691*	*0.500*	*0.500*	*0.695*	*0.500*	*0.640*
23	*0.675*	*0.725*	*0.764*	*0.691*	*0.750*	*0.750*	*0.688*	*0.600*	*0.680*
24	*0.595*	*0.705*	*0.806*	*0.689*	*0.750*	*0.750*	*0.689*	*0.500*	*0.760*
25	*0.661*	*0.725*	*0.749*	*0.690*	*0.750*	*0.750*	*0.684*	*0.700*	*0.840*
26	*0.648*	*0.760*	*0.821*	*0.689*	*0.750*	*0.750*	*0.689*	*0.700*	*0.760*
27	*0.622*	*0.700*	*0.782*	*0.621*	*0.750*	*0.750*	*0.689*	*0.700*	*0.760*
28	*0.606*	*0.773*	*0.769*	*0.657*	*0.750*	*0.750*	*0.635*	*0.700*	*0.840*
29	*0.673*	*0.675*	*0.682*	*0.689*	*0.750*	*0.750*	*0.692*	*0.700*	*0.600*
30	*0.665*	*0.720*	*0.827*	*0.689*	*0.750*	*0.500*	*0.693*	*0.700*	*0.560*
31	*0.671*	*0.675*	*0.704*	*0.689*	*0.750*	*0.750*	*0.691*	*0.700*	*0.600*
32	*0.630*	*0.750*	*0.754*	*0.689*	*0.750*	*0.750*	*0.688*	*0.600*	*0.880*

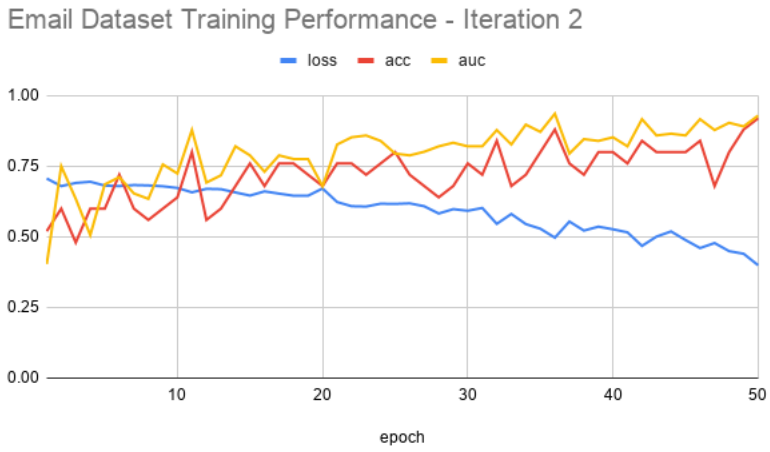


Figure A.7. SEAL – Email Dataset – Training – Iteration.2

Figure A.8. SEAL – Email Dataset – Validation – Iteration.2

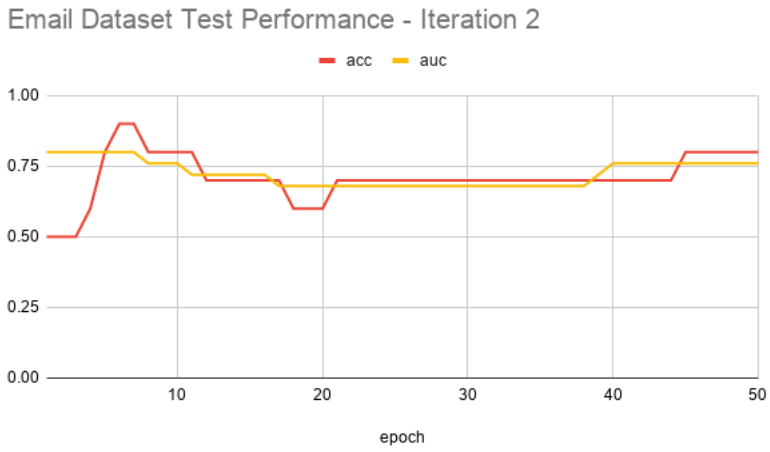


Figure A.9. SEAL – Email Dataset – Test – Iteration.2

Table A.4. SEAL-ESG Student Attendance Dataset Iteration Best Results

Iteration	SEAL-ESG – Student Attendance Dataset								
	Training			Validation			Test		
	loss	acc	auc	loss	acc	auc	loss	acc	auc
1	*0.451*	*0.800*	*0.875*	*0.564*	*0.667*	*0.844*	*0.680*	*0.656*	*0.709*
2	***0.467***	***0.840***	***0.862***	***0.721***	***0.667***	***0.500***	***0.613***	***0.719***	***0.740***
3	*0.241*	*0.909*	*0.968*	*0.473*	*0.833*	*0.906*	*0.674*	*0.719*	*0.764*
4	*0.319*	*0.862*	*0.935*	*0.416*	*0.917*	*0.906*	*0.665*	*0.719*	*0.803*
5	*0.553*	*0.720*	*0.788*	*0.719*	*0.583*	*0.719*	*0.676*	*0.688*	*0.689*
6	*0.563*	*0.720*	*0.779*	*0.708*	*0.583*	*0.625*	*0.683*	*0.625*	*0.824*
7	*0.508*	*0.764*	*0.830*	*0.702*	*0.583*	*0.813*	*0.684*	*0.688*	*0.773*
8	*0.321*	*0.879*	*0.937*	*0.408*	*1.000*	*1.000*	*0.682*	*0.688*	*0.826*
9	*0.660*	*0.600*	*0.747*	*0.721*	*0.333*	*0.563*	*0.687*	*0.563*	*0.686*
10	*0.644*	*0.670*	*0.761*	*0.721*	*0.417*	*0.594*	*0.683*	*0.656*	*0.697*
11	*0.573*	*0.736*	*0.790*	*0.716*	*0.667*	*0.563*	*0.661*	*0.750*	*0.732*
12	*0.510*	*0.776*	*0.849*	*0.537*	*0.750*	*0.813*	*0.652*	*0.656*	*0.717*
13	*0.261*	*0.910*	*0.968*	*0.498*	*0.833*	*0.875*	*0.689*	*0.656*	*0.658*
14	*0.119*	*0.970*	*0.996*	*0.507*	*0.833*	*0.844*	*0.691*	*0.688*	*0.695*
15	*0.174*	*0.936*	*0.985*	*0.389*	*0.833*	*0.938*	*0.690*	*0.719*	*0.734*
16	*0.382*	*0.836*	*0.913*	*0.605*	*0.833*	*0.906*	*0.689*	*0.594*	*0.623*
17	*0.519*	*0.770*	*0.847*	*0.520*	*0.750*	*0.781*	*0.687*	*0.656*	*0.660*
18	*0.404*	*0.830*	*0.901*	*0.558*	*0.833*	*0.750*	*0.687*	*0.594*	*0.682*
19	*0.155*	*0.955*	*0.993*	*0.289*	*0.917*	*1.000*	*0.689*	*0.719*	*0.715*
20	*0.110*	*0.966*	*0.991*	*0.371*	*0.917*	*0.938*	*0.660*	*0.719*	*0.709*
21	*0.645*	*0.680*	*0.739*	*0.683*	*0.583*	*0.719*	*0.693*	*0.531*	*0.605*
22	*0.625*	*0.690*	*0.770*	*0.665*	*0.667*	*0.688*	*0.692*	*0.500*	*0.607*
23	*0.600*	*0.691*	*0.770*	*0.603*	*0.750*	*0.750*	*0.690*	*0.563*	*0.639*
24	*0.350*	*0.853*	*0.934*	*0.545*	*0.833*	*0.781*	*0.691*	*0.656*	*0.684*
25	*0.567*	*0.720*	*0.800*	*0.613*	*0.667*	*0.719*	*0.684*	*0.594*	*0.645*
26	*0.456*	*0.820*	*0.882*	*0.621*	*0.833*	*0.719*	*0.691*	*0.594*	*0.619*
27	*0.257*	*0.909*	*0.969*	*0.388*	*0.917*	*0.938*	*0.650*	*0.688*	*0.725*
28	*0.289*	*0.888*	*0.950*	*0.456*	*0.833*	*0.844*	*0.653*	*0.688*	*0.740*
29	*0.649*	*0.680*	*0.729*	*0.705*	*0.500*	*0.750*	*0.694*	*0.500*	*0.588*
30	*0.649*	*0.690*	*0.718*	*0.686*	*0.500*	*0.750*	*0.692*	*0.500*	*0.602*
31	*0.616*	*0.700*	*0.759*	*0.664*	*0.583*	*0.781*	*0.692*	*0.531*	*0.576*
32	*0.379*	*0.836*	*0.924*	*0.593*	*0.833*	*0.750*	*0.692*	*0.625*	*0.672*

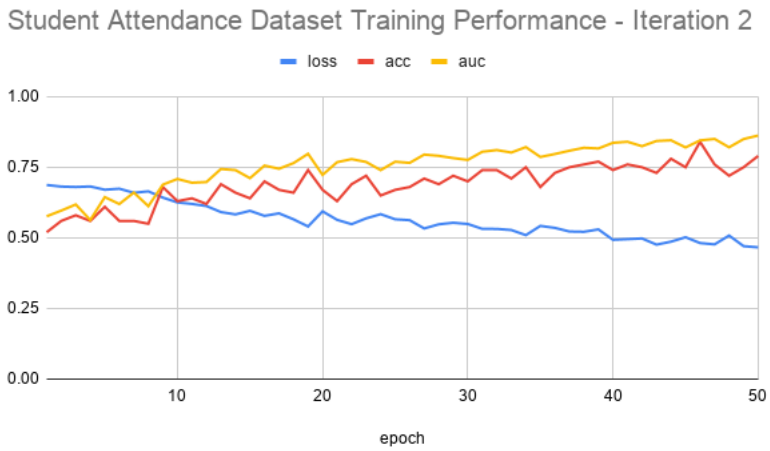


Figure A.10. SEAL – Student Attendance Dataset – Training – Iteration.2

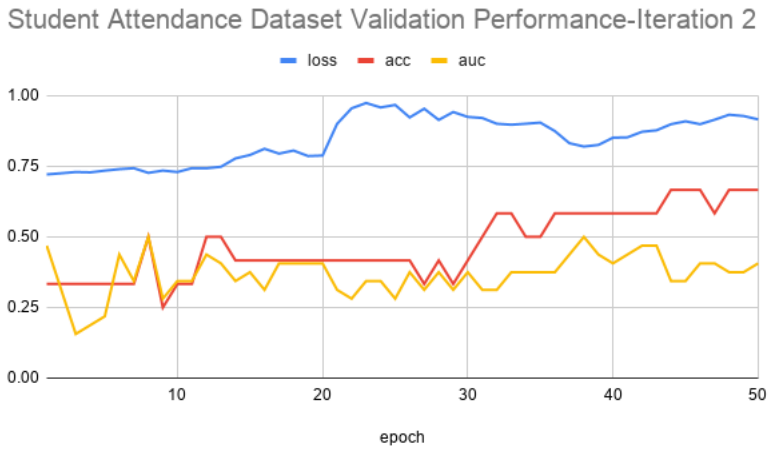


Figure A.11. SEAL – Student Attendance Dataset – Validation – Iteration.2

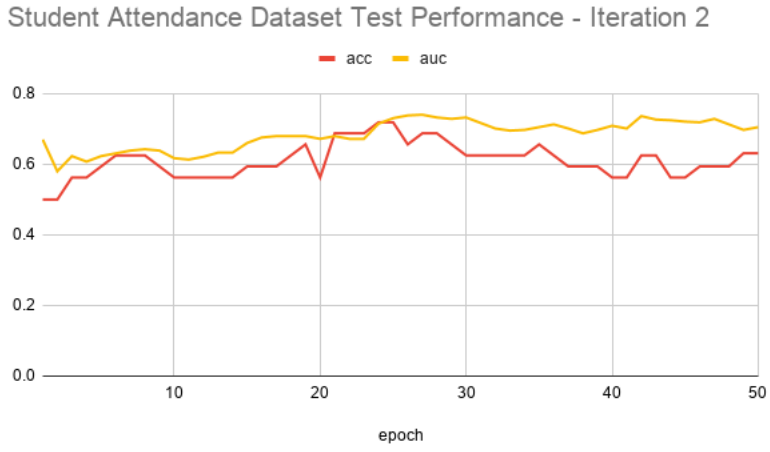


Figure A.12. SEAL – Student Attendance Dataset – Test – Iteration.2

APPENDIX B

DEEPLINKER – ESG DATASET RESULTS

Table B.1. DeepLinker-ESG ISELTA Dataset Iteration Best Results

Iteration	DeepLinker-ESG – ISELTA Dataset								
	Training			Validation			Test		
	loss	acc	auc	loss	acc	auc	loss	acc	auc
1	*0.690*	*0.553*	*NaN*	*0.683*	*0.750*	*NaN*	*0.689*	*0.563*	*0.590*
2	*0.693*	*0.502*	*NaN*	*0.693*	*0.500*	*NaN*	*0.694*	*0.479*	*0.526*
3	*0.692*	*0.542*	*NaN*	*0.683*	*0.625*	*NaN*	*0.691*	*0.604*	*0.642*
4	*5.274*	*0.539*	*NaN*	*0.655*	*0.625*	*NaN*	*0.763*	*0.547*	*0.500*
5	*0.692*	*0.547*	*NaN*	*0.674*	*0.813*	*NaN*	*0.694*	*0.469*	*0.605*
6	*2.767*	*0.561*	*NaN*	*0.699*	*0.500*	*NaN*	*0.899*	*0.547*	*0.548*
7	*0.691*	*0.530*	*NaN*	*0.615*	*0.938*	*NaN*	*0.691*	*0.656*	*0.705*
8	***0.693***	***0.552***	***NaN***	***0.652***	***0.875***	***NaN***	***0.688***	***0.594***	***0.587***
9	*0.693*	*0.547*	*NaN*	*0.691*	*0.563*	*NaN*	*0.683*	*0.688*	*0.653*
10	*0.693*	*0.517*	*NaN*	*0.686*	*0.688*	*NaN*	*0.693*	*0.557*	*0.603*
11	*0.691*	*0.531*	*NaN*	*0.616*	*0.500*	*NaN*	*0.702*	*0.526*	*0.585*
12	*0.612*	*0.642*	*NaN*	*0.577*	*0.750*	*NaN*	*0.626*	*0.708*	*0.684*
13	*0.691*	*0.555*	*NaN*	*0.686*	*0.594*	*NaN*	*0.693*	*0.599*	*0.559*
14	*0.679*	*0.613*	*NaN*	*0.921*	*0.406*	*NaN*	*0.851*	*0.547*	*0.456*
15	*0.692*	*0.538*	*NaN*	*0.689*	*0.625*	*NaN*	*0.690*	*0.682*	*0.646*
16	*0.652*	*0.641*	*NaN*	*0.664*	*0.781*	*NaN*	*0.622*	*0.729*	*0.777*
17	*0.691*	*0.534*	*NaN*	*0.667*	*0.594*	*NaN*	*0.695*	*0.516*	*0.587*
18	*0.691*	*0.553*	*NaN*	*0.678*	*0.594*	*NaN*	*0.692*	*0.578*	*0.627*
19	*0.674*	*0.567*	*NaN*	*0.689*	*0.719*	*NaN*	*0.645*	*0.708*	*0.761*
20	*0.747*	*0.563*	*NaN*	*0.623*	*0.750*	*NaN*	*0.687*	*0.656*	*0.653*
21	*0.519*	*0.000*	*NaN*	*0.531*	*0.000*	*NaN*	*0.479*	*0.535*	*0.519*
22	*0.692*	*0.536*	*NaN*	*0.676*	*0.688*	*NaN*	*0.694*	*0.604*	*0.599*
23	*0.637*	*0.617*	*NaN*	*0.571*	*0.813*	*NaN*	*0.633*	*0.703*	*0.732*
24	*0.707*	*0.545*	*NaN*	*0.654*	*0.625*	*NaN*	*0.695*	*0.578*	*0.665*

Figure B.1. DeepLinker - ISELTA Dataset – Training – Iteration.8

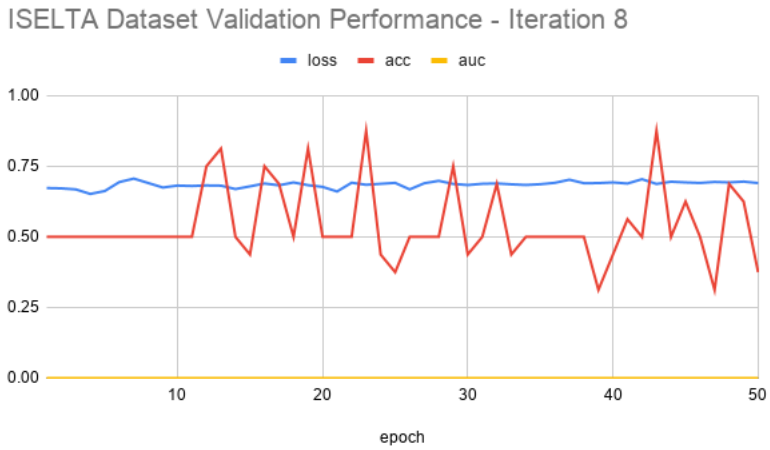


Figure B.2. DeepLinker - ISELTA Dataset – Validation – Iteration.8

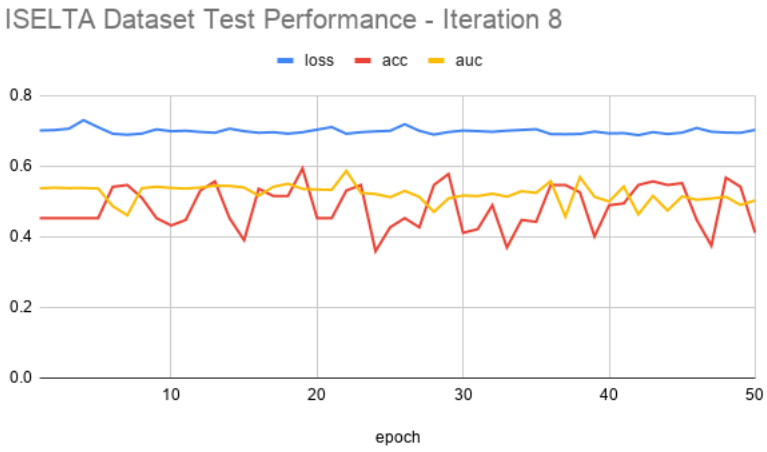


Figure B.3. DeepLinker - ISELTA Dataset – Test – Iteration.8

Table B.2. DeepLinker-ESG Bank Account Dataset Iteration Best Results

Iteration	DeepLinker-ESG – Bank Account Dataset								
	Training			Validation			Test		
	loss	acc	auc	loss	acc	auc	loss	acc	auc
1	*0.668*	*0.607*	*NaN*	*0.671*	*0.667*	*NaN*	*0.689*	*0.688*	*0.672*
2	*0.647*	*0.652*	*NaN*	*0.663*	*0.500*	*NaN*	*0.676*	*0.656*	*0.578*
3	*0.578*	*0.714*	*NaN*	*0.566*	*0.833*	*NaN*	*0.695*	*0.656*	*0.594*
4	*0.678*	*0.616*	*NaN*	*0.684*	*0.667*	*NaN*	*0.688*	*0.781*	*0.766*
5	*0.676*	*0.554*	*NaN*	*0.692*	*0.500*	*NaN*	*0.693*	*0.500*	*0.660*
6	*0.679*	*0.589*	*NaN*	*0.596*	*0.667*	*NaN*	*0.695*	*0.531*	*0.551*
7	*0.667*	*0.616*	*NaN*	*0.686*	*0.667*	*NaN*	*0.693*	*0.781*	*0.738*
8	*0.675*	*0.643*	*NaN*	*0.671*	*0.667*	*NaN*	*0.681*	*0.781*	*0.703*
9	***0.685***	***0.563***	***NaN***	***0.683***	***0.833***	***NaN***	***0.688***	***0.516***	***0.695***
10	*0.666*	*0.589*	*NaN*	*0.684*	*0.833*	*NaN*	*0.685*	*0.594*	*0.641*
11	*0.605*	*0.696*	*NaN*	*0.693*	*0.667*	*NaN*	*0.692*	*0.656*	*0.648*
12	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
13	*0.720*	*0.555*	*NaN*	*0.627*	*0.833*	*NaN*	*0.696*	*0.500*	*0.715*
14	*0.684*	*0.563*	*NaN*	*0.728*	*0.500*	*NaN*	*0.676*	*0.500*	*0.600*
15	*0.956*	*0.719*	*NaN*	*1.030*	*0.667*	*NaN*	*1.606*	*0.641*	*0.637*
16	*2.543*	*0.617*	*NaN*	*0.693*	*0.500*	*NaN*	*0.696*	*0.563*	*0.586*
17	*0.771*	*0.594*	*NaN*	*0.660*	*0.500*	*NaN*	*0.647*	*0.625*	*0.678*

(cont. on next page)

Table B.2. (cont.)									
18	*0.652*	*0.547*	*NaN*	*0.623*	*0.500*	*NaN*	*0.693*	*0.500*	*0.602*
19	*0.647*	*0.633*	*NaN*	*0.627*	*0.833*	*NaN*	*0.721*	*0.656*	*0.570*
20	*0.685*	*0.617*	*NaN*	*0.691*	*0.833*	*NaN*	*0.685*	*0.719*	*0.633*
21	*0.669*	*0.633*	*NaN*	*0.683*	*0.667*	*NaN*	*0.691*	*0.500*	*0.687*
22	*0.679*	*0.602*	*NaN*	*0.670*	*0.833*	*NaN*	*0.692*	*0.594*	*0.531*
23	*0.594*	*0.680*	*NaN*	*0.694*	*0.667*	*NaN*	*0.677*	*0.750*	*0.652*
24	*0.680*	*0.594*	*NaN*	*0.696*	*0.833*	*NaN*	*0.690*	*0.750*	*0.682*

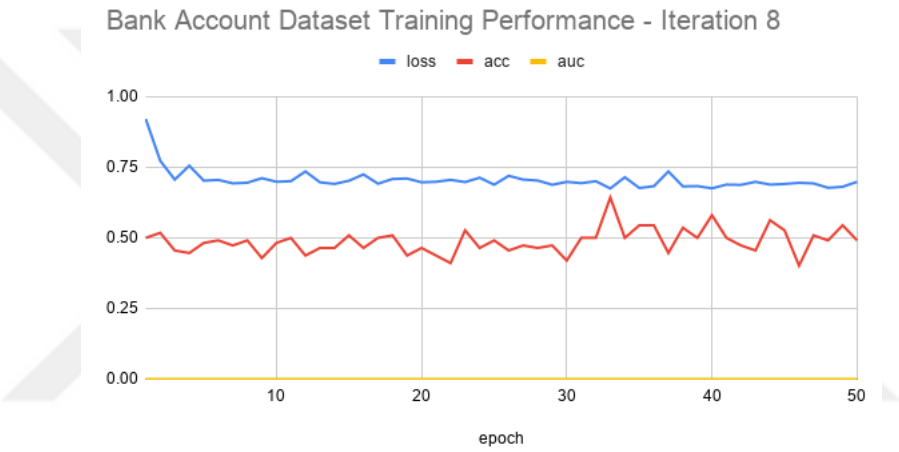


Figure B.4. DeepLinker – Bank Account Dataset – Training – Iteration.8

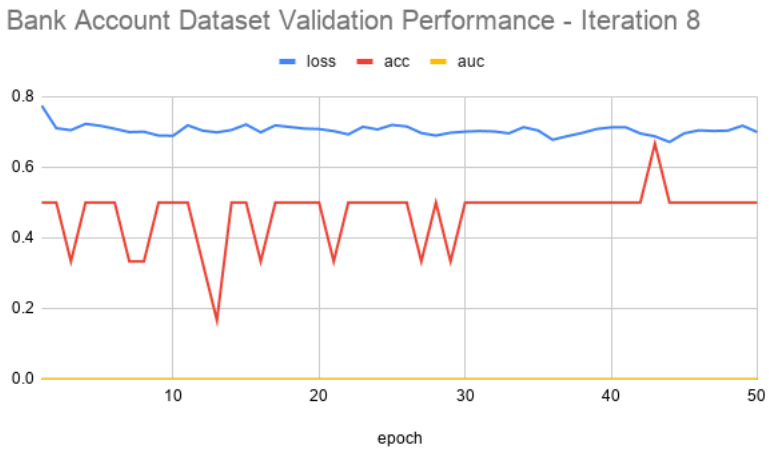


Figure B.5. DeepLinker – Bank Account Dataset – Validation – Iteration.8

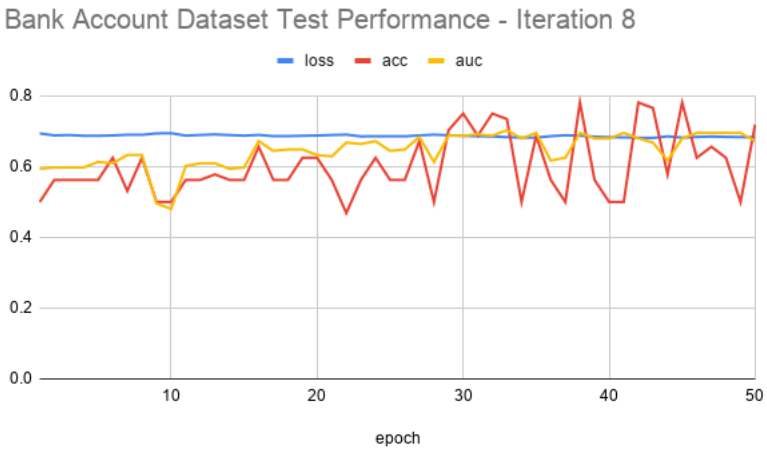


Figure B.6. DeepLinker – Bank Account Dataset – Test – Iteration.8

Table B.3. DeepLinker-ESG Student Attendance Dataset Iteration Best Results

Iteration	DeepLinker-ESG – Student Attendance Dataset								
	Training			Validation			Test		
	loss	acc	auc	loss	acc	auc	loss	acc	auc
1	0.702	0.563	0.000	0.627	0.714	0.000	0.662	0.521	0.685
2	0.685	0.518	0.000	0.650	0.714	0.000	0.676	0.521	0.659
3	0.688	0.585	0.000	0.562	0.714	0.000	0.691	0.635	0.670
4	0.688	0.581	0.000	0.659	0.643	0.000	0.694	0.531	0.711
5	0.883	0.577	0.000	0.590	0.786	0.000	0.686	0.552	0.667
6	0.689	0.540	0.000	0.684	0.571	0.000	0.691	0.552	0.524
7	0.689	0.551	0.000	0.686	0.714	0.000	0.690	0.479	0.703
8	0.684	0.574	0.000	0.646	0.571	0.000	0.683	0.625	0.625
9	0.690	0.526	0.000	0.685	0.571	0.000	0.692	0.563	0.568
10	0.683	0.540	0.000	0.681	0.571	0.000	0.673	0.646	0.708
11	0.686	0.559	0.000	0.706	0.571	0.000	0.688	0.542	0.634
12	0.681	0.559	0.000	0.679	0.714	0.000	0.686	0.583	0.607
13	0.689	0.573	0.000	0.665	0.714	0.000	0.681	0.573	0.688
14	0.693	0.566	0.000	0.666	0.786	0.000	0.684	0.646	0.664
15	0.681	0.566	0.000	0.619	0.571	0.000	0.667	0.656	0.664
16	0.681	0.576	0.000	0.677	0.643	0.000	0.693	0.615	0.582
17	0.691	0.528	0.000	0.672	0.714	0.000	0.685	0.583	0.758

(cont. on next page)

Table B.3. (cont.)									
18	4.665	0.545	0.000	0.653	0.571	0.000	0.775	0.479	0.539
19	0.694	0.573	0.000	0.659	0.643	0.000	0.703	0.594	0.495
20	0.670	0.573	0.000	0.591	0.571	0.000	0.684	0.583	0.640
21	0.685	0.569	0.000	0.691	0.643	0.000	0.662	0.646	0.638
22	0.688	0.535	0.000	0.687	0.643	0.000	0.693	0.531	0.531
23	0.686	0.573	0.000	0.623	0.714	0.000	0.696	0.573	0.533
24	1.443	0.601	0.000	0.698	0.571	0.000	0.690	0.594	0.627

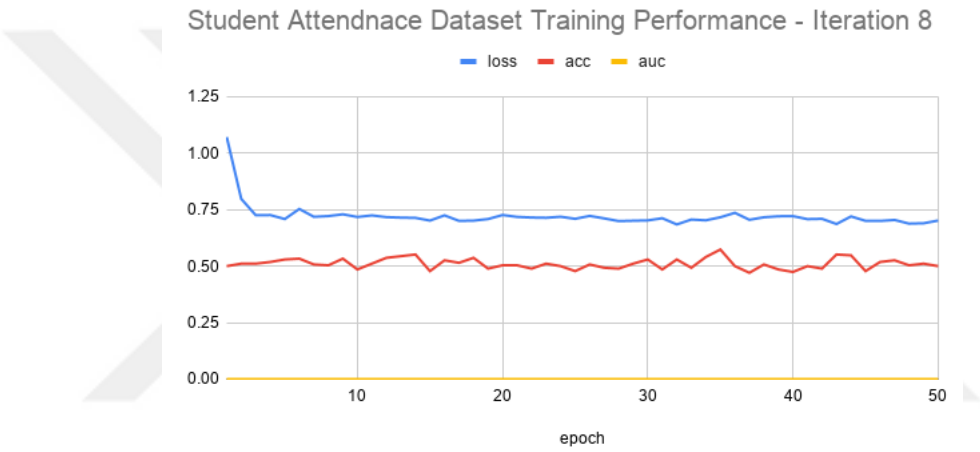


Figure B.7. DeepLinker – Student Attendance Dataset – Training – Iteration.8

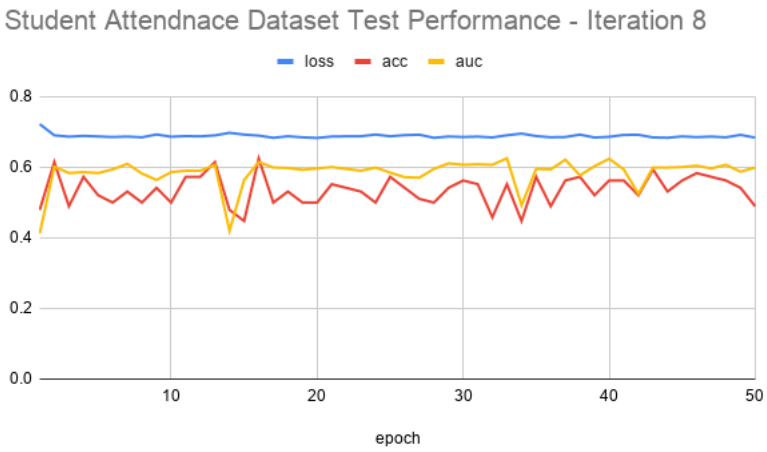


Figure B.8. DeepLinker – Student Attendance Dataset – Validation – Iteration.8

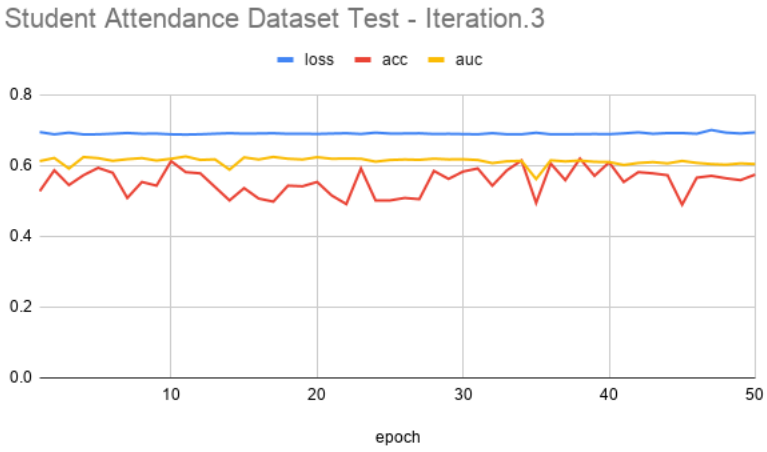


Figure B.9. DeepLinker – Student Attendance Dataset – Test – Iteration.8

Table B.4. DeepLinker-ESG Email Dataset Iteration Best Results

Iteration	DeepLinker-ESG – Email Dataset								
	Training			Validation			Test		
	loss	acc	auc	loss	acc	auc	loss	acc	auc
1	0.662	0.635	0.000	0.671	0.600	0.000	0.682	0.571	0.699
2	0.681	0.521	0.000	0.655	0.600	0.000	0.653	0.500	0.561
3	0.642	0.646	0.000	0.620	1.000	0.000	0.676	0.714	0.643
4	0.645	0.625	0.000	0.535	0.600	0.000	0.673	0.500	0.694
5	0.666	0.615	0.000	0.694	0.600	0.000	0.697	0.464	0.316
6	0.671	0.594	0.000	0.538	0.600	0.000	0.690	0.571	0.717
7	0.687	0.604	0.000	0.656	1.000	0.000	0.681	0.679	0.633
8	0.676	0.594	0.000	0.668	1.000	0.000	0.687	0.607	0.577
9	0.649	0.615	0.000	0.681	0.600	0.000	0.683	0.643	0.781
10	0.667	0.583	0.000	0.525	0.800	0.000	0.669	0.500	0.531
11	0.654	0.604	0.000	0.672	0.600	0.000	0.681	0.679	0.704
12	0.673	0.594	0.000	0.610	0.800	0.000	0.705	0.643	0.546
13	0.683	0.625	0.000	0.692	0.600	0.000	0.688	0.571	0.500
14	6.214	0.604	0.000	1.663	0.600	0.000	1.803	0.500	0.500
15	0.678	0.573	0.000	0.644	0.800	0.000	0.674	0.607	0.668
16	0.677	0.573	0.000	0.596	0.800	0.000	0.629	0.821	0.883

(cont. on next page)

Table B.4. (cont.)									
17	0.687	0.552	0.000	0.687	0.600	0.000	0.692	0.571	0.592
18	0.673	0.552	0.000	0.676	0.600	0.000	0.693	0.571	0.541
19	0.673	0.583	0.000	0.669	0.800	0.000	0.693	0.714	0.612
20	0.673	0.604	0.000	0.678	1.000	0.000	0.691	0.750	0.684
21	0.690	0.594	0.000	0.692	0.800	0.000	0.680	0.607	0.546
22	0.663	0.594	0.000	0.605	0.600	0.000	0.683	0.607	0.556
23	0.674	0.567	0.000	0.689	0.719	0.000	0.645	0.708	0.761
24	0.747	0.563	0.000	0.623	0.750	0.000	0.687	0.656	0.653

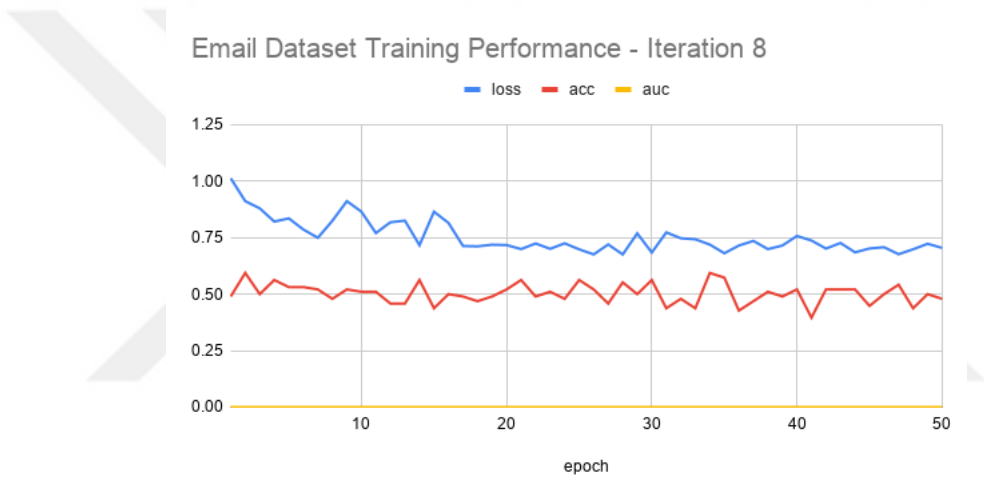


Figure B.10. DeepLinker - Email Dataset – Training – Iteration.8

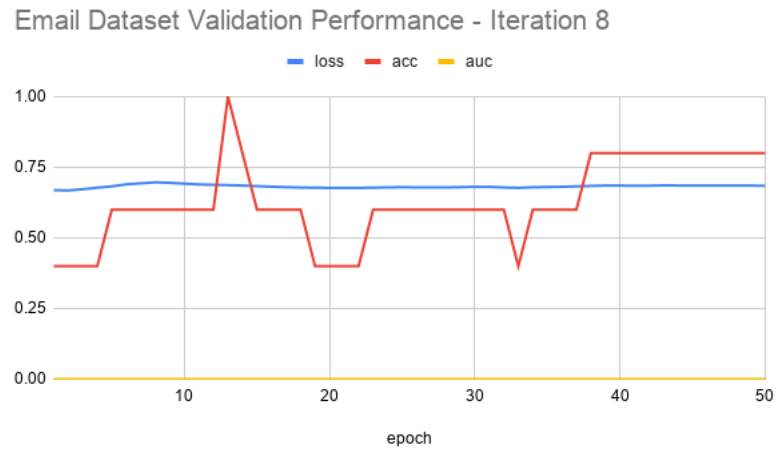


Figure B.11. DeepLinker - Email Dataset – Validation – Iteration.8

Figure B.12. DeepLinker - Email Dataset – Test – Iteration.8