San Jose State University

SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

Spring 2023

Malware Classification using Graph Neural Networks

Manasa Mananjaya
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation
Mananjaya, Manasa, "Malware Classification using Graph Neural Networks" (2023). *Master's Projects*. 1268.
DOI: https://doi.org/10.31979/etd.68ya-hj74
https://scholarworks.sjsu.edu/etd_projects/1268

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Malware Classification using Graph Neural Networks

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Manasa Mananjaya

May 2023

© 2023

Manasa Mananjaya

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Malware Classification using Graph Neural Networks

by

Manasa Mananjaya

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

Dr. Fabio Di Troia	Department of Computer Science
Dr. William Andreopoulos	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science

ABSTRACT

Malware Classification using Graph Neural Networks

by Manasa Mananjaya

Word embeddings are widely recognized as important in natural language processing for capturing semantic relationships between words. In this study, we conduct experiments to explore the effectiveness of word embedding techniques in classifying malware. Specifically, we evaluate the performance of Graph Neural Network (GNN) applied to knowledge graphs constructed from opcode sequences of malware files. In the first set of experiments, Graph Convolution Network (GCN) is applied to knowledge graphs built with different word embedding techniques such as Bag-of-words, TF-IDF, and Word2Vec. Our results indicate that Word2Vec produces the most effective word embeddings, serving as a baseline for comparison with three GNN models- Graph Convolution network, Graph Attention network (GAT), and GraphSAGE network (GraphSAGE). For the next set of experiments, we generate vector embeddings of various lengths using Word2Vec and construct knowledge graphs with these embeddings as node features. Through performance comparison of the GNN models, we show that larger vector embeddings improve the models' performance in classifying the malware files into their respective families. Our experiments demonstrate that word embedding techniques can enhance feature engineering in malware analysis.

ACKNOWLEDGMENTS

I express my appreciation to Dr. Fabio Di Troia, my advisor, for his encouragement and mentorship during my research. His vast knowledge and valuable suggestions have been essential in directing my research toward the right path and ensuring its progress. I am thankful to my committee members, Dr. William Andreopoulos and Dr. Thomas Austin, for their valuable time and constructive comments.

Additionally, I would like to acknowledge the continuous support of my family and friends throughout this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The threat of malware to the security of computer systems and networks is growing rapidly. Malicious software can be used to steal sensitive data, gain unauthorized access, disrupt services, and cause damage to systems. Due to their increasing complexity and sophistication, malware is becoming difficult to detect and analyze. Traditional signature-based methods are often insufficient to detect new and unknown malware, while behavioral-based methods may produce high false positives. Machine learning techniques have emerged as a promising approach to malware classification, offering the potential for accurate and efficient detection of both known and unknown malware.

One such machine learning technique called Graph Neural Network (GNN) has recently gained attention in the field of malware analysis as a powerful tool for capturing the structural relationships between features of malware samples [1]. GNNs can process data represented as graphs, where individual elements are depicted as nodes and edges depict relationships between the elements [2]. They have proven effective in different fields of study, such as Natural Language Processing (NLP), Computer Vision, and Social Network Analysis, where the data is represented as a graph.

On the other hand, word embedding techniques are implemented to represent words in a high-dimensional space as vectors. These vectors capture the meaning and structural relationships between words and can be used for various NLP tasks. By applying word embedding techniques to malware samples, it is possible to capture the semantic relationships between different parts of the code and use them for classifying malware [3].

In this research, we explore the use of three GNN models for malware classification using word embedding techniques. We focus on the application of GNNs to knowledge graphs constructed from opcode sequences of malware files. Opcode sequences are a representation of the behavior of a program, consisting of instructions executed by the program. Knowledge graphs are graphs that represent structured knowledge in a form that can be processed by machines. In our case, we construct knowledge graphs where nodes represent opcodes, and edges represent the co-occurrence of opcodes in malware samples.

We evaluate the effectiveness of different word embedding methods such as Word2Vec [4], TF-IDF [5], and Bag-of-Words [6] in classifying malware using GNNs. First, we investigate the performance of GCN applied to knowledge graphs built using various word embedding techniques. The best word embedding technique is then deduced and used to build knowledge graphs. Finally, the performances of GCN, GAT, and GraphSAGE in classifying the word-embedded knowledge graphs into their respective families are evaluated.

The remaining sections of this paper are organized in the following manner. In Chapter 2, we discuss previous research conducted in the area of machine learning for malware analysis, including word embedding techniques and GNNs. Chapter 3 describes the data set used in our experiments, the methodology for constructing knowledge graphs, and the results of our classification experiments. Finally, we conclude our report and present potential directions for future work in Chapter 4.

CHAPTER 2

Background

Malware is a software program designed with the malicious intent of causing harm to computer systems, stealing sensitive information, or gaining unauthorized access to the network. They are difficult to detect and analyze due to their increasing complexity and sophistication, posing a significant threat to the security of computer systems and networks [7]. According to the Cybersecurity Ventures Report 2021, cybercrime will cause damages worth $10.5 trillion annually by 2025 [8]. Malware is a primary weapon used by cybercriminals to infiltrate computer systems and networks. The sheer volume and diversity of malware pose a significant challenge to cybersecurity experts. For instance, in 2019, the number of malware families increased by 13.7%, reaching a total of 971,390 [9]. Moreover, cybercriminals use various obfuscation techniques to make it harder for traditional signature-based methods to detect malware. In addition, the detection speed of new malware variants is crucial, as delays can lead to serious security breaches. Therefore, the need for effective and efficient malware detection techniques is essential to protect computer systems and networks from malware attacks.

Most companies rely on conventional methods for detecting malware, such as signature-based and behavioral-based detection methods. Signature-based malware detection methods make use of signatures or patterns to identify known malware. These methods are based on the idea that malware has distinguishing characteristics that can be utilized to identify it [10]. They compare the code or behavior of a file to a database of known signatures to detect malware. Although this technique is fast and efficient, it is ineffective in detecting unknown malware.

On the other hand, malware detection methods based on behavior focus on the actions performed by the malware rather than just its signature [11]. These methods monitor the system's activities and detect unusual behavior that may indicate malware activity. They overcome the drawback of traditional based methods in detecting new and unknown malware that do not have signatures in the database. However, behavioral-based methods can produce high false positives since legitimate software can also exhibit abnormal behavior.

As traditional signature-based and behavioral-based methods of malware detection are insufficient to detect new and unknown malware, researchers have turned to machine learning for improved malware detection.

2.1 Previous Work

In recent years, research has been focused on developing advanced methods to detect malware using machine learning techniques. Among these techniques, graph neural networks (GNNs) and word embedding have gained considerable attention for their effectiveness in identifying and classifying malware. In [12], the authors conducted a survey on malware detection using graph representation learning. They analyzed various graph-based methods for detecting malware and discussed the advantages and limitations of these methods. The survey concluded that GNNs show potential as a viable method for malware detection due to their ability to capture the complex relationships between different features of malware.

A Dynamic Evolving Graph Convolutional Network (DEGCN) was proposed in [13] to detect malware. The malware files are represented as graphs where the API calls are represented as nodes and the order in which they appear is captured through the edges between the nodes. The DEGCN model dynamically adjusts the node weights based on the significance of the API call and updates the edge weights according to their temporal sequence. The proposed model achieved a 98.3% detection rate on a dataset of 1,400 malware samples.

The proposed malware detection method in [14] implements GCNs for identifying malware. The authors used a graph representation of malware, where nodes represent API calls and edges represent their dependencies. This method achieved an accuracy of 98.6% on a dataset of 3,512 malware samples. In a similar context, [15] implemented GATs for intelligent transportation systems. The authors used a graph-based representation of network traffic, where nodes represented the source and destination IP addresses, and edges represented the communication between these addresses. An accuracy of 97.4% was achieved by the proposed method on a dataset of 400,000 network packets.

The authors of [16] proposed a multi-view attention-based deep learning framework for detecting malware in smart healthcare systems. They used multiple views of malware, including API calls, system calls, and static features, and applied attention mechanisms to capture the most relevant features. The attention mechanism focuses on the most significant segments of the input data, thereby reducing the feature space's dimensionality and enhancing the classification accuracy. According to the results, the proposed framework performs better with an accuracy of 99.4% as compared to Support Vector Machine (SVM) and Convolution Neural Networks (CNNs).

In [17], an intelligent malware detection method based on GCN is proposed. Over 15,000 malware samples are represented as graphs and evaluated using GCN. The performance of the proposed method is evaluated against traditional machine learning methods such as SVM, and other graph-based methods such as GAT and ChebNet. This method shows promise in improving the detection of malware through the use of GCNs. Comparably, the authors of [18] proposed a GNN model that uses a similarity-based approach to cluster malware samples with similar structures into the same category, regardless of their behavior. One of the strengths of the proposed method is that it does not need any feature engineering or prior knowledge about malware. The approach extracts the structural information of malware samples automatically and captures the underlying similarities between them. This makes the approach more robust and generalizable to the new malware samples.

Rather than traditional graph-based methods, [19] implements a novel attention network that uses multi-feature alignment and fusion to detect malware. The proposed model combines the strength of GCNs and attention mechanisms to capture both local and global features of malware effectively. To evaluate the proposed model, the authors conducted experiments on a dataset of 10,000 benign samples and 10,000 malware samples and achieved an accuracy of 99.2% and an AUC of 0.998. The multi-feature alignment and fusion technique improves the alignment of multiple features and enhances the model's performance by providing high accuracy and robustness.

Various word embedding techniques are explored in [20]. The authors propose a novel method for representing malware samples as sequences of opcodes, which are then converted into Word2Vec embeddings or HMM states. The experiments were conducted on a dataset of 7,000 malware samples from 7 families. The experiments assessed how effectively 8 distinct machine learning techniques classified malware. The results showed that the machine learning methods based on Word2Vec embeddings outperformed those based on HMM states. Specifically, the Word2Vec-RF model achieved the highest accuracy of 96.2%, while the HMM-RF model achieved an accuracy of 96%. This shows that Word2Vec-based models outperform HMM-based models in terms of accuracy and computational efficiency.

In a similar context, [21] compares the performance of three machine learning techniques, Word2Vec, PCA2Vec, and HMM2Vec, for classifying malware. The results show that Word2Vec-based techniques perform the best and are more computationally efficient than HMM2Vec and PCA2Vec. The paper provides a useful comparison of different machine learning techniques with word embedding for malware classification, which can help researchers and practitioners in the field make informed decisions about which techniques to use for their specific applications.

During our review of previous research, we understood that utilizing word embedding techniques for feature engineering can enhance a model's effectiveness. By integrating this with GNNs, we can create a robust model capable of detecting malware. This is the foundation of our study.

2.2 Word Embedding Techniques

Word embedding techniques are used in NLP to represent words as high-dimensional vectors of numerical values. These techniques map words with comparable meanings to comparable vectors in a high-dimensional space, allowing for mathematical operations to be performed on words. This enables machine learning algorithms to process text data in a more efficient and effective manner, improving the performance of tasks such as text classification. In this section, we explore three different techniques of word embedding and evaluate their effectiveness in classifying malware using GNNs. These techniques are utilized to generate feature vectors that are subsequently utilized as node features within knowledge graphs.

2.2.1 Bag-of-Words

Bag-of-words (BoW) [6] is a simple and widely used word embedding technique in NLP. This method counts the occurrences of each word in a text document to represent it as a vector. BoW considers each word in the document to be independent of the other words in the text and disregards their order, syntax, or structure.

It is necessary to create a vocabulary comprising tokenized text data to generate a BoW model. We build a matrix of word frequency counts for each document in the corpus using this vocabulary, where each document is represented by a row and each word in the vocabulary is represented by a column. In order to take into consideration the different document lengths and phrase frequencies, we finally normalize the matrix.

Let W stand for the vocabulary set, D for the collection of documents, and *n(d, w)* for the frequency of word w in a document d. The BoW representation of document d is a vector *x(d)* of size |W|, where each element of the vector is given by:

$$x(d)[w] = n(d, w)$$

In the experiments described in Chapter 3, we use the BoW model to generate one-dimensional feature vectors. These vectors will serve as node features during the classification of graphs. More information on BoW can be found in [22] and [23].

2.2.2 TF-IDF

A prominent word embedding method in NLP, TF-IDF (Term Frequency-Inverse Document Frequency) [5], converts textual information into a numerical representation. It is a statistical measurement that computes each word's importance in a manuscript. The basic principle of the TF-IDF technique is that a word's relevance in a document inversely correlates with its frequency (TF) in that document and across all documents (IDF). In other words, a word is likely to be more significant for that particular document if it occurs frequently there but infrequently elsewhere.

TF-IDF score is computed by multiplying TF and IDF of a word w in a document d:

$$TF - IDF(w, d) = IDF(w) * TF(w, d)$$

TF is the ratio of the number of occurrences of a word w in a document d to the total number of words in d:

$$TF(w, d) = \frac{\text{number of occurrence of w in d}}{\text{total number of words in d}}$$

The logarithm of the ratio of the total number of documents N to the number of documents containing the word w gives the IDF value:

$$IDF(w) = \log\left(\frac{N}{\text{number of documents that include } w}\right)$$

The logarithmic function is used to reduce the impact of rare words on the IDF score. The IDF score of words that frequently appear in documents will be lower, and as a result, will have a lower impact on the TF-IDF score. The TF-IDF scores for each word in a document may then be computed and used as features in machine learning models. More information on TF-IDF can be found in [24].

2.2.3 Word2Vec

Word2Vec is a shallow neural network that is used to generate word embeddings [25]. Word embeddings are word representations that are distributed across a high-dimensional vector space, with each dimension representing a feature of the word. These embeddings can be used in a variety of NLP tasks such as text categorization, sentiment analysis, and language modeling [26].

Word2Vec creates word embeddings using a neural network that has been trained on a large text corpus. This neural network learns to either predict a word given its context or to anticipate a word given its nearby words. The word embeddings are subsequently created using the weights from the neural network's hidden layer. Word2Vec's ability to capture semantic relationships between words is its biggest advantage. For example, words with similar meanings, such as "car" and "automobile" will have similar embeddings.

Two architectures used to train Word2Vec are the Continuous Bag of Words (CBOW) and the Skip-Gram model [27]. The CBOW technique utilizes a group of words surrounding the target word to make a prediction, whereas the Skip-Gram approach takes the target word as input and tries to anticipate the surrounding context words [28]. We experiment with the Skip-Gram model in this research.

2.3 Graph Neural Networks

Graph Neural Network is a deep learning algorithm that is designed to analyze structured data represented as graphs. Unlike traditional neural networks that take fixed-length data as input, GNNs take graphs as input where individual elements are represented as nodes and edges represent the relationship between the elements [29].

Mathematically, GNNs are defined as a series of iterative graph convolution operations, which can be represented as

$$h_v^{(k+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} W^{(k)} h_u^{(k)} + b^{(k)} \right)$$

where $h_v^{(k)}$ is the representation of node v at the k-th iteration, $\mathcal{N}(v)$ is the set of neighboring nodes of v, $W^{(k)}$ and $b^{(k)}$ are the learnable weight matrix and bias vector at the k-th iteration, and σ is a non-linear activation function such as ReLU or sigmoid [30].

In our case of malware classification, the nodes in the input graphs represent the opcodes whereas the edges connect the opcodes that appear together frequently. The GNN performs message passing between the nodes to capture information about the relationships between them. This involves computing node embeddings based on the embeddings of its neighbors, and then using these embeddings to update the central node's representation. This process can be repeated multiple times to capture higher-level relationships between nodes. This paper presents the implementation of three graph neural network models to classify malware files.

2.3.1 Graph Convolution Network

Graph Convolutional Network [31], a variant of GNN, incorporates convolutional layers, which allow for shared weights and translation invariance, as well as pooling layers, which allow for hierarchical learning. GCNs learn a set of filters that can operate on the graph structure to extract features from the data. The filters are typically defined as functions that operate on the node's local neighborhood and produce a new representation for the node.

During training, the weights of the filters are learned through backpropagation, which enables the GCN to learn to extract meaningful features from the graph structure. By applying these filters repeatedly, the GCN is able to learn hierarchical representations of the graph. In the classification phase, the feature vectors of each node are taken as input by the GCN and a label for each graph is produced. Labeling the graph is accomplished by applying a pooling operation to the output of the last layer of the GCN. The pooling layer aggregates the feature vectors of all nodes into a single vector. This vector is then fed into a fully connected layer, which produces a final output vector that represents the predicted class probabilities for the input graph. Chapter 3 provides the architecture of GCN and presents several experiments that are conducted using various word embedding techniques.

2.3.2 Graph Attention Network

Graph Attention Networks are a popular graph-based machine learning approach designed by [32]. GATs differ from GCNs by employing an attention mechanism to learn the importance of each node's neighbors for a given task. This is achieved through a series of weighted linear combinations of the neighbors' hidden states, with the weights learned through a self-attention mechanism. In other words, GATs use the graph structure to determine which nodes are most relevant for a given task, rather than treating all nodes equally.

GAT optimizes the loss function with respect to model parameters during the training phase. This typically involves computing the model's predictions for a set of labeled examples and comparing them to the true labels using a loss function. Backpropagation is then used to update the model parameters. Classifying new examples involves computing the hidden states of all nodes in the graph and obtaining a probability distribution over the possible labels. The label with the highest probability is then assigned to the graph. More information on the application of GATs in text classification can be found at [33] and [34].

2.3.3 GraphSAGE Network

GraphSAGE (Graph Sample and Aggregate) networks [35] are a class of GNNs that learn representations for nodes in a graph by aggregating information from their local neighborhoods. This model aims to overcome the drawbacks of traditional graph-based learning methods by leveraging graph convolutions, which can learn from both local and global information.

The GraphSAGE algorithm converts every node in the input graph to a low-dimension vector. A multi-layer neural network operates on each node and its neighbors in the graph to achieve this. At each layer, the model aggregates information from the local neighborhood of each node by sampling a fixed number of neighbors and performing a mean or max pooling operation. The resulting representations are then passed through a non-linear activation function and fed into the next layer. This process is carried out repeatedly for a specified number of layers until the final node embeddings are obtained. This ability of the GraphSAGE network makes it highly efficient in generating node embedding for unseen data.

In our study, GraphSAGE learns embeddings for each opcode by taking into account the relationships between adjacent opcodes. A summary vector is computed for the entire graph based on the embeddings of its constituent nodes. This vector is then passed through a fully-connected neural network to obtain the final graph-level classification [36]. [37] provides more details on how GraphSAGE networks can be implemented for text classification.

CHAPTER 3

Experiments and Result Analysis

This chapter focuses on the malware data utilized in the study and its preprocessing. We provide a brief overview of feature engineering and highlight the experiments performed using different word embedding techniques and GNNs.

3.1 Dataset

The dataset experimented with in this study is taken from the VirusShare website that hosts malware files belonging to various families. It consists of 13,597 malware families with at least one malware file belonging to each family. However, due to the significant number of families and the large number of opcodes in each file, classifying all families requires extensive computational resources, rendering it infeasible. Therefore, we limited our experiments to only five families listed in Table 1. To ensure balance in the dataset, 1,000 samples are selected randomly from these five malware families, resulting in a total of 5,000 samples. Other implementations using this dataset can be found in [20] and [21]. We will briefly discuss the characteristics of each malware family in this section.

Table 1: Malware Families

Family	Type of Malware	No. of Samples
BHO	Trojan	3843
OnLineGames	Password Stealer	13164
Renos	Trojan Downloader	23980
VBInject	VirTool	15171
Winwebsec	Rogue	13277

BHO - This is a type of Trojan malware that is used by attackers for malicious activities like tracking user activities or installing other malware in user systems [38].

OnLineGames - This type of malware is used to target online gamers. It is typically disguised as a component in legitimate game installations or distributed through fake games. Once installed, it can steal sensitive information such as login credentials, banking information, and game items [39].

Renos - Renos is another type of Trojan malware that is typically installed by itself on a computer through security vulnerabilities or social engineering tactics. It performs various malicious activities such as displaying fake alerts or redirecting web traffic [40].

VBInject - Malware of this family injects malicious code into legitimate processes running on the operating systems. Attackers use this malware to steal sensitive information or log keystrokes [41].

Winwebsec - Windows Web Security or Winwebsec is a rogue antivirus software that masquerades as a legitimate antivirus program. It steals personal information and tricks users into paying for unnecessary antivirus licenses [42].

3.2 Dataset Preprocessing

For our research, we classify the executable malware files into their respective families using a `.csv` file containing the file names and families as reference. Specifically, we sort 1,000 original malware files into each of the five families, resulting in a total of 5,000 files. The malware files are disassembled into `.asm` binary files for opcode extraction. This was achieved on a Linux system using the Objdump command, which is part of the GNU Binutils package. Opcodes extracted from each binary file are stored in a text file with the same file name. Including all of the distinct opcodes present in each file would have resulted in additional overhead during the machine learning model training process, as there are a considerable number of such opcodes. Moreover, the majority of opcodes contributed to less than 1% of the total number of opcodes. The top 50 opcodes and their frequencies are shown in Figure 1.

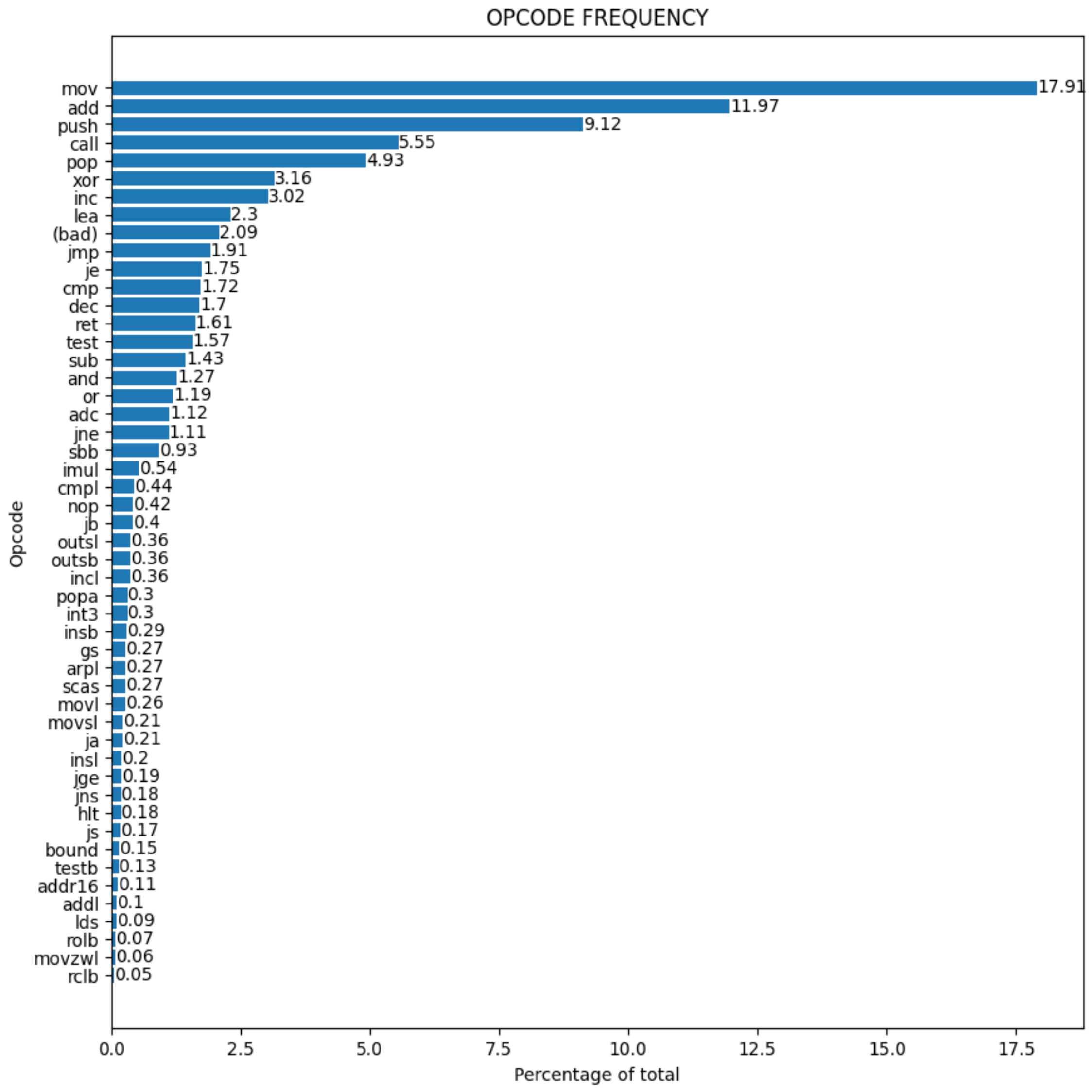


Figure 1: Opcode Frequency

3.2.1 Binary Classification using Word2Vec-CNN

It can be observed that a large portion of the opcodes in the top 50 opcodes are infrequent. To determine how many opcodes to utilize in our experiments, we conducted binary classification using BHO and OnLineGames malware families. 1,000 malware files from each family are considered for classification. Our methodology involved implementing a Convolutional Neural Network (CNN) with the Word2Vec embedding technique. The experiments consisted of training the CNN model with the top 10, 20, 30, 40, and 50 opcodes, with embedded vector lengths of 2, 10, 50, and 100. Figure 2 displays the classification accuracy results obtained by the Word2Vec-CNN model for varying numbers of opcodes and vector lengths. It can be observed that there is no significant difference in performance when using 50 opcodes or 10 opcodes. Therefore, it is more practical to use fewer opcodes to reduce computation time and overhead during the training of machine learning models.

It can be observed that the highest average accuracy is achieved when utilizing the top 20 opcodes across all vector lengths. Consequently, we extracted the top 20 opcodes and processed the opcode files to contain only these opcodes, which are subsequently utilized for further experimentation. For each file, a pre-processing step was performed to remove any punctuations and tokenize the text into individual words.

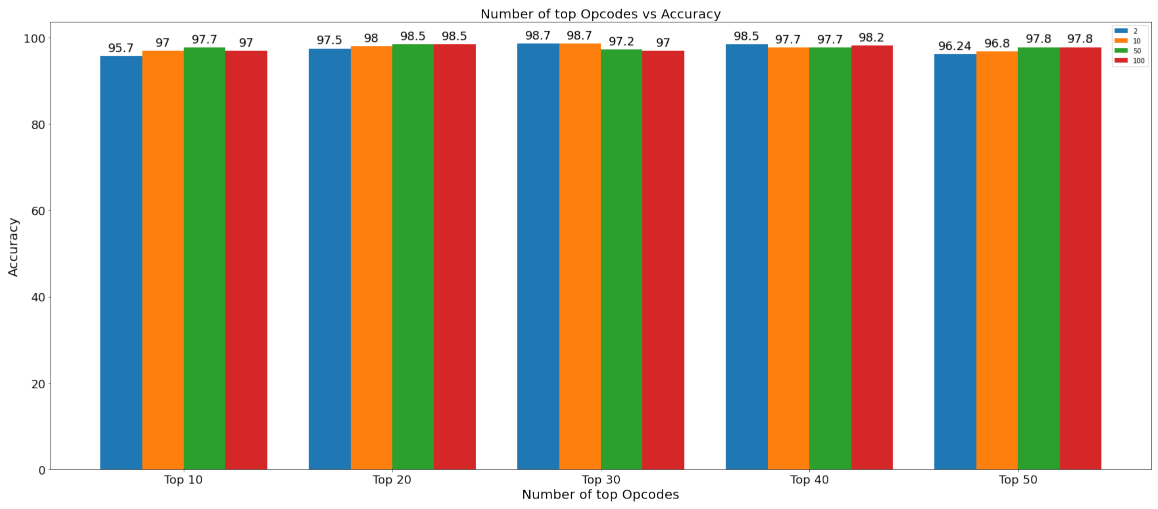


Figure 2: Word2Vec-CNN: Binary Classification Results

3.3 Feature Vector Generation

In this section, we describe different word embedding methods that are implemented to create feature vectors for the sequence of opcodes obtained from the malware files. We also examine the impact of each embedding technique on the performance of the malware file classification process. All of the embedding techniques are tested using the same GCN model described in Section 3.4.1. Additionally, a separate model was developed that did not implement any of the word embedding techniques, which served as a base model for comparison with other word embedding implemented models. The resulting feature vectors are subsequently incorporated into the knowledge graphs as discussed in Section 3.3.4.

3.3.1 GCN-Bag-of-Words

Bag-of-Words is a widely used word embedding technique in NLP [7]. It is a simple yet effective way to extract features and create a feature vector without considering the meaning or semantics of the opcode sequences. It enables us to determine how the frequency of certain opcodes influences the type of malware family we are dealing with. In our implementation of BoW, we create a feature vector of length 20, where each vector value represents the frequency of a particular opcode in the malware file.

To ensure consistent vector lengths, a value of zero is appended to the feature vector if an opcode is missing. This step is necessary because the BoW technique used to generate feature vectors requires a fixed-length vector for each malware file. By appending zeros to the end of the vector, the missing opcodes are effectively represented as non-existent features, which allows for consistent vector lengths to be maintained across all malware files. This feature vector is incorporated in the knowledge graphs as described in Section 3.3.4. Figure 3 depicts how the feature vector is generated using the BoW method.

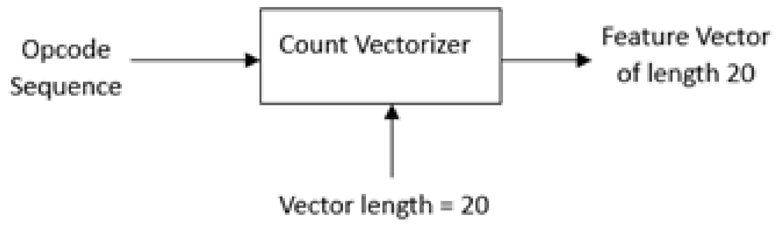


Figure 3: Bag-of-Words Feature Generation

3.3.2 GCN-TF-IDF

This section presents the TF-IDF technique for generating feature vectors for each malware file containing the opcode sequences. In our implementation, we first create a document-term matrix, where each row represents a document (i.e., malware file) and each column represents an opcode. The entries in the matrix correspond to the term frequency of each opcode in each document. Next, we compute the inverse document frequency for each opcode as described in Section 2.2.2. Finally, we calculate the TF-IDF score for each opcode by multiplying its TF score with the IDF value. The resulting TF-IDF matrix is then used to generate a feature vector for each file, where each vector represents a document and contains the TF-IDF scores for every opcode in the malware file. Figure 4 illustrates the process of generating feature vectors using the TF-IDF vectorizer.

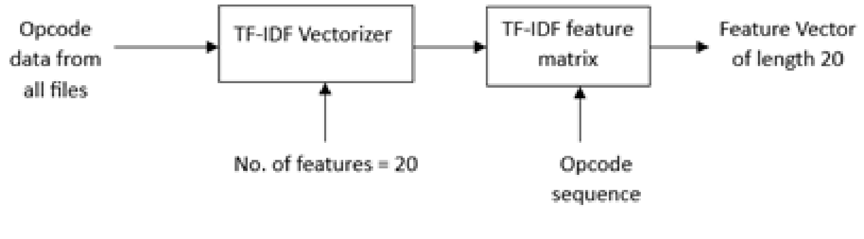


Figure 4: TF-IDF Feature Generation

3.3.3 GCN-Word2Vec

Word2Vec is an important word embedding method used to generate feature vectors in this research [7]. In our experiment, we implement the `gensim` Word2Vec model and train it using the opcode sequences. The vector length is kept as 100 and the window size is set to the default value of 5. The Word2Vec library offers two training techniques - Skip-Gram and Continuous Bag of Words (CBOW). To conduct our experiments, we opted to use the CBOW algorithm to train Word2Vec. The trained Word2Vec model is then employed to generate feature vectors for each document by averaging the opcode vectors of all the opcodes present in the document. Figure 5 illustrates the usage of the Word2Vec model in generating feature vectors for each malware file.

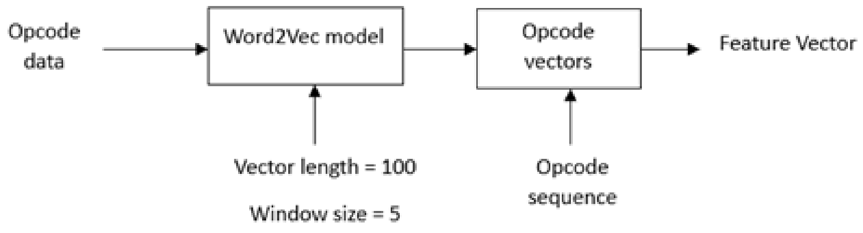


Figure 5: Word2Vec Feature Generation

3.3.4 Creating Knowledge Graphs

We utilize Python's `NetworkX` library to create knowledge graphs from the opcode files. The nodes in the graph represent the opcodes, and the edges between them indicate the relationship between the opcodes. Specifically, we create an edge between two opcodes if they occur consecutively in the file. To compute the edge weights, we calculate the bi-gram frequency of the opcode pair.

To establish a baseline for comparison, we generate 5,000 knowledge graphs from the opcode files without any word embedding. This baseline model is used to compare with the word-embedded graphs. After creating the graph, we save it in a `.pkl` file format, along with its corresponding label. During classification, this data is retrieved from the `.pkl` file and used for further analysis. An example of a regular graph is shown in Figure A.15.

To generate word-embedded graphs, we store the feature vectors generated by the word embedding techniques in the `.pkl` files along with the graph and label data. During the training phase, these feature vectors are embedded as node features in the graph that is loaded. This additional information enhances the performance of the GNN models. Figures A.16, A.17, and A.18 illustrate the knowledge graphs created using BoW, TF-IDF, and Word2Vec techniques respectively. Each of these techniques generates 5,000 knowledge graphs, which are used in our experiments.

3.3.5 Results for Word Embedding Experiments

The classification results obtained for various word embedding techniques are summarized in Table 2. The same GCN model described in Section 3.4.1 is used to classify the graphs generated using these word embedding techniques. These results help us in deciding which word embedding technique to continue with for our implementation using GNN models. We can see that only Word2Vec is giving improved classification results as compared to our baseline model. On the other hand, TF-IDF and BoW worsen the classification accuracy with BoW giving 59.80% accuracy and TF-IDF giving just 22.6% accuracy.

It should be observed that the dimension of feature vectors generated by BoW and TF-IDF techniques are 20x1, whereas the feature vectors generated by Word2Vec are of size 20x100. This indicates the dimensionality of the vector space that each technique uses to represent the text data. A larger vector size allows for more complex and nuanced relationships between opcodes to be captured, which can improve the performance in downstream tasks such as classification. Therefore, we select Word2Vec as the primary word embedding technique for our GNN models. Section 3.4 describes this in detail.

Table 2: Accuracy of Word Embedding Techniques

Model	Accuracy
Baseline Model	71.60%
GCN-BoW	59.80%
GCN-TFIDF	22.60%
GCN-Word2Vec	91.91%

3.4 GNN Implementation

This section presents the architecture of the three GNNs used in this research for malware classification. All models are designed to handle graph data with Word2Vec generated feature vectors that are embedded as node features in the graphs. In our experiments, we vary the vector length of feature vectors generated by Word2Vec. More information on these experiments is available in Section 3.5.

In our implementation, the first step involves loading all the graph data from the stored `.pkl` files. The feature vectors are then read from the files and embedded in the nodes after the graph is loaded. Before training the GNN models, we use graph generators to create graph data generators, which are used to feed the graph data to the GNN models during the training phase. This enables the models to learn from the graph data with varying numbers of nodes and edges.

3.4.1 Word2Vec-GCN

Graph Convolutional Network is a neural network designed to work with graph-structured data [43], and it performs message passing over the graph to compute node embeddings. `GCNSupervisedGraphClassification` class from the `stellargraph` library in Python is used to implement the GCN model.

The GCN model architecture consists of two graph convolutional layers with 64 units each and `ReLU` activation function. A dropout rate is set to 0.4 and is applied to the convolutional layers to avoid the model from overfitting. The `global average pooling` layer aggregates the node features of the graph into a single vector representation. The output of the `global average pooling` layer is then fed into two fully connected `dense` layers, the first with 32 units and `ReLU` activation, and the second with 5 units and `softmax` activation, which generates a probability distribution over the five possible classes. The `Adam` optimizer is used to minimize `sparse categorical cross-entropy`, which computes the difference between the predicted and true class labels. The accuracy metric is used to assess the model's performance on the test data. These values are summarized in Table 3. Experiment results obtained with GCN are explained in Section 3.5.

Table 3: GCN Hyperparameter Values

Hyperparameter	Value
Number of GCN layers	2
Number of units per GCN layer	64
Dense layer sizes	[32, 5]
Activation	[relu, softmax]
Dropout rate	0.4
Learning rate	0.001
Optimization algorithm	Adam
Loss function	Sparse categorical cross-entropy

3.4.2 Word2Vec-GAT

Graph Attention Network is designed to use self-attention mechanism to learn the node embeddings in the graph by aggregating information from the neighboring nodes. The `GATConv` layer, available in the `spektral` library of Python, is used to implement the GAT model. In our implementation, we use two GAT layers that consist of 64 hidden units with a dropout rate of 0.5 and `elu` activation function.

The `attn_heads` parameter defines the number of attention heads used by the GAT model. Each head computes a separate attention coefficient for each neighbor of a node and then concatenates the results. The value of this hyperparameter is set to 8 and the dropout rate is set to 0.4 in the GAT layers. The `global sum pooling` layer with 64 units and `relu` activation function and an output layer with 5 units and `softmax` activation function are added. The `Adam` optimizer is used with a `learning rate` of 0.005, `categorical cross-entropy` loss function is used to compute the variance between the predicted and actual labels, and the accuracy metric is used to assess the performance of the model. These values are summarized in Table 4. Experiment results obtained with GAT are explained in Section 3.5.

Table 4: GAT Hyperparameter Values

Hyperparameter	Value
Number of GAT layers	2
Number of units per GAT layer	64
Dense layer sizes	[64,5]
Activation	[elu, relu, softmax]
Attention heads	8
Dropout rate	0.4
Learning rate	0.005
Optimization algorithm	Adam
Loss function	Categorical cross-entropy

3.4.3 Word2Vec-GraphSAGE

GraphSAGE is a graph neural network that learns node embeddings by aggregating information from a node's local neighborhood [36]. In our implementation, we use `GraphSAGENodeGenerator` to generate training and validation batches, with batch size and the number of samples specified in the generator. The model consists of two `GraphSAGEConv` layers, available in Python's `spektral` library, with hidden dimensions of 32 and `ReLU` activation function.

The dropout rate is set to 0.5 in each layer. `Global max pooling` operation is applied to obtain a single feature vector representing the entire graph, which is then fed into a `dense` output layer with a `softmax` activation function to generate the final classification output. The model is compiled using the `Adam` optimizer with a `learning rate` of 0.005, `categorical cross-entropy` loss, and accuracy as the evaluation metric. These values are summarized in Table 5.

Table 5: GraphSAGE Hyperparameter Values

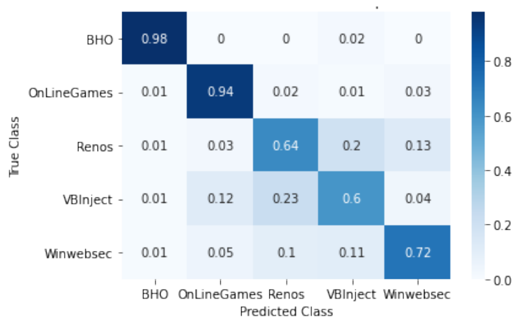
Hyperparameter	Value
Number of GraphSAGE layers	2
Number of units per GraphSAGE layer	32
Dense layer sizes	[32, 5]
Activation	[relu, softmax]
Dropout rate	0.5
Learning rate	0.005
Optimization algorithm	Adam
Loss function	Categorical cross-entropy

3.5 Classification Results

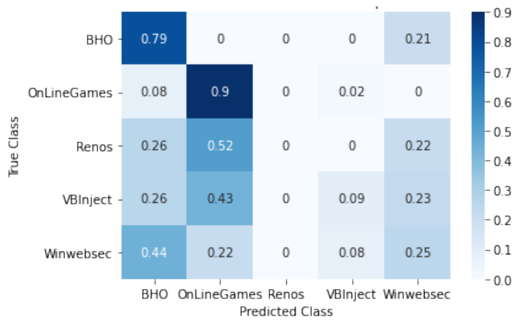
We train GNN models with graph samples using the best hyperparameter values to investigate their classification performance. A baseline result is first established for graph classification without any word embeddings to observe the effect of word embeddings on classification performance. Next, we experiment with Word2Vec embeddings, varying the vector length from 1 to 100. 5,000 graph samples are generated for each vector length category and the classification performance is evaluated using accuracy and classification matrices for each GNN model. Comparing the classification results helps in gaining insights into how the quality of feature vectors affects the classification performance of the GNN models.

3.5.1 GCN Results

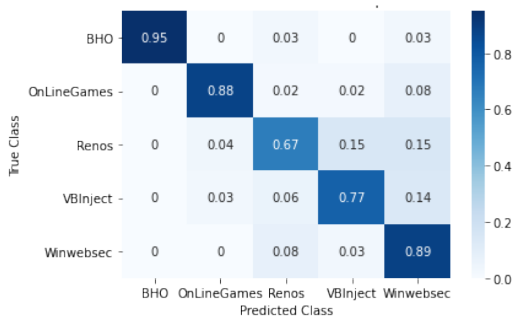
Figure 6 gives the confusion matrices for GCN. This model achieves an accuracy of 79.60% for the baseline model, 60.20% for Word2Vec with vector length of 1, 84.70% for Word2Vec with vector length of 20, 85.3% for Word2Vec with vector length of 50 and 91.10% for Word2Vec with vector length of 100.



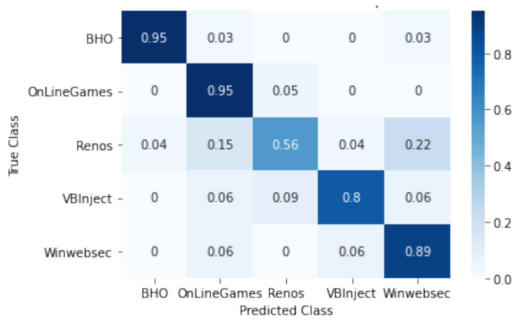
(a) Baseline



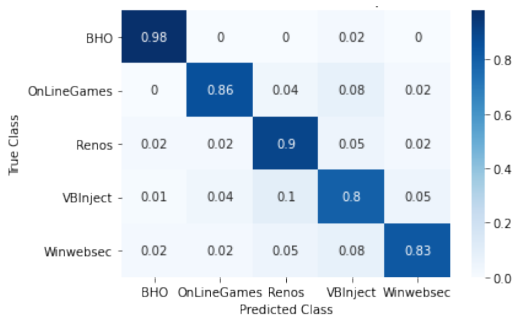
(b) Vector length=1



(c) Vector length=20



(d) Vector length=50

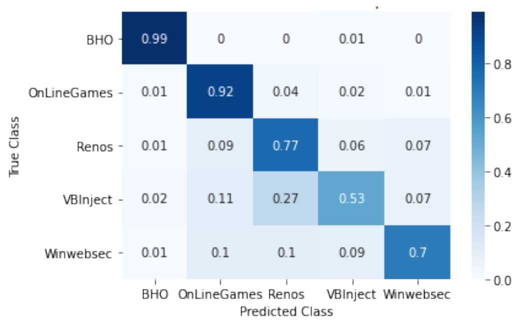


(e) Vector length=100

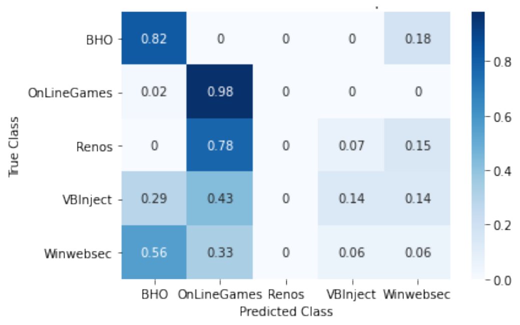
Figure 6: Classification Matrices for GCN

3.5.2 GAT Results

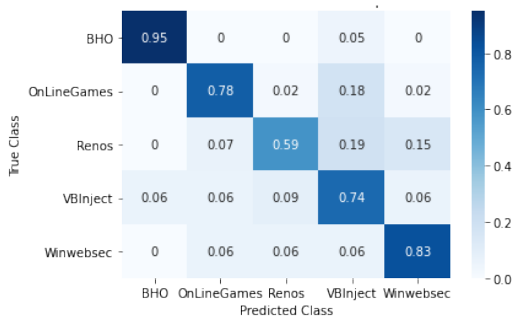
Figure 7 gives the confusion matrices for GAT. This model achieves an accuracy of 73.80% for the baseline model, 42.90% for Word2Vec with vector length of 1, 80.80% for Word2Vec with vector length of 20, 83.80% for Word2Vec with vector length of 50 and 87.30% for Word2Vec with vector length of 100.



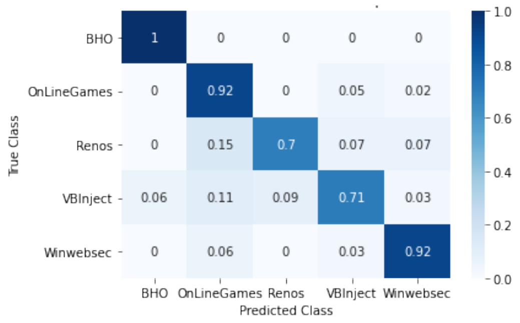
(a) Baseline



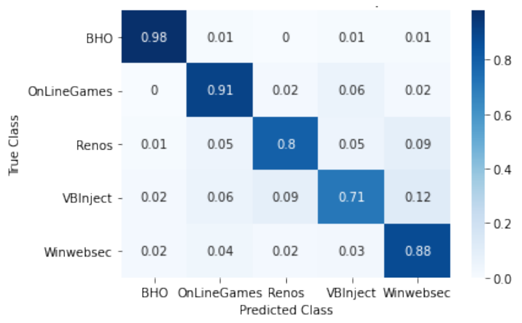
(b) Vector length=1



(c) Vector length=20



(d) Vector length=50

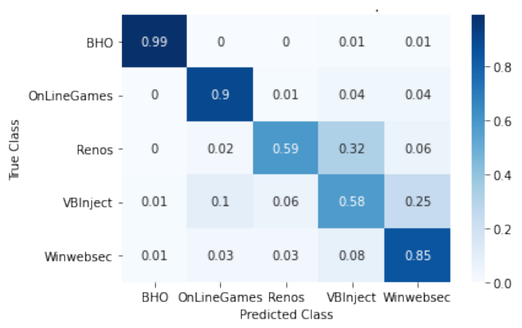


(e) Vector length=100

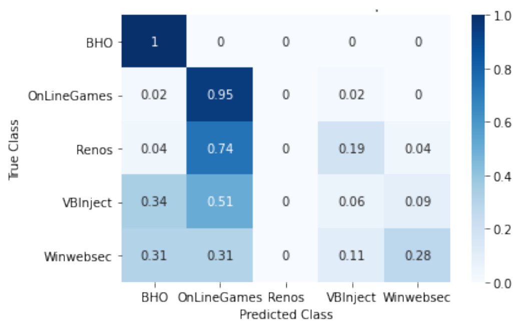
Figure 7: Classification Matrices for GAT

3.5.3 GraphSAGE Results

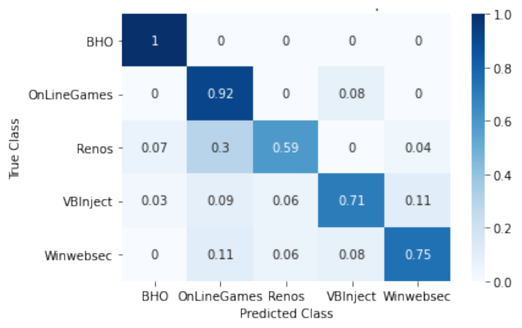
Figure 8 gives the confusion matrices for GraphSAGE. This graph model achieves an accuracy of 75.90% for the baseline model, 47.50% for Word2Vec with vector length of 1, 76.80% for Word2Vec with vector length of 20, 82.70% for Word2Vec with vector length of 50 and 84.70% for Word2Vec with vector length of 100.



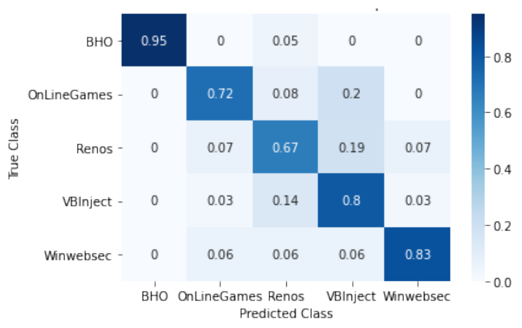
(a) Baseline



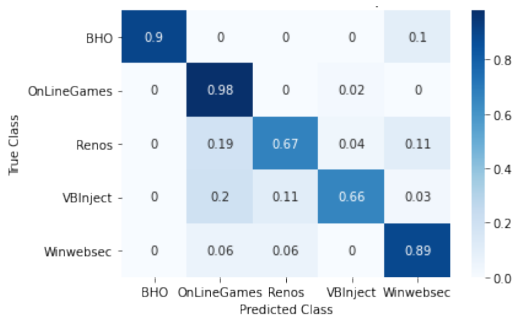
(b) Vector length=1



(c) Vector length=20



(d) Vector length=50



(e) Vector length=100

Figure 8: Classification Matrices for GraphSAGE

3.5.4 Discussion

Figure 9 shows the accuracy achieved by each GNN architecture for the baseline model and Word2Vec embeddings with vector lengths of 1, 20, 50, and 100. The results indicate that the classification accuracy improves significantly as the length of the embedded vector increases. The experiments are concluded at vector length 100 because there is no significant improvement in accuracy observed beyond this length, even when it is increased up to 200. The results also show that GCN outperforms GAT and GraphSAGE, achieving an accuracy of 91.10% for a vector length of 100. GAT and GraphSAGE produced very similar results, with GAT performing slightly better.

Based on the findings, it can be inferred that as the length of the Word2Vec vectors increases, the models become capable of capturing more fine-grained details of the opcode sequence. This, in turn, leads to the creation of higher-quality feature vectors, which are more effectively utilized by the GNNs to capture the underlying graph structure and classify nodes accurately.

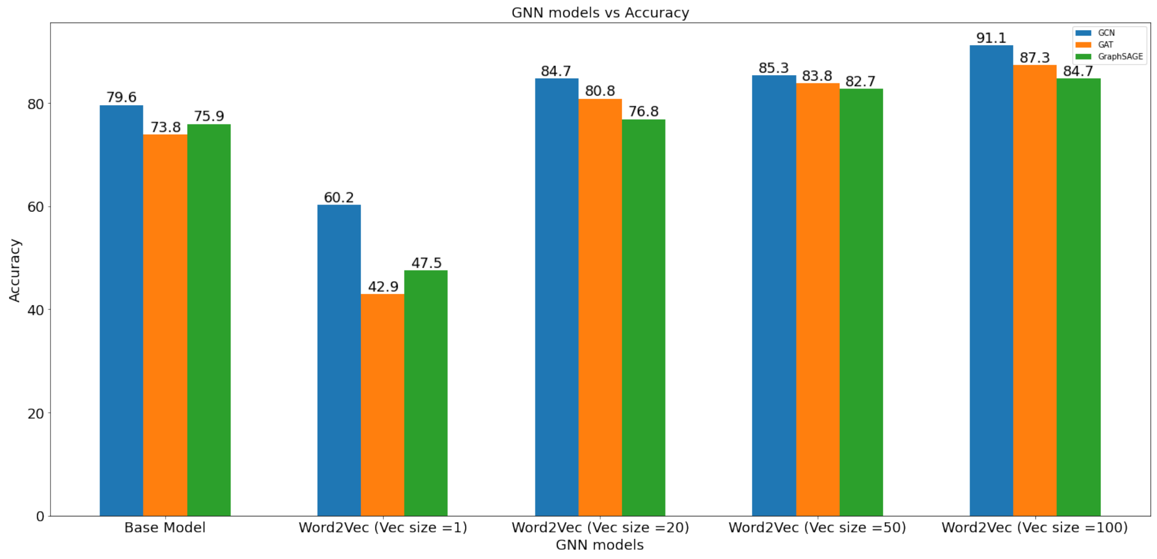


Figure 9: Accuracy for GNN models with varying Word2Vec vector lengths

CHAPTER 4

Conclusion and Future Study

In this study, we analyzed the impact of different word embedding techniques on the performance of Graph Neural Networks in classifying malware files with opcode sequences. The results of our experiments provide strong evidence that using word embeddings can improve feature engineering in malware analysis, resulting in improved classification performance. We evaluated how well Graph Convolution Network, Graph Attention Network, and GraphSAGE network perform in classifying malware files using knowledge graphs constructed from opcode sequences. Our results indicate that GCN outperforms GAT and GraphSAGE, achieving an accuracy of 91.10% for an embedded vector of length 100.

The first set of experiments investigated the impact of different word embedding techniques, including Word2Vec, TF-IDF, and Bag-of-words, on the classification performance of GNN models. The results showed that Word2Vec produces the most effective word embeddings, serving as a baseline for comparison in subsequent experiments. We then generated vector embeddings of various lengths using Word2Vec and constructed knowledge graphs with these embeddings as node features. Through performance comparison of the GNN models, we demonstrated that feature-embedded graphs with larger feature vectors improve the models' performance in classifying the malware files into their respective families.

Our results indicate that the length of the Word2Vec vectors has a significant impact on the models' classification performance. As the length of the embedded vector increases, the models become capable of capturing more fine-grained details of the opcode sequence, leading to the creation of higher-quality feature vectors that are more effectively utilized by the GNNs to capture the underlying graph structure and classify nodes accurately. The experiments showed that the classification accuracy improves significantly as the length of the embedded vector increases. However, there was no noticeable change in accuracy beyond a vector length of 100.

Our study has several implications for the field of malware analysis. First, the use of GNNs for malware classification has shown promising results, indicating the potential of using graph-based approaches for malware analysis. Second, the effectiveness of word embeddings in improving feature engineering in malware analysis highlights the importance of using appropriate feature extraction techniques. Finally, our study highlights the importance of selecting appropriate GNN architectures and hyperparameters for graph-based classification tasks.

One possible direction for future work in this field is to investigate the effectiveness of other word embedding techniques, such as GloVe [44] and FastText [45]. Moreover, it would be worthwhile to explore the impact of other graph construction techniques, such as subgraph sampling [46] and random walks [47], on the classification performance of GNN models. As we are dealing with only five malware families with 1,000 malware files in each, it would be valuable to assess the performance of GNN models on larger datasets to evaluate their scalability and robustness.

Another potential future research can focus on exploring other GNN architectures, such as the recently proposed Transformer-based GNNs [48], and comparing their performance with traditional GNN models. Other than word embedding generated feature vectors, additional features, such as file size and entropy, can be added to the graph to improve the classification performance of GNNs. Furthermore, incorporating temporal information into the graph, such as the order in which opcodes were executed, could potentially enhance the GNN's ability to classify malware.

LIST OF REFERENCES

[1] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, ‘‘Graph neural networks: A review of methods and applications,’’ 2021.

[2] L. Ding, X. Chen, and Y. Xiang, ‘‘Negative-supervised capsule graph neural network for few-shot text classification,’’ *arXiv preprint arXiv:2101.00736*, pp. 6875--6887, 2021.

[3] B. Wang, A. Wang, F. Chen, Y. Wang, and C.-C. J. Kuo, ‘‘Evaluating word embedding models: methods and experimental results,’’ *APSIPA Transactions on Signal and Information Processing*, 2019. [Online]. Available: https://doi.org/10.1017%2Fatsip.2019.12

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, ‘‘Efficient estimation of word representations in vector space,’’ 2013.

[5] S. Robertson, ‘‘Understanding inverse document frequency: On theoretical arguments for idf,’’ *Journal of Documentation - J DOC*, vol. 60, pp. 503--520, 10 2004.

[6] W. Qader, M. M. Ameen, and B. Ahmed, ‘‘An overview of bag of words;importance, implementation, applications, and challenges,’’ 06 2019, pp. 200--204.

[7] M. Stamp, M. Alazab, and A. Shalaginov, *Malware analysis using artificial intelligence and deep learning*, 1st ed. Switzerland: Springer, Dec. 2020.

[8] C. Ventures, ‘‘Cybercrime damages $6 trillion by 2025,’’ 2021. [Online]. Available: https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/

[9] Kaspersky, ‘‘Malware variety grew by 13.7% in 2019,’’ 2019. [Online]. Available: https://usa.kaspersky.com/about/press-releases/2019_malware-variety-grew-by-137-percent-in-2019

[10] O. Savenko, A. Nicheporuk, I. Hurman, and S. Lysenko, ‘‘Dynamic signature-based malware detection technique based on api call tracing,’’ in *ICTERI Workshops*, 2019.

[11] W. Liu, P. Ren, K. Liu, and H. Duan, ‘‘Behavior-based malware analysis and detection,’’ in *2010 International Conference on Intelligent Computing and Integrated Systems.* IEEE, 2010, pp. 455--458.

[12] T. Bilot, N. E. Madhoun, K. A. Agha, and A. Zouaoui, ‘‘A survey on malware detection with graph representation learning,’’ 2023.

[13] Z. Zhang, Y. Li, W. Wang, H. Song, and H. Dong, ‘‘Malware detection with dynamic evolving graph convolutional networks,’’ *Int. J. Intell. Syst.*, vol. 37, no. 10, p. 7261–7280, mar 2022. [Online]. Available: https://doi.org/10.1002/int.22880

[14] S. Li, Q. Zhou, R. Zhou, J. Li, and H. Chen, ‘‘Intelligent malware detection based on graph convolutional network,’’ *Journal of Supercomputing*, vol. 78, no. 5, pp. 4182--4198, 2022.

[15] C. Catal, H. Gündüz, and A. Ozcan, ‘‘Malware detection based on graph attention networks for intelligent transportation systems,’’ *Electronics*, vol. 10, p. 2534, 10 2021.

[16] V. Ravi, M. Alazab, S. Selvaganapathy, and R. Chaganti, ‘‘A multi-view attention-based deep learning framework for malware detection in smart healthcare systems,’’ *Computer Communications*, vol. 195, pp. 73--81, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366422003231

[17] S. Li, Q. Zhou, R. Zhou, and Q. Lv, ‘‘Intelligent malware detection based on graph convolutional network,’’ *The Journal of Supercomputing*, vol. 78, 02 2022.

[18] Y.-H. Chen, J.-L. Chen, and R.-F. Deng, ‘‘Similarity-based malware classification using graph neural networks,’’ *Applied Sciences*, vol. 12, no. 21, 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/21/10837

[19] X. Yang, D. Yang, and Y. Li, ‘‘A hybrid attention network for malware detection based on multi-feature aligned and fusion,’’ *Electronics*, vol. 12, no. 3, 2023. [Online]. Available: https://www.mdpi.com/2079-9292/12/3/713

[20] A. Kale, F. Di Troia, and M. Stamp, ‘‘Malware classification with word embedding features,’’ 03 2021.

[21] A. Chandak, W. Lee, and M. Stamp, ‘‘A comparison of word2vec, hmm2vec, and pca2vec for malware classification,’’ 03 2021.

[22] W. Qader, M. M. Ameen, and B. Ahmed, ‘‘An overview of bag of words;importance, implementation, applications, and challenges,’’ 06 2019, pp. 200--204.

[23] K. Juluru, H.-H. Shih, K. N. Keshava Murthy, and P. Elnajjar, ‘‘Bag-of-words technique in natural language processing: A primer for radiologists,’’ *RadioGraphics*, vol. 41, no. 5, pp. 1420--1426, 2021, pMID: 34388050. [Online]. Available: https://doi.org/10.1148/rg.2021210025

[24] S. Qaiser and R. Ali, ‘‘Text mining: Use of tf-idf to examine the relevance of words to documents,’’ *International Journal of Computer Applications*, vol. 181, 07 2018.

[25] T. Mikolov, K. Chen, G. Corrado, and J. Dean, ‘‘Efficient estimation of word representations in vector space,’’ 2013.

[26] T. P. Adewumi, F. Liwicki, and M. Liwicki, ‘‘Word2vec: Optimal hyper-parameters and their impact on nlp downstream tasks,’’ 2021.

[27] T. Mikolov, K. Chen, G. Corrado, and J. Dean, ‘‘Efficient estimation of word representations in vector space,’’ 2013.

[28] S. Alashri, S. Alzahrani, M. Alhoshan, I. Alkhanen, S. Alghunaim, and M. Alhassoun, ‘‘Lexi-augmenter: Lexicon-based model for tweets sentiment analysis,’’ in *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, 2019, pp. 7--10.

[29] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, ‘‘The graph neural network model,’’ *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61--80, 2009.

[30] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, ‘‘The graph neural network model,’’ *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61--80, 2009.

[31] S. Zhang, H. Tong, J. Xu, and J. Ye, ‘‘Graph convolutional networks: a comprehensive review,’’ *Computational Social Networks*, vol. 6, no. 1, p. 11, 2019.

[32] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, ‘‘Graph attention networks,’’ 2018.

[33] J. Huang, N. Tao, H. Chen, Q. Deng, W. Wang, and J. Wang, ‘‘Semi-supervised text classification based on graph attention neural networks,’’ in *2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD)*, 2021, pp. 325--330.

[34] Y. Liu and X. Gou, ‘‘A text classification method based on graph attention networks,’’ in *2021 International Conference on Information Technology and Biomedical Engineering (ICITBE)*, 2021, pp. 35--39.

[35] W. L. Hamilton, R. Ying, and J. Leskovec, ‘‘Inductive representation learning on large graphs,’’ 2018.

[36] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, “Graph convolutional neural networks for web-scale recommender systems,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &amp Data Mining.* ACM, jul 2018. [Online]. Available: https://doi.org/10.1145%2F3219819.3219890

[37] L. Yao, C. Mao, and Y. Luo, “Graph convolutional networks for text classification,” 2018.

[38] Microsoft Security Intelligence, “Trojan:win32/bho,” https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/BHO&threatId=-2147364778, 2010.

[39] Microsoft Security Intelligence, “Pws:win32/onlinegames,” https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=PWS%3AWin32%2FOnLineGames, 2010.

[40] Microsoft Security Intelligence, “TrojanDownloader:Win32/Renos,” https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=TrojanDownloader:Win32/Renos&threatId=16054, 2010.

[41] Microsoft Security Intelligence, “VBInject,” https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=VirTool:Win32/VBInject%26ThreatID=-2147367171, 2010.

[42] Microsoft Security Intelligence, “Winwebsec,” https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Win32%2fWinwebsec, 2010.

[43] A. Voytetskiy, A. Herbert, and M. Poptsova, “Graph neural networks for z-dna prediction in genomes,” *bioRxiv*, 2022. [Online]. Available: https://www.biorxiv.org/content/early/2022/08/25/2022.08.23.504929

[44] J. Pennington, R. Socher, and C. Manning, “Glove: Global vectors for word representation,” vol. 14, 01 2014, pp. 1532--1543.

[45] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching word vectors with subword information,” 2017.

[46] J. Wang, P. Chen, B. Ma, J. Zhou, Z. Ruan, G. Chen, and Q. Xuan, “Sampling subgraph network with application to graph classification,” 2021.

[47] D. Jin, R. Wang, M. Ge, D. He, X. Li, W. Lin, and W. Zhang, “Raw-gnn: Random walk aggregation based graph neural network,” 2022.

[48] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim, “Graph transformer networks,” 2020.

APPENDIX

Appendix

A.1 Additional Experiments

We conducted an additional experiment where we implemented BERT, a pre-trained language model that generates contextualized word embeddings. Bidirectional Encoder Representations from Transformers (BERT) has the ability to capture word context and produce more significant representations. The following sections present the implementation details and the outcomes achieved by training GNN models with BERT embeddings.

A.1.1 BERT Implementation

To implement BERT, the BERT model and tokenizer are imported from the `transformers` library in Python. To accommodate the BERT model's ability to process text of length 512, the opcode sequence from the malware file is split into chunks of length 512 and each chunk is processed sequentially. The tokenizer is then used to tokenize the opcodes and generate token IDs. For each opcode in the sequence, the model takes these token IDs as input and generates embeddings of length 768. To generate feature vectors for unique opcodes in the file, the feature vector values for the same opcodes are averaged. These features are then used as node embeddings in the graphs for classification.

A.1.2 Results

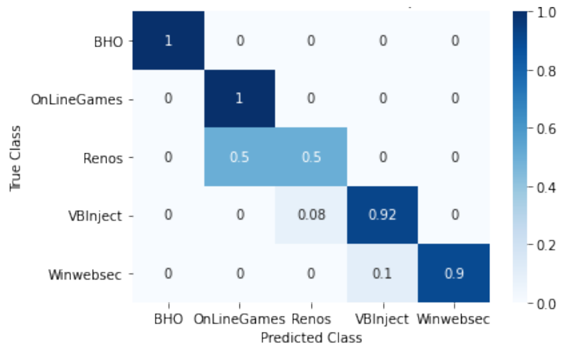
The performance results of GCN, GAT, and GraphSAGE using BERT embedding method are summarized in Table A.6. An accuracy of 89.90% is achieved by GCN, 87.80% by GAT, and 87% by GraphSAGE.

The obtained results are quite comparable to those achieved by GNN models trained with Word2Vec for a vector length of 100. It is evident from the results that using a pre-trained model like BERT in our case is not providing any significant advantage in terms of improved classification performance when compared to Word2Vec, which is not pre-trained.

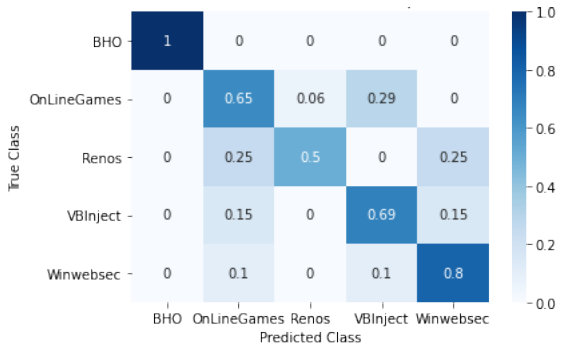
Table A.6: BERT Accuracy

Model	Accuracy
GCN	89.90%
GAT	87.80%
GraphSAGE	87.00%

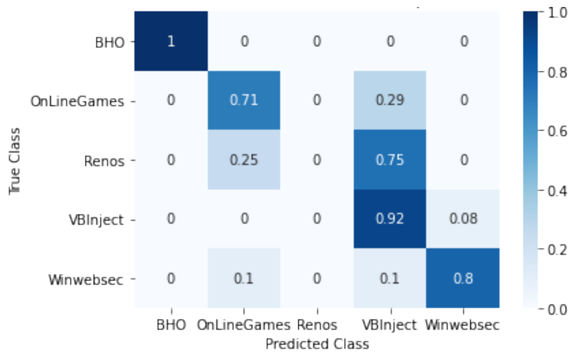
The classification matrices for BERT are shown in Figure A.10.



(a) GCN-BERT



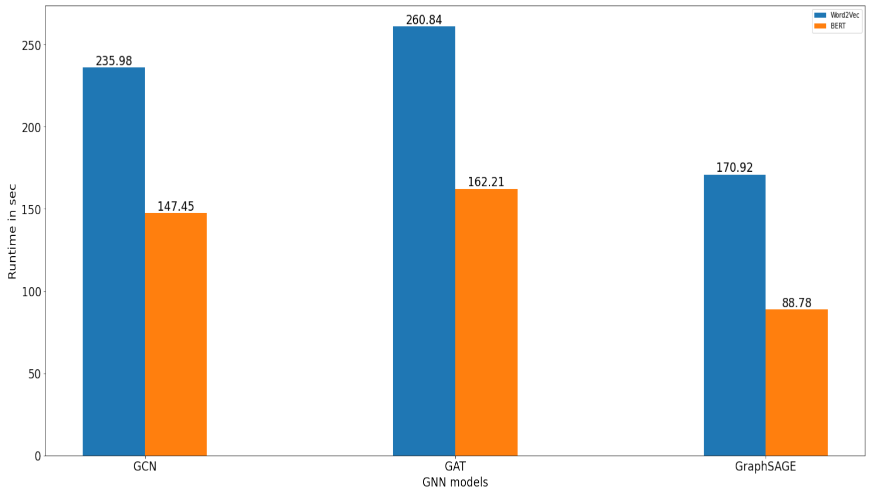
(b) GAT-BERT



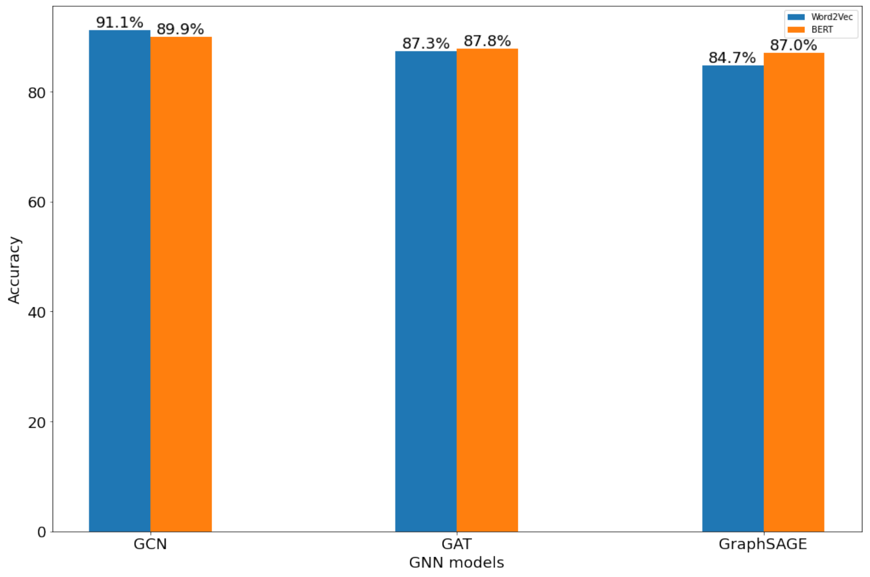
(c) GraphSAGE-BERT

Figure A.10: Classification Matrices for BERT

The classification results obtained using BERT are compared to the results obtained using Word2Vec with vector length of 100. As shown in Figure A.11, it can be observed that the accuracies achieved by BERT and Word2Vec are quite similar. However, BERT outperforms Word2Vec in terms of runtime, taking approximately half the time during training.



(a) Run Time Comparison



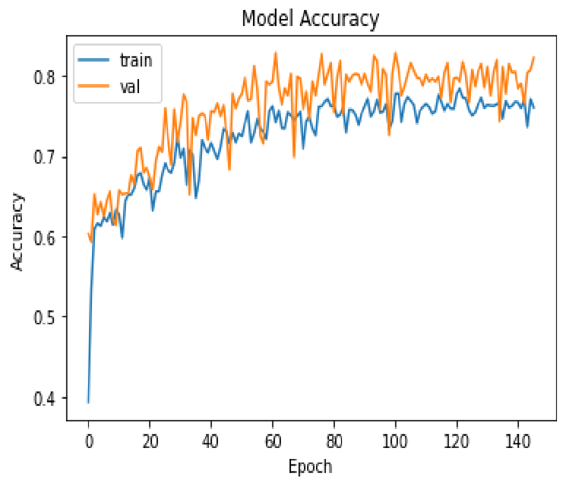
(b) Accuracy Comparison

Figure A.11: BERT vs Word2Vec

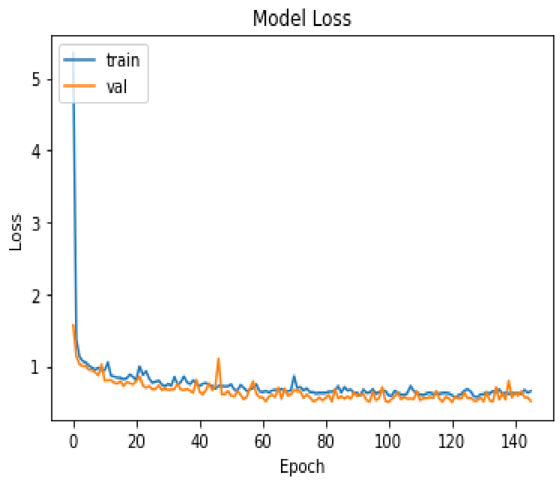
A.2 Additional Results

A.2.1 Accuracy-Loss Graphs

Below are the accuracy-loss graphs of all three models, taken during their training phase with a Word2Vec embedding of vector length 100.

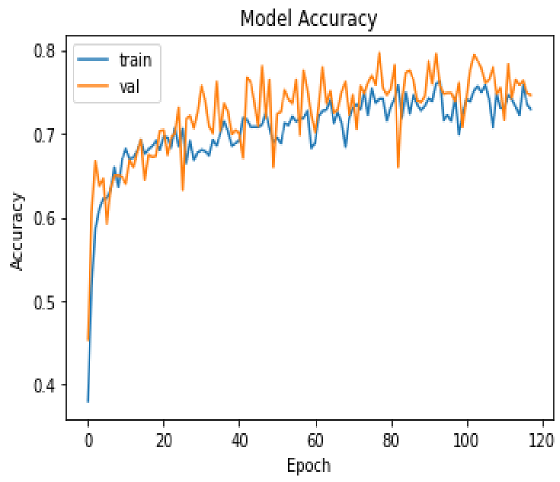


(a) Accuracy

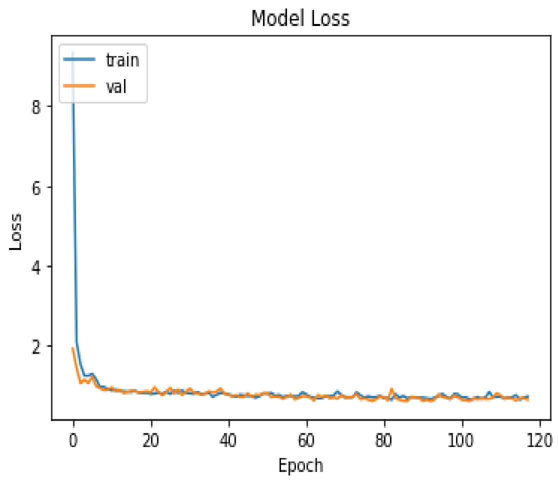


(b) Loss

Figure A.12: GCN Accuracy and Loss Graphs

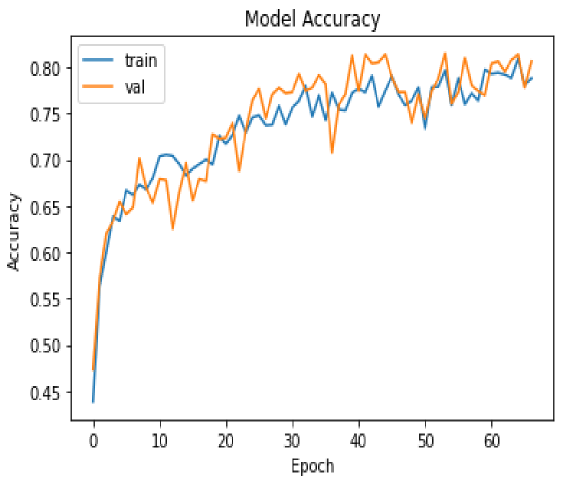


(a) Accuracy

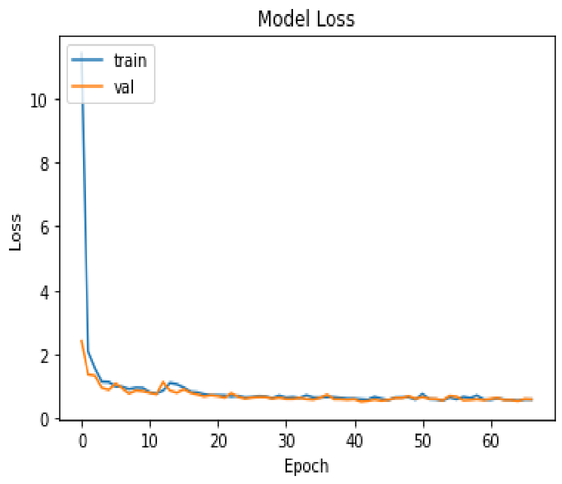


(b) Loss

Figure A.13: GAT Accuracy and Loss Graphs



(a) Accuracy



(b) Loss

Figure A.14: GraphSAGE Accuracy and Loss Graphs

A.2.2 Knowledge Graphs

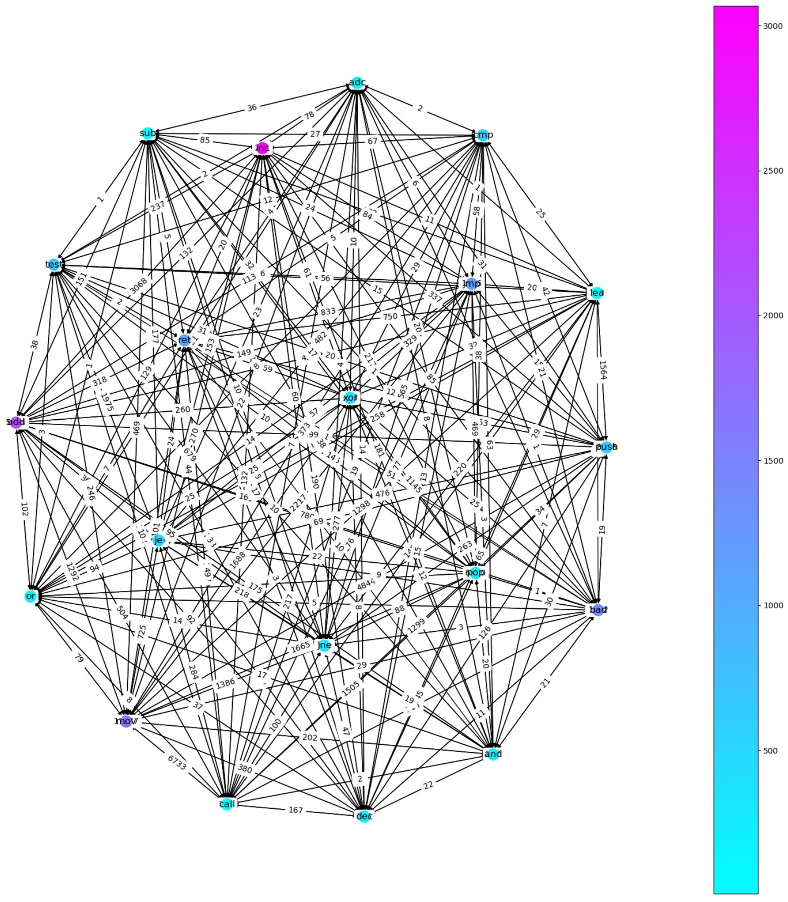


Figure A.15: Regular Graph

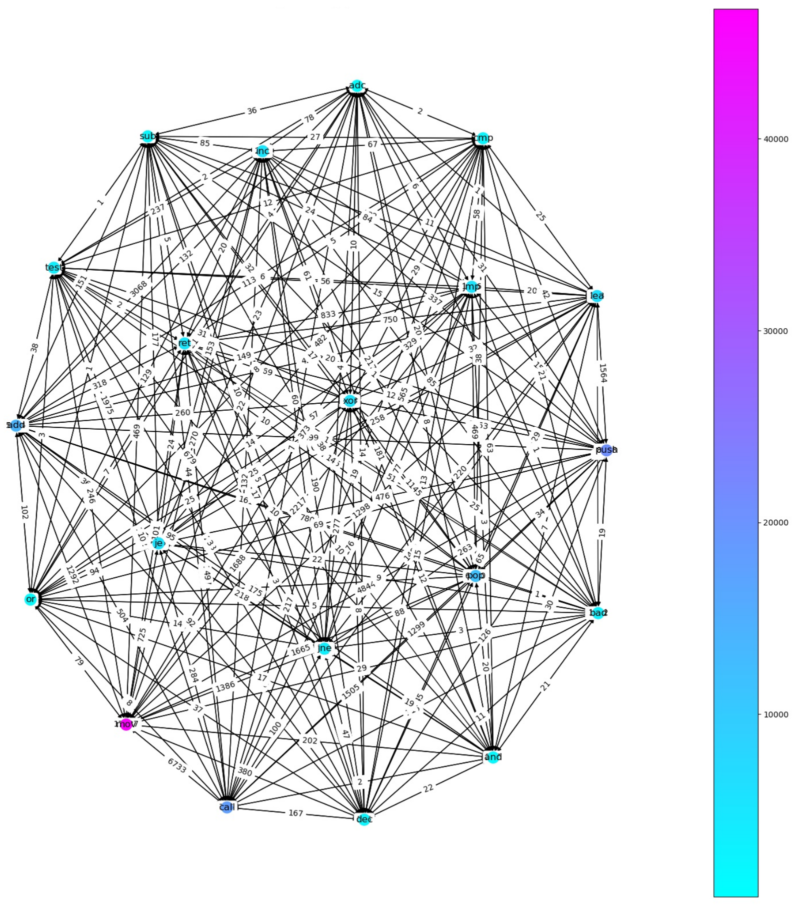


Figure A.16: Bag-of-Words Graph

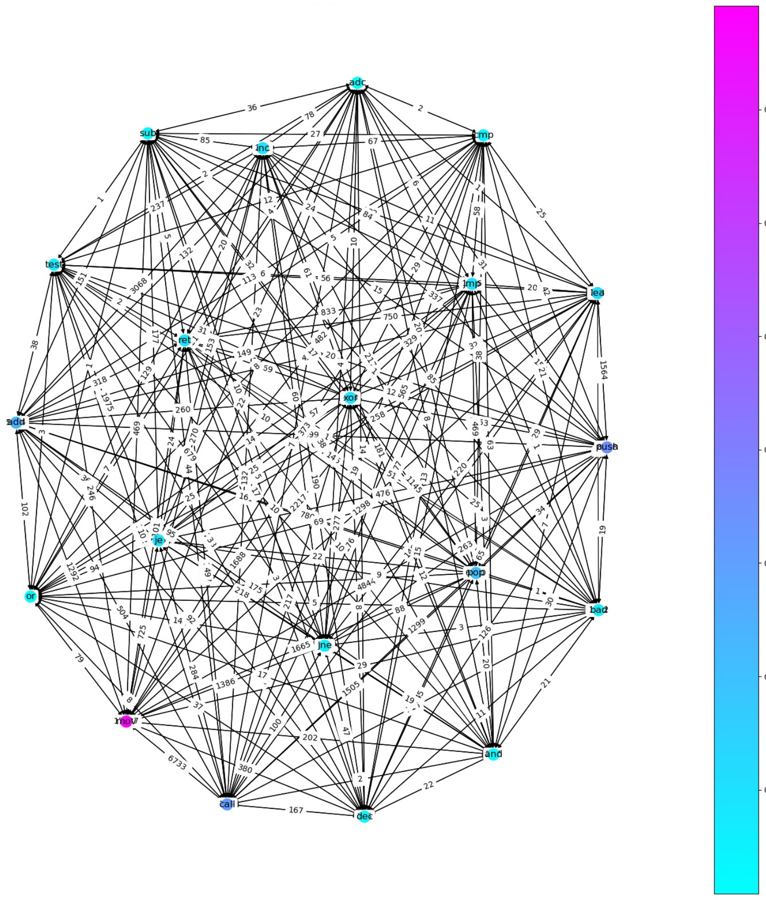


Figure A.17: TF-IDF Graph

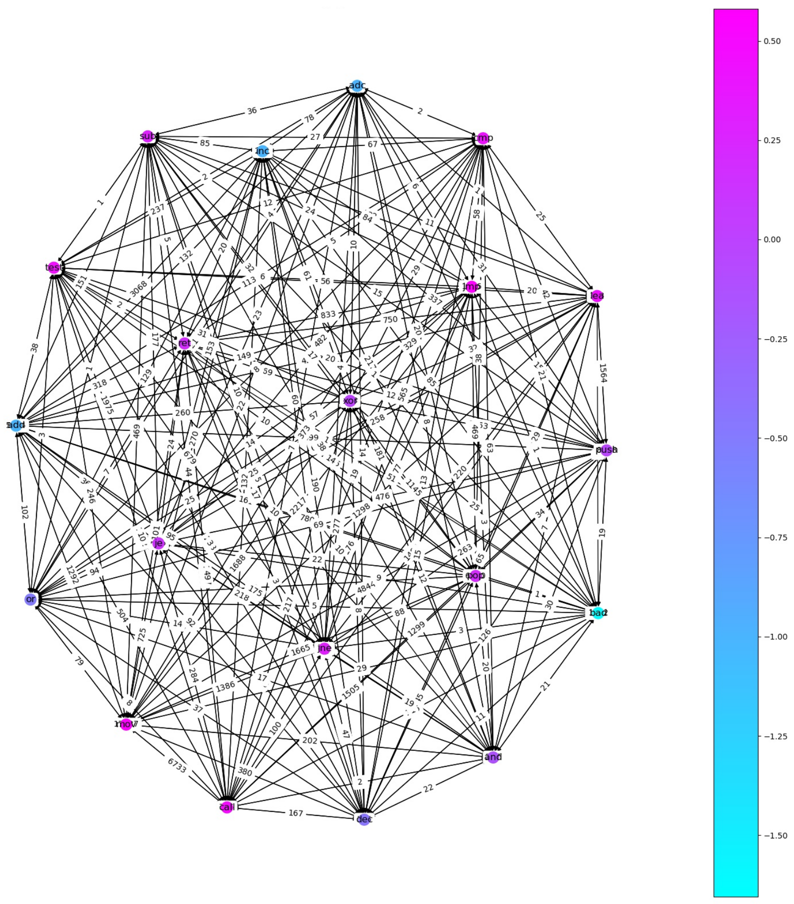


Figure A.18: Word2Vec Graph