THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Blockchain Transaction Graph Analysis Using Graph Neural Networks

Diep Hung VU

A thesis submitted for the degree of Bachelor of Engineering at

The University of Queensland in November 2021

School of Information Technology and Electrical Engineering

Diep Hung Vu
d.vu@uqconnect.edu.au
November 1st, 2021

Prof Amin Abbosh
Head of School
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, Q 4072

Dear Professor Amin,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the division of Software Engineering, I present the following thesis entitled "Blockchain Transaction Graph Analysis Using Graph Neural Networks". This work was performed under the supervision of A. Prof. Marius Portmann.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,
Diep Hung Vu

Acknowledgments

I would like to express my appreciation to A. Prof. Marius Portmann, my thesis supervisor for his dedication in guiding and supporting me throughout the project. Back to the initial stage of the thesis research, he has introduced me to the concepts of Blockchain, Bitcoin and Graph Neural Networks, which are, as a consequence, the areas that I find most interested in. In addition, Dr Marius offered me research-path guidance on various methodologies and case studies. I also want to give thanks to Morris and Evan for helping me to get a better technical understanding of Graph Neural Networks. Last but not least, encouragement and mental support from my family and friends are the most valuable gifts throughout my university studies.

Abstract

Over the last few years, Blockchain technology and Bitcoin digital currency - an application of Blockchain have gained a considerable amount of attention from the public community. The Bitcoin market number has grown remarkably to hundreds of thousands of transactions per day [1]. By downloading the public Bitcoin transaction data within the time range of interests, this thesis is devoted to constructing and analysing the graph that represents the transactions between the Bitcoin addresses.

The analysis tasks would be achieved via extracting and comparing the statistical properties of the associated transaction graphs. In addition, we would then apply the Graph Neural Networks (GNN) for the purpose of visualising the embedding graph nodes in 2D dimensions as well as building and evaluating the link prediction models implemented by two different GNN instances, including Graph Convolutional Networks and GraphSage in order to juxtapose their performance in estimating the likelihood connectivity between two address nodes.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviations	
AEST	Australian Eastern Standard Time
API	Application Programming Interface
AML	Anti Money Laundering
AUC	Area under the ROC Curve
BTC	Bitcoin
BTG	Bitcoin Transaction Graph
DGL	Deep Graph Library
GB	Gigabyte
GNN	Graph Neural Networks
GPU	Graphics processing unit
GUI	Graphical User Interface
LSTM	Long Short-Term Memory
PCA	Principal Component Analysis
PR	PageRank
REST	Representational State Transfer
ROC	Receiver Operating Characteristic
t-SNE	t-distributed Stochastic Neighbor Embedding
UMAP	Uniform Manifold Approximation and Projection

Chapter 1

Introduction

Blockchain - the distributed ledgers ensures the integrity of information be sent between the end-users with a high level of security and automation without the need for the centralized authority to be operated. Bitcoin is the most popular cryptocurrency and the first blockchain implementation, started by Satoshi Nakamoto with the first block on the chain (the Genesis block [10] - block 0) was created on January 3, 2009. From its release to the public until now, there have been more than 700,000 blocks created [9] with the data size reaching over 350 GB [11]. In addition, with the rising attraction from the public community in cryptocurrencies, the number of Bitcoin transactions has grown significantly due to the high demands from consumers.

The entire Bitcoin blockchain database is publicly held and distributed among the end-users in the peer-to-peer network. By accessing that available data, we can formulate the Bitcoin transactions as a graph, and come up with heuristics via graph analysis with the supports from the modern graph framework such as Graph Neural Networks (GNN).

Such associated researches related to Bitcoin transaction graph analysis whether using GNN or not have been done with different approaches and results. One of the publicly available research conducted in 2012 that by downloading the full history of the Bitcoin blockchain, the author of [12] aims to study and estimate how the Bitcoin values were being sent, acquire and managed in each user account by tracking the addresses of the transactions. The fascinating finding is when all the large (50,000 bitcoins) transactions were separated, mostly all of them are strongly related to the single large transaction involving more than 90,000 Bitcoins that occurred in November 2010 by analysing the subgraph of those transactions without using GNN. Other research public recently in 2020 [13] concentrated on using GNN to analyse the interconnections between the transactions to monitor the malicious behaviours, such as money laundering in the Bitcoin transaction graph. Their approach is to highlight the advancement in performance when applying GNN in predicting illicit transactions.

1.1 Description of Work

The goal of this thesis is to build the Bitcoin transaction graph with various block heights [14] range whose data is publicly provided by Blockchain.com [14] and SoChain [15] REST APIs. Following, we would analyse and compare those graphs with and without applying Graph Neural Networks.

Initially, we would do some basic analysis by computing and comparing the graph statistics such as the node degrees, clustering coefficient, weakly/strongly connected components, and use the PageRank algorithm to extract the most important node in the transactions network.

Furthermore, we then apply the Graph Neural Networks with different predefined node features to visualise the node embeddings of each weak with different dimensionality reduction techniques.

Last but not least, we would develop and evaluate a link-prediction model by using GNN. As a result, with the two given address nodes, we could estimate whether they would conduct a transaction in the future.

1.2 Thesis Outline

Firstly, the general background of the technical terms used in this project will be introduced in the next chapter. The related works of this project are then discussed in Chapter 3. Following, we will detail the methodology for building the Bitcoin transaction graph (Chapter 4), visualising the node embeddings (Chapter 5) and implementing the link prediction models (Chapter 6). Last but not least, the conclusion and future work will be stated in the last chapter (Chapter 7).

Chapter 2

Background

This chapter explains the technical concepts and technologies that were employed in the development process, including Blockchain, Bitcoin, Graph Neural Networks and their variants.

2.1 Blockchain

Blockchain technology is a Distributed Ledger Database [16] that contains a chain of blocks. Each block links to the previous block with the cryptographic hash, which ensures that the data in a block can be stored securely and perpetually. The data of a block includes the list of transactions that are generated and transferred by the end-user within the peer-to-peer network. In addition, the transactions within a block vary in time duration and it is not essential that different blocks contain the same number of transactions. Given the structure of exchanging the transactions in the distributed system, the intermediaries parties, such as the bank or the government could be omitted in the Blockchain system, which plays a pivotal role in creating, verifying, and managing the data in the traditional transactions exchanges.

Each block in the chain is linked to the previous one by including the block hash of the previous block in its Block Header. The hash value for a block can be determined by hashing the Block Header twice using the SHA-256 algorithm. The block hash of the current block, therefore, uniquely identifies that block in the Blockchain and is affected by the block hash of the previous one. If any information of a block is adjusted, even though with a small amount, the hash value of the current block is modified as well, which would lead to the inconsistency between the bond between the block and the following one since it contains the invalid hash value of the earlier block.

The Blockchain system is acknowledged with a high level of security due to its consensus protocol that allows a valid block is created and added to the Blockchain ledger. The core solution to solve the problem of security and avoid doublespending (the issue of the same digital token can be sent more than once) [17] is the incentive and "Proof-of-Work" (PoW) [18]. There are certain nodes in the Blockchain network, known as "miners", with a high capacity of computing power, that are competing in generating a new valid block and broadcasting it to the peers in the network. When the "mined"

blocks are confirmed, then the miners would be awarded the currency and the transaction fee [19]. In

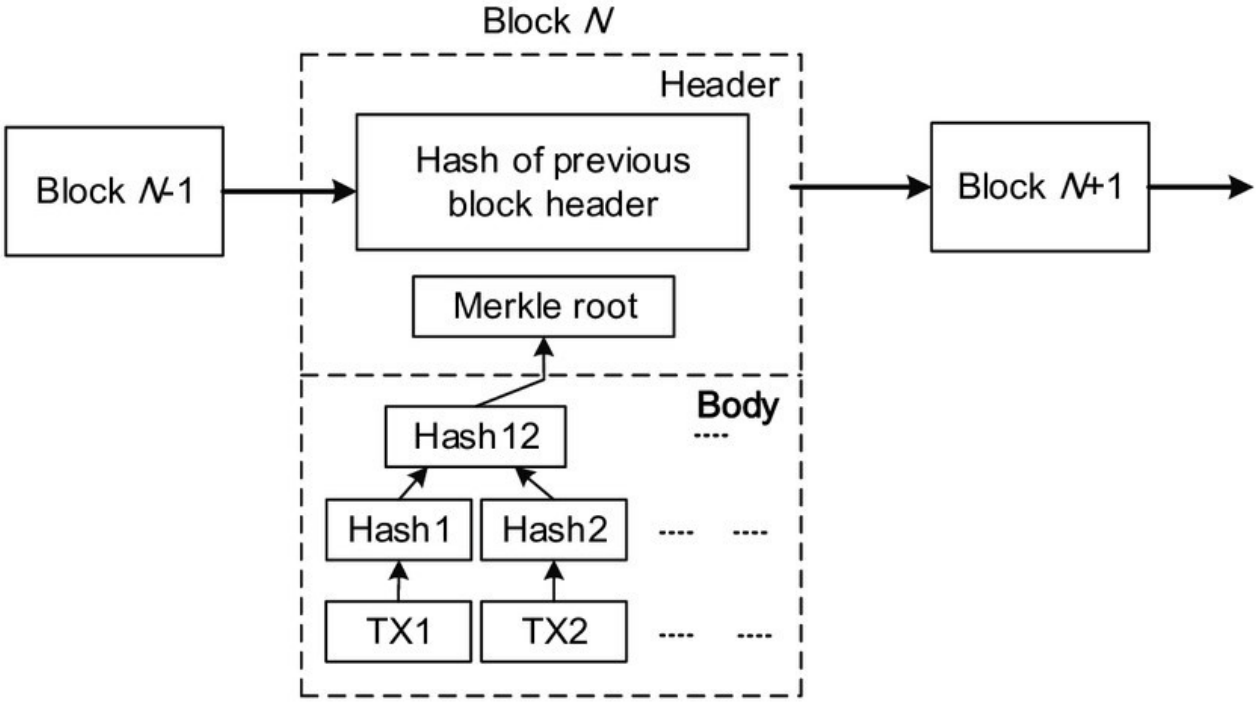


Figure 2.1: The Blockchain structure [2].

general, two distinguished hashing data structures in the blockchain are included in the Block Header (Figure 2.1). One of which is the hash pointer of the current block to the previous block, the other one, called the Merkle root, is the hash that points to all the transactions data of that block. The data structure that stores the transactions in each block is the Merkel Tree, the tree in which in each leave and non-leave node are the cryptographic hash of the transaction data and its children nodes respectively. Therefore, this securely encodes the block data and. Besides, the Merkle tree ensures that the transaction can be traversed in logarithmic space and time [20].

2.2 Bitcoin

Bitcoin is a cryptocurrency powered by Blockchain technology that allows online payment to be sent directly from one end-user to the others anonymously without the intervention of the third parties institution [18].

Figure 2.2: The Bitcoin transaction structure [3].

The Bitcoin transaction (Figure 2.2) is the data structure that holds the information about Bitcoin transferring from one or more addresses to one or more addresses. The transactions are stored in Block Body within a Block (Figure 1) that has three main parts: The metadata, input(s) and output(s).

2.2.1 Input(s)

To conduct a transaction, the senders need to specify the coins that they received in the earlier transactions in order to verify the amount of Bitcoin that they want to send less than or equal to their balance. Therefore, the "`in`" of each transaction is the list of the previous output(s) that the users received the coins in the past. For each input, it contains the "`hash`" pointer to the previous transaction with the index "`n`" of the outputs which proves that the user owns those coins.

There is also the "`scriptSig`" which specifies the signature and the public key of the sender to show that they actually have the ability to claim those previous transaction outputs.

2.2.2 Output(s)

The "`out`" is the list of each output to each Bitcoin address, which contains the "`value`" of Bitcoins to be spent, in `satoshis` (i.e., smallest Bitcoin unit). It is a must that the sum of the output values is less than or equal to the sum of the input values.

The "`scriptPubkey`" contains the Bitcoin scripting language (which begins with "OP" in (Figure 2.2)) and a public key hash of the recipient, or more simply, the Bitcoin address (which begins with "`69e02`" in (Figure 2.2)). Thus, these coins can be redeemed by providing a valid signature and public

key. If that provided information was verified by the Bitcoin scripting language with no errors, the transaction would then be successful.

2.2.3 Bitcoin Address

The Bitcoin address is a hash of the public key that identifies which accounts conduct a transaction. For example, `1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2`. It can be shared with other people to allow them to transfer the Bitcoin to the given address. In contrast to traditional bank account numbers, Bitcoin addresses can be generated autonomously at no cost with no limitations.

In addition to the address that contains the hash string of digits and characters, there is another type of address named Coinbase, which is the input to the first transaction of each block with the arbitrary data. This is because the first transaction is generated by Bitcoins to reward the miner for their efforts in creating a block. Therefore, that input would be generated from nothing and not be referenced from any previous transaction output.

2.3 Bitcoin Transaction Graph

The Bitcoin transaction graph can be modelled with various types depending on the purpose of the project goal. There are three common graph types that all share the common feature as a directed graph and each of which clearly illustrates the source and destination of Bitcoin transaction flow.

2.3.1 Graph Model 1

The first graph model (Figure 2.3) can be represented with the transactions or addresses as vertices and the references between the transaction to transaction or money flow from the transaction to address are the graph edges.

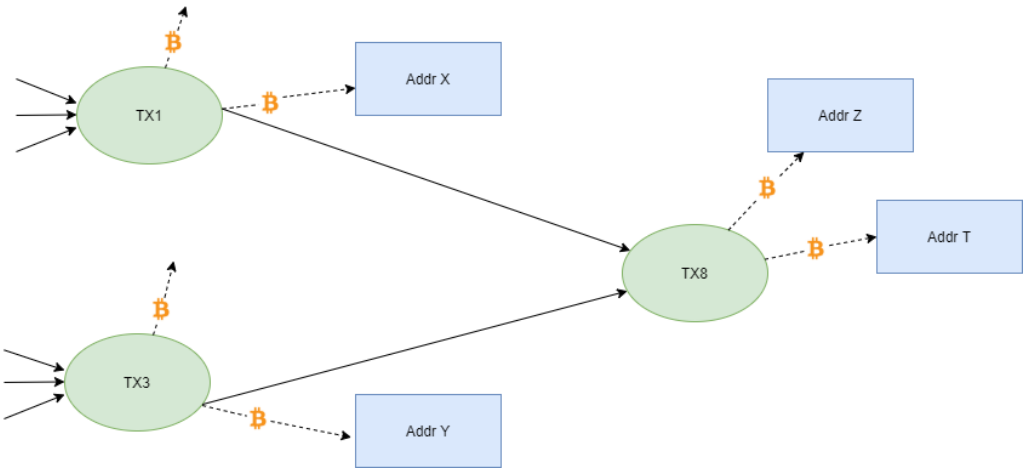


Figure 2.3: Graph Model 1.

This graph model explicitly demonstrates the data structure of Bitcoin transactions with the inputs of each transaction are referenced from the previous ones, and the transaction outputs contain the recipient addresses.

2.3.2 Graph Model 2

For the second graph model (Figure 2.4), it is implicitly implied that any addresses from the previous transactions that are related to the current transaction can directly connect to the current one.

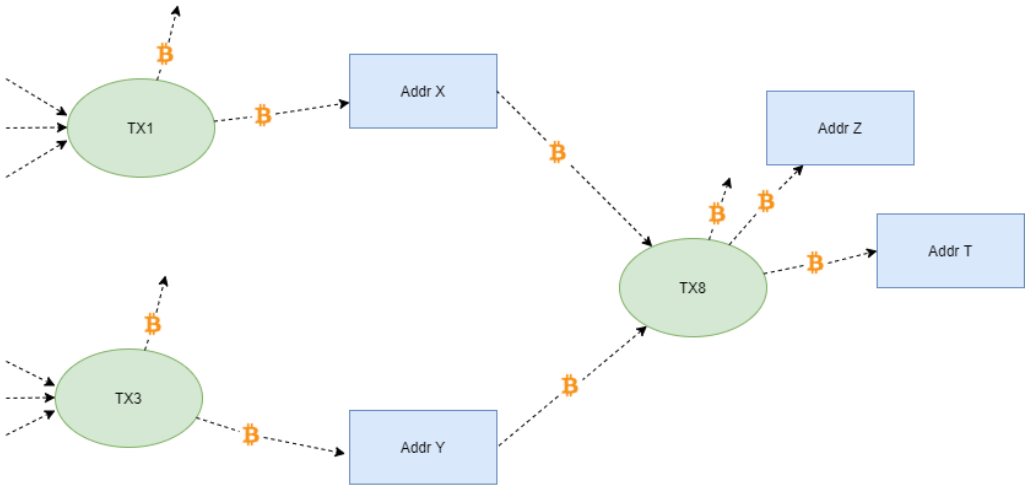


Figure 2.4: Graph Model 2.

Hence, there are two edge types within this graph model, including transaction to address and address to transition. The transaction vertices can be considered as the indirect references of the money flows between the Bitcoin's addresses.

2.3.3 Graph Model 3

The third graph model (Figure 2.5) has only one address and edge type, which are the Bitcoin addresses and money flow among them respectively.

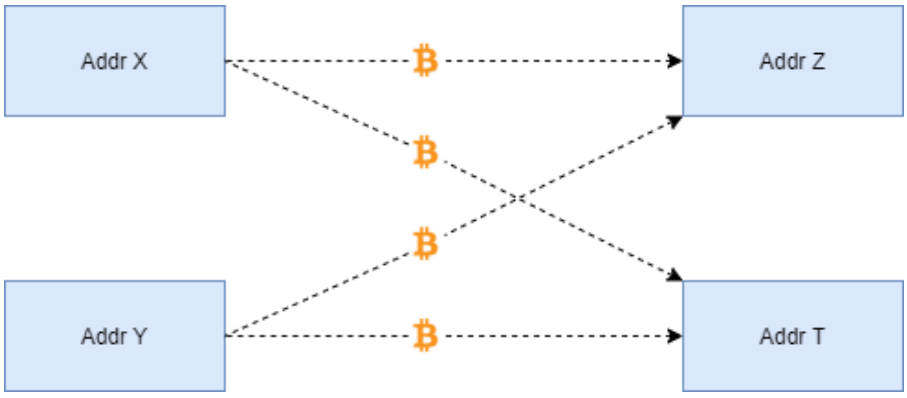


Figure 2.5: Graph Model 3.

Within a transaction, any input address does have the full connection to all of the output addresses. This graph model is also used in this project in further analysis because of its simplicity in vertex and edge demonstration and it also reflects the direct Bitcoin transfer between the Bitcoin addresses.

2.4 Node Embeddings

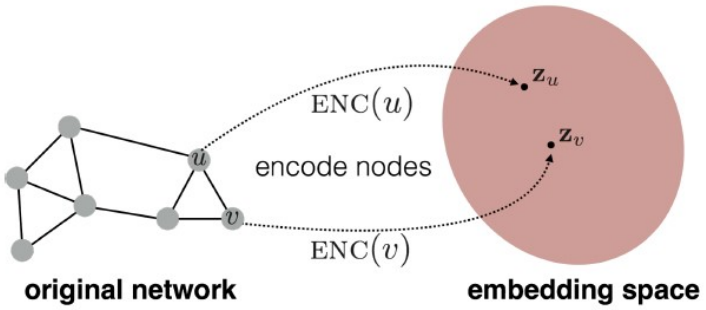


Figure 2.6: Embeddings transformation.

Figure 2.6 demonstrates the transformation of a node in the original graph to the embedding space via the encoder function $ENC(\cdot)$. The goal is to learn that function to optimize the distances in the embedding space reflect the similarity of the nodes in the original network.

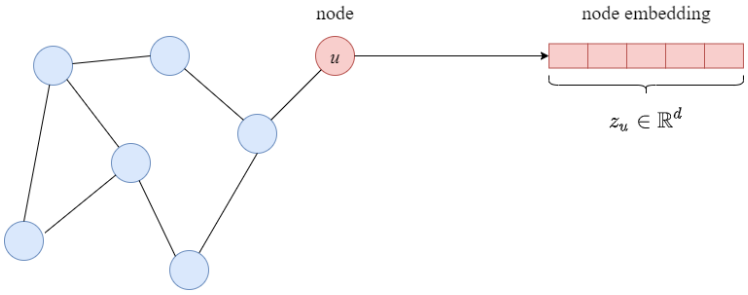


Figure 2.7: Encoding function definition.

Formally, the encoder is the function that maps nodes $u \in \mathcal{V}$ (where $\mathcal{V}$ is the set of all vertices) to their corresponding vector embeddings $z_u \in \mathbb{R}^d$ (Figure 2.7). In the most straightforward and simple case, the encoder can be written as:

$$ENC: \mathbb{R}^{|\mathcal{V}|} \rightarrow \mathbb{R}^d$$

It means that the encoder takes all nodes from the original graph and transform them into the embeddings.

2.5 Graph Neural Networks

Graph Neural Networks (GNN) is the connectionist model that appropriates the problem of capturing the relationships of graphs via message passing between their nodes [21]. In GNN, the encoder is defined as a complex function that depends on the structure of the input graph and the given node features in order to generate the node embeddings representation.

$$ENC(u) = f(A,X)$$

Where

$A \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the adjacency matrix for the graph.
$X \in \mathbb{R}^{|\mathcal{V}|\times d}$ is the matrix for node features.

2.5.1 Message Passing Framework

GNN uses the graph structure A and the node features matrix X to express the vector representation of each target node u, which is z_u.

In each GNN layer, to determine the features aggregation for a single node u, the message passing function aggregates the features from its respective neighbourhood $\mathcal{N}(u)$, and so on for the next GNN layer. The computation graph of the GNN creates a tree structure by spreading the neighbourhood around the target node (Figure 2.8).

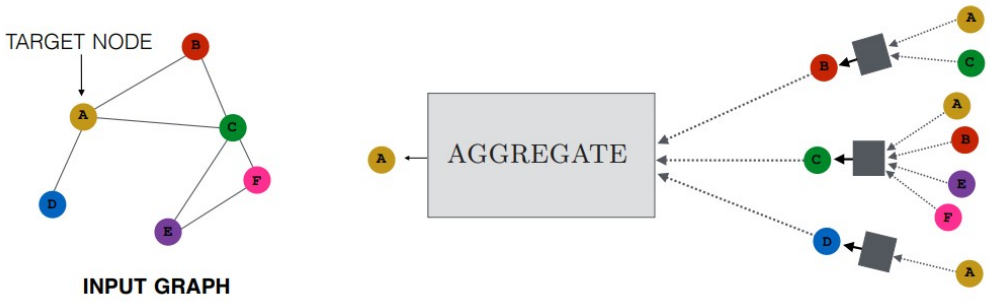


Figure 2.8: Overview of 2-layer GNN model [4].

After k-th layer, the representation vector of node u is $h_u^{(k)}$. We can formulate the message-passing update function at the $(k+1)$-th layer as:

$$h_u^{(k+1)} = UPDATE^{(k)}(h_u^{(k)}, AGGREGATE^{(k)}(h_v^{(k)}, \forall v \in \mathcal{N}(u)))$$

Initially, $h_u^{(0)} = X_u$ and $\mathcal{N}(u)$ is the set of u's neighbour nodes. At each GNN layer, the $AGGREGATE$ function takes the embedding features of all nodes v adjacent to u then generates the aggregated message based on its neighbourhood features. To calculate the new feature for node u at layer $k+1$, the $UPDATE$ function then combines that aggregated message from its neighbourhood, together with

its feature in the previous k layer. After performing K GNN layers, the output of the final layer can be used to demonstrate the embedding for each node. For example:

$$z_u = h_u^{(K)},\ \forall u \in \mathcal{V}$$

Different GNN instantiation follows different neighbourhood aggregation and self feature update strategies (e.g., neural network). The choice of $AGGREGATE^{(k)}$ and $UPDATE^{(k)}$ in GNNs are essential. In this project, two modern GNN approaches used to analyse the Bitcoin transactions network are Graph Convolutional Networks [22] and GraphSage [4].

2.5.2 Graph Convolutional Networks (GCN)

In GCN [22], the embedding of node u at the k-th layer is defined as:

$$h_u^{(k)} = \sigma\left(W_{self}^{(k)} h_u^{(k-1)} + \sum_{v \in \mathcal{N}(u)} W_{neigh}^{(k)} \frac{h_v^{(k-1)}}{|\mathcal{N}(u)|} + b^{(k)}\right)$$

Where $W_{self}^{(k)}, W_{neigh}^{(k)} \in \mathbb{R}^{\,d^{(k)} \times d^{(k+1)}}$ are trainable parameter matrices and σ demonstrates a non-linearity activation function (e.g., a sigmoid, tanh or ReLU). The bias term $b^{(k)} \in \mathbb{R}^{\,d^{(k)}}$ is usually neglected for simplicity notation, but better performance would be achieved when the bias term is included.

In general, neighbourhood message passing and aggregation in a common GNN framework is equivalent to a typical multi-layer perceptron (MLP) or Elman-style recurrent neural network (RNN) [23] because the node embeddings at each layer would multiply to a linear operator W and apply the non-linearity activation function.

Initially, the embeddings of all adjacent nodes of u are summed and normalised by its node degree, then, the neighbourhood aggregated information is combined with the embeddings of node u in the previous layer $k-1$. Eventually, the elementwise non-linearity is applied to generate the final embedding of the node in a single GNN layer. The equivalent representation of a GCN layer with the $UPDATE$ and $AGGREGATE$ can be interpreted as:

$$h_u^{(k)} = UPDATE^{(k)}\left(h_u^{(k)}, m_{\mathcal{N}(u)}^{(k)}\right) \;=\; \sigma\left(W_{self}^{(k)} h_u^{(k-1)} + m_{\mathcal{N}(u)}^{(k)} + b^{(k)}\right)$$

Where

$$m_{\mathcal{N}(u)}^{(k)} = AGGREGATE^{(k)}\left(h_v^{(k-1)}, \forall v \in \mathcal{N}(u)\right) = \sum_{v \in \mathcal{N}(u)} W_{neigh}^{(k)} \frac{h_v^{(k-1)}}{|\mathcal{N}(u)|}$$

Notice that for most cases, the trainable parameters $W_{self}^{(k)}, W_{neigh}^{(k)}$ and $b^{(k)}$ can be shared among all the nodes to generate its embedding features.

2.5.3 GraphSage

GraphSage is a spatial GNN framework that learns the embeddings by sampling and aggregating neighbours from multiple hops. Unlike GNN, GraphSage is an inductive learning algorithm which

means that instead of using a full neighbour set, GraphSage uniformly samples a fixed-size set of neighbours to aggregate information.

According to the GraphSage paper [4], the embedding of node u at the k-th layer is defined as:

$$h_u^{(k)} = \sigma(W_{self}^{(k)}.CONCAT(h_u^{(k-1)}, AGGREGATE(\{h_v^{(k-1)}, \in \mathcal{N}(u)\}))$$

To combine the features of the target node and its neighbours, instead of summing them like GCN, GraphSage uses $CONCAT(\cdot)$ to simply concatenate the node's current representation $h_u^{(k-1)}$, together with the aggregated neighbourhood feature vectors $h_{\mathcal{N}(u)}^{(k-1)}$ GraphSage flexibly gives the options for choosing the $AGGREGATE(\cdot)$ function that can be one of the following:

- **Mean aggregator**: Take the weighted average of neighbour features. GraphSage with a mean aggregator can be interpreted as an inductive version of GCN.

$$\sum_{v \in \mathcal{N}(u)} \frac{h_v^{(k-1)}}{|\mathcal{N}(u)|}$$

- **LSTM aggregator**: Apply LSTM to a reshuffle of neighbours. Since the LSTM aggregator is not permutation invariant, it requires a specific order of the nodes.

$$LSTM\left(\left[h_v^{(k-1)}, \forall v \in \pi(\mathcal{N}(u))\right]\right)$$

- **Pool aggregator**: In this approach, each neighbour's vector is independently fed through fully connected neural networks, and then is followed by an elementwise pooling operation (either $mean(\cdot)$ or $max(\cdot)$) to aggregate features information across the neighbour set.

$$max\left(\{\sigma\left(W_{neigh}^{(k)} h_v^{(k-1)} + b^{(k)}\right), \forall v \in \mathcal{N}(u)\ \}\right)$$

2.6 Deep Graph Library (DGL)

Figure 2.9: DGL [5].

There is various GNN Python library that supports deep learning on graphs. For example, PyTorch Geometric [24], Deep Graph Library (DGL) [5], Graph Nets [25] and Spektral [26]. However, in this project, DGL is the library in use because of its efficiency and scalability and easy-to-use. Furthermore, DGL is memory efficient in message-passing primitive for training GNN, which is scalable to significant graphs training with multi-GPU acceleration and distributed training interface.

Chapter 3

Related Works

3.1 Bitcoin transaction graph analysis

A considerable number of publications have been made on the topic of Bitcoin blockchain analysis by building the transaction graph. There was a research published in 2012 [12] which is considered as the pioneer in clarifying the concerns about the typical behaviour of the users in the Bitcoin blockchain network. The authors (Dorit R. and Adi S.) focused on inspecting the transactions flow between the addresses, to learn the way they acquired and spent Bitcoin with the aim of protecting users' privacy. By downloading the full Bitcoin blockchain data, the authors constructed the graph with respect to all the largest transactions in the Bitcoin scheme that was greater than or equal to 50,000. There were 364 such transactions. By starting with the earliest one which contained 90,000 BTC's made on November 8^{th} 2010, there was an interesting finding that all of the remaining 363 transactions are the descendent of the initial transaction. Furthermore, there were four common transaction graph types have been identified as a result of this research, including:

- **Chains**

  A typical practice of Bitcoin users is to generate the chains of serial transactions.

- **Fork-Merge Patterns and Self Loops**

  Another common approach in the Bitcoin transactions network is when one address wants to send a large number of Bitcoins to another one, the coins tend to be divided up into smaller values and transferred via several intermediate addresses. Each of those split Bitcoin values is then being sent, mostly in full, to the same destination address directly or via other intermediaries.

- **Keeping Bitcoins in "Saving Accounts"**

  In the transactions flow, there is a certain amount of Bitcoin is put aside instead of fully transferred which forms a "saving accounts". As a result, these Bitcoins are not then circulating in the system.

- **Binary Tree-Like Distributions.**

A large amount of Bitcoin is transferred and distributed among the ad-dresses by dividing up to two equally amount at each step. This results in a binary-tree like structure.

3.2 Bitcoin transaction graph analysis with GNN

There were active researches in recent years related to the anti-money laundering (AML) regulations which play an important role in preserving the financial systems, especially for the cryptocurrency exchanging market where the user identity is anonymous. In 2019, the Elliptic dataset [27] was published, containing the time-series graph of more than 200K nodes representing the Bitcoin transactions with 166 node features and over 234K edges describing the direct payment flows (Graph Model 1, Figure 2.3). Such dataset opened up machine learning on the blockchain network where it is known as "the largest labelled transaction dataset publicly available in any cryptocurrency" [28]. The authors of [28] conducted binary classification on that dataset to predict whether a transaction is illicit using variations of different machine learning approaches, such as Logistic Regression, Random Forest, Multilayer Perceptron, and Graph Convolutional Networks (GCN). The result is, the model with two-layer GCN outperforms the Logistic Regression, with the F1 score of GCN being 0.682 and a maximum of 0.543 for the Logistic Regression. It reflects the advantages of using the graph-based model compared to the one agnostic to the graph structure. However, as stated by the authors, the predefined input features of the Elliptic dataset are quite informative. By using these features only, the Random Forest variant achieves the highest F1 score, at 0.796. Being motivated by this work, the three authors in Bournemouth University, United Kingdom [13] did propose a novel approach in 2020 by combining the two-layer GCN with linear layer. Concisely, the output node embeddings from the two GCN layers are concatenated with the output from the linear layer having the initial node features as input. The combined output is then passed with non-linearity activation function (i.e. ReLU), and forwarded into two sequential linear layers (Figure 3.1)

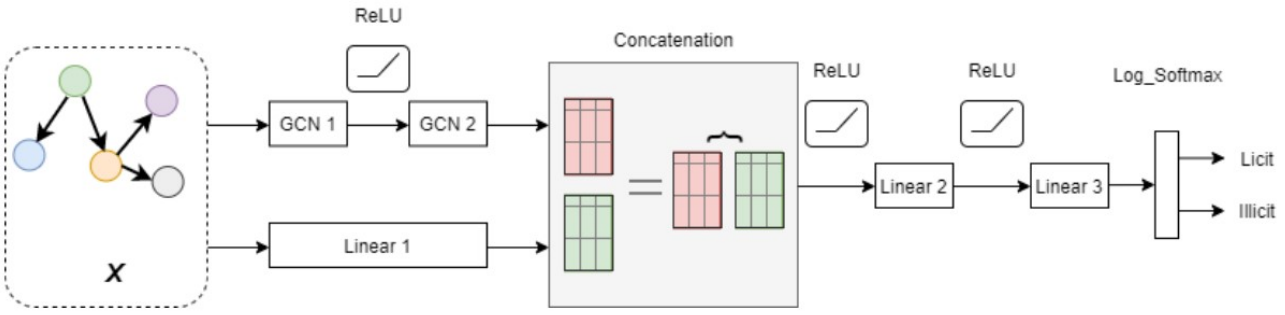


Figure 3.1: Architecture of the method provided by [6], where X describes the original node features matrix. The output represents the predictions of licit/illicit transactions.

As a consequence, the F1 score of this approach reaches 0.773 which considerably outperforms the GCN's experiment used in [28].

Chapter 4

Building Bitcoin transaction graph

This chapter details how the dataset was defined and the progress of building the Bitcoin transaction graph. Moreover, some basic graph analysis (without applying GNN) would be conducted via collecting the graph statistical properties such as nodes/edges count, degrees distribution, weakly/strongly connected components, clustering coefficient and PageRank.

4.1 Dataset description

This thesis mainly focuses on investigating one of the most active Bitcoin trading duration of history until September 2021, beginning from the event when the Chief Executive Officer of Tesla, Inc announced Bitcoin was accepted as a legitimate payment to buy their car products on March 24^{th} [29] to their rejection of that payment method on May 13^{th} [30]. Even though some fluctuations were witnessed, the value of Bitcoin within that period was capped at the highest in history (counted to September 24^{th} 2021), at roughly 63,503.46 USD on April 13^{th} (Figure 4.1).

We will collect the Bitcoin transactions data from that period (7 weeks in total, beginning from March 24^{th} to May 13^{th}) to build the 7 separated transaction graphs with each of which has the duration of one week. After the time durations were identified, we then needed to find the block heights that were mined from the start of starting day to the end of the ending day of each week. The table 4.1 below demonstrates the list of block heights transaction data needs to be fetched corresponding to each predetermined week.

4.2 Download Bitcoin transaction data

With the block heights are provided as parameters, the idea is to write the Python script to fetch all the Bitcoin transaction data within that block range and then build the graph that each node represents the Bitcoin address. For every transaction, each input address (output address from the earlier transaction) is linked to all of the output addresses. If a transaction does not contain the input address which is the coinbase transaction (the first transaction in a block to reward the miner), we would name the

Figure 4.1: Bitcoin to USD Chart in 2021 (from January 1^{st} to September 24^{th}) [7].

Time Duration	Block Heights	Blocks Count
Week 1 (March 24^{th} - March 31^{st})	677,146 - 676,105	1,041
Week 2 (April 1^{st} - April 7^{th})	678,021 - 677,147	874
Week 3 (April 8^{th} - April 14^{th})	679,186 - 678,022	1,164
Week 4 (April 15^{th} - April 21^{st})	680,019 - 679,187	832
Week 5 (April 22^{nd} - April 28^{th})	680,981 - 680,020	961
Week 6 (April 29^{th} - May 5^{th})	682,068 - 680,982	1,086
Week 7 (May 6^{th} - May 13^{th})	683,344 - 682,069	1,275

Table 4.1: Block heights with respect to the period [9].

input address as "`Coinbase`". The Listing 1 below describes the implementation for downloading the Bitcoin transaction of a given block height via fetching the data from the REST APIs provided by Blockchain.com [31] and SoChain [15]. For any exceptions returned from the server, it would sleep for 5 seconds and resend the request to the server. As a result, all the transaction graph edges of that block would then be extended to the *.csv* file via the `f_object`.

Listing 1: Download transaction data within a block.

```
1  def get_block(f_object, b_height):
2      """Append the csv file with the edge list get from a block.
3
4      Parameters
5      ----------
6      b_height: Block height
7      string
8
9      Returns
10     -------
```

```
    void
    """
    while True:
        try:
            # Get block hash of given block height
            b_info = f'https://chain.so//api/v2/get_blockhash/BTC/{b_height}'
            b_hash = requests.get(b_info).json()["data"]["blockhash"]
            # Get full block data
            b_url = f'https://blockchain.info/rawblock/{b_hash}'
            b_data = requests.get(b_url).json()
            with lock:
                # For each transaction in a block
                for index, tx in enumerate(b_data["tx"]):
                    inputs = tx["inputs"]
                    outputs = tx["out"]
                    # Connect each input address to all output addresses
                    for i in inputs:
                        if i["prev_out"] and "addr" in i["prev_out"]:
                            i_address = i["prev_out"]["addr"]
                        else:
                            i_address = "Coinbase"
                        for o in outputs:
                            if o["spent"] and "addr" in o:
                                o_address = o["addr"]
                                # Each csv_row contains a link
                                # between one input address to one output address
                                csv_row = [i_address, o_address]
                                # Append that row to csv file
                                writer_object = csv.writer(f_object)
                                writer_object.writerow(csv_row)
            break
        # Exception from the server
        except:
            # Sleep for 5 seconds and resend the request
            sleep(5)
```

For building the full transaction graph of given block heights range, the `get_block` function would be called concurrently via multi-threading. A detailed implementation is mentioned in the Appendix A.1.

4.3 Results

Table 4.2 shows the nodes, edges and transactions count for the graphs of all weeks. On average, there were about two million transactions in each weak. However, the third weak (April 8^{th} - April 14^{th}) observed the most active time compares to the others with 2,377,24 transactions. Interestingly in that time period, as stated above, the price of Bitcoin was recorded as the top highest in history, at approximately 63,503.46 USD on April 13^{th}.

Time Duration	#Nodes	#Edges	#Transactions
Week 1 (March 24^{th} - March 31^{st})	5,819,494	55,091,298	2,071,943
Week 2 (April 1^{st} - April 7^{th})	4,118,085	35,902,089	1,770,571
Week 3 (April 8^{th} - April 14^{th})	5,649,328	62,084,449	2,377,241
Week 4 (April 15^{th} - April 21^{st})	4,930,159	31,511,389	1,804,494
Week 5 (April 22^{nd} - April 28^{th})	3,595,532	21,526,790	1,909,604
Week 6 (April 29^{th} - May 5^{th})	5,379,547	23,56,465	1,906,800
Week 7 (May 6^{th} - May 13^{th})	5,05,602	26,830,671	2,183,133

Table 4.2: Graph statistics.

Table 4.2 illustrates the visualisation of the transaction graph in week 2 (April 1^{st} - April 7^{th}). In general, what we could interpret from the transaction graph was there would be a highly dense connected component (e.g, the connected component located in the centre in Figure 4.2) surrounded by the much sparser sub-graphs or a single node.

The graph in Figure 4.2 is visualised by Graphia [32] - one of the most efficient graph visualisation tools. This graph is the only one to be shown because drawing such a giant network requires much computational effort and memory allocation. Moreover, as Graphia can only run as a GUI application, we used it on our personal computer with a 4-core processor and 16 GB of memory. As a consequence, the week 2 transaction graph was the only graph that fits our computational resources to ensure the application runs without being crashed.

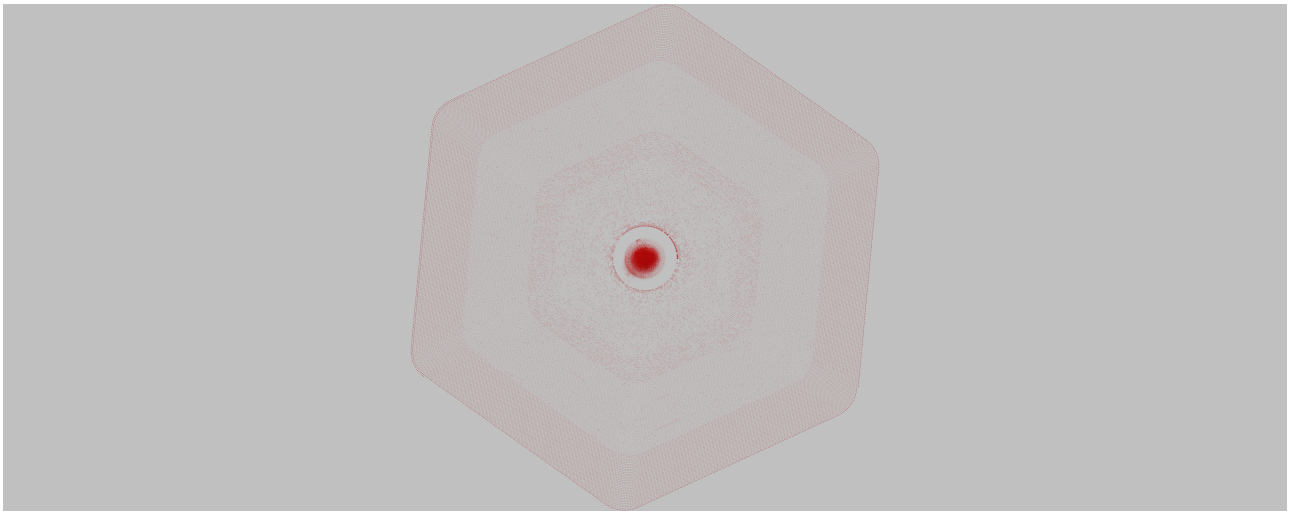


Figure 4.2: Week 2 original graph.

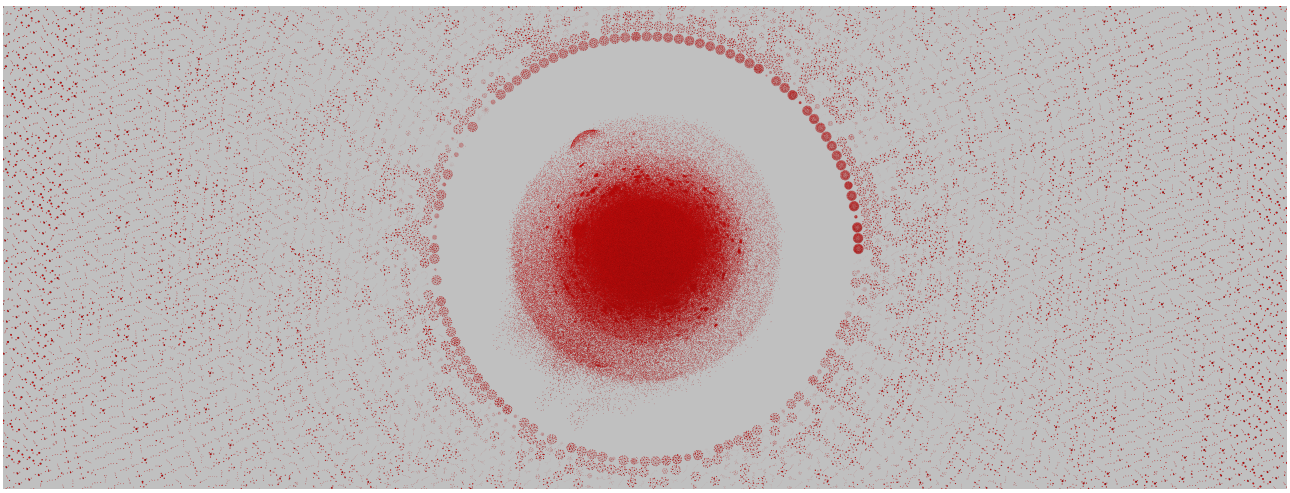
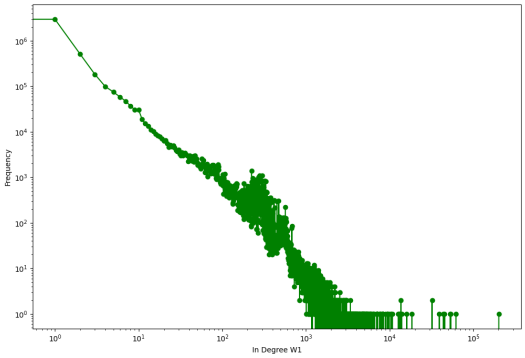


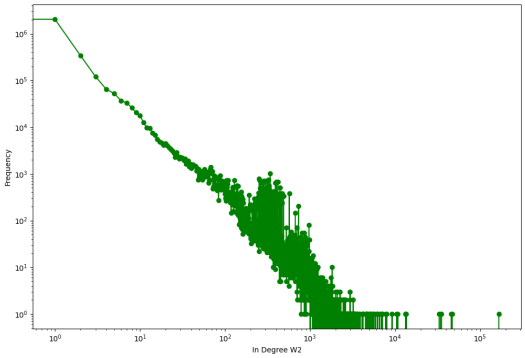
Figure 4.3: Week 2 original graph (Zoomed).

Concerning graph statistics, we extracted and drew the distributions of the In Degrees, Out Degrees, Weakly/Strongly Connected Components, Clustering Coefficients and PageRanks, which are described in the subsection below. The coding implementation resulted in these results were mentioned in Appendix A.2.

4.3.1 Degrees

In such a giant graph of each week duration, the downward trend of the number of nodes (frequency) with respect to the in/out degrees value could be observed with most (more than 80%) of the nodes having the in/out degrees higher than 10^5 while there was a node that was considered as the centre of the graph with hundreds of thousand in/out degrees (Figures 4.4, 4.5).





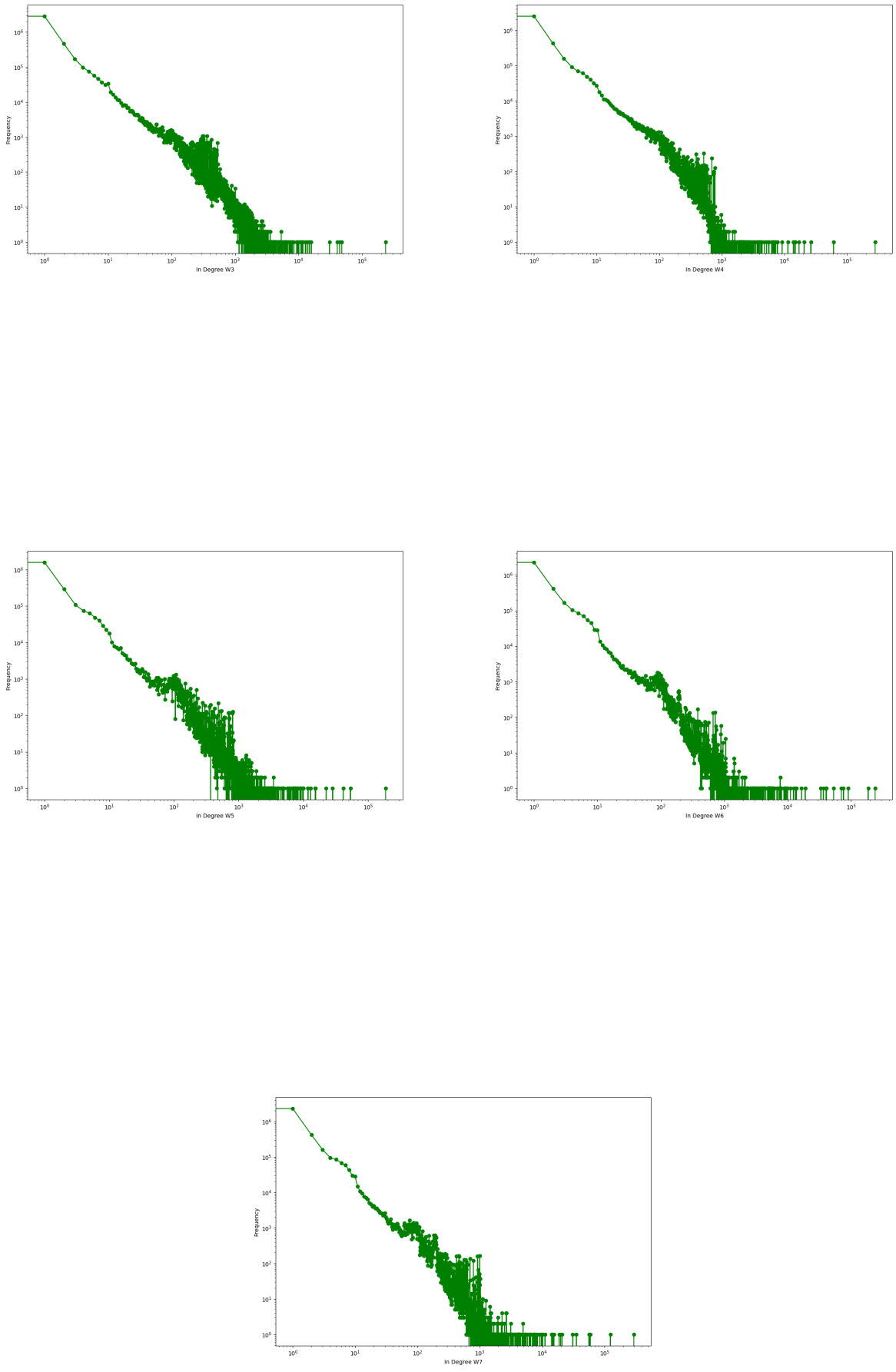
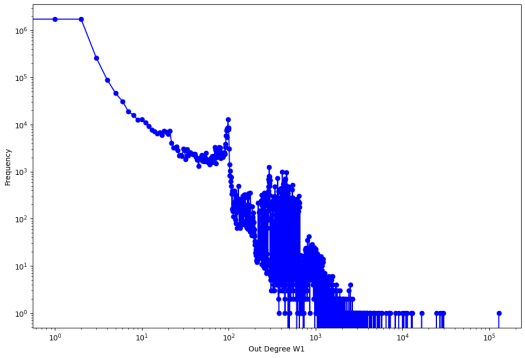
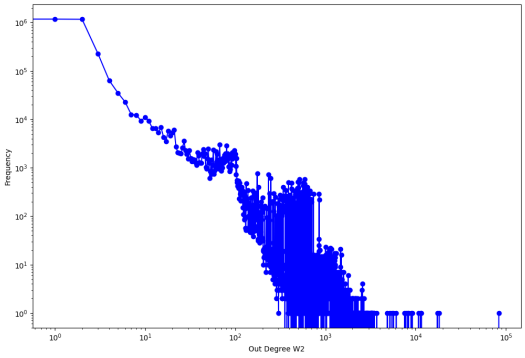
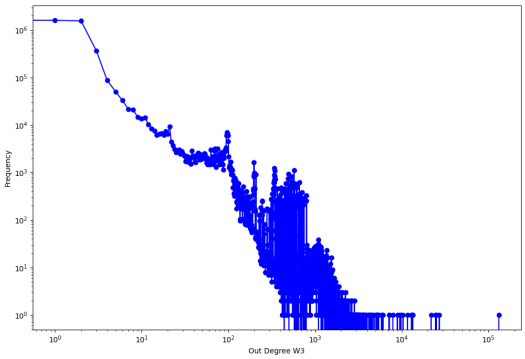
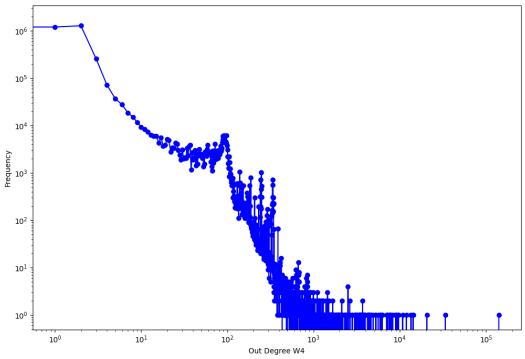


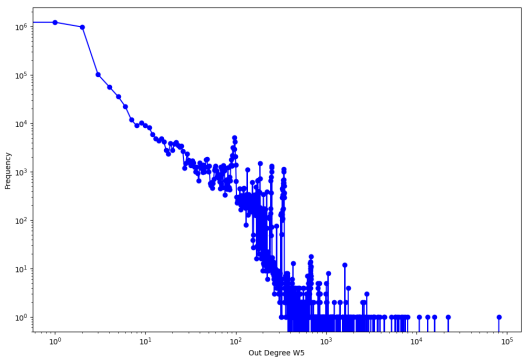
Figure 4.4: In Degrees

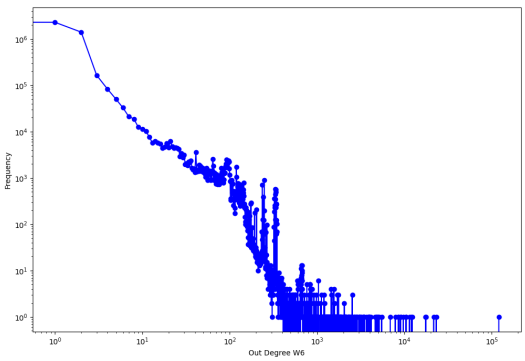

Frequency
Out Degree W1


Frequency
Out Degree W2


Frequency
Out Degree W3


Frequency
Out Degree W4


Frequency
Out Degree W5


Frequency
Out Degree W6

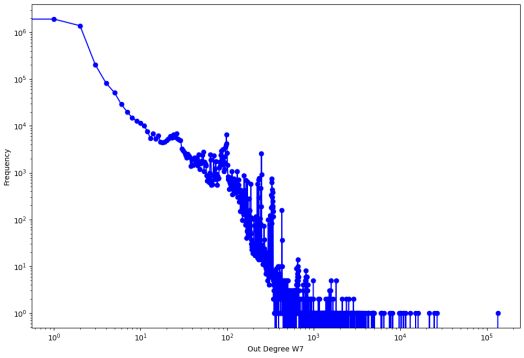
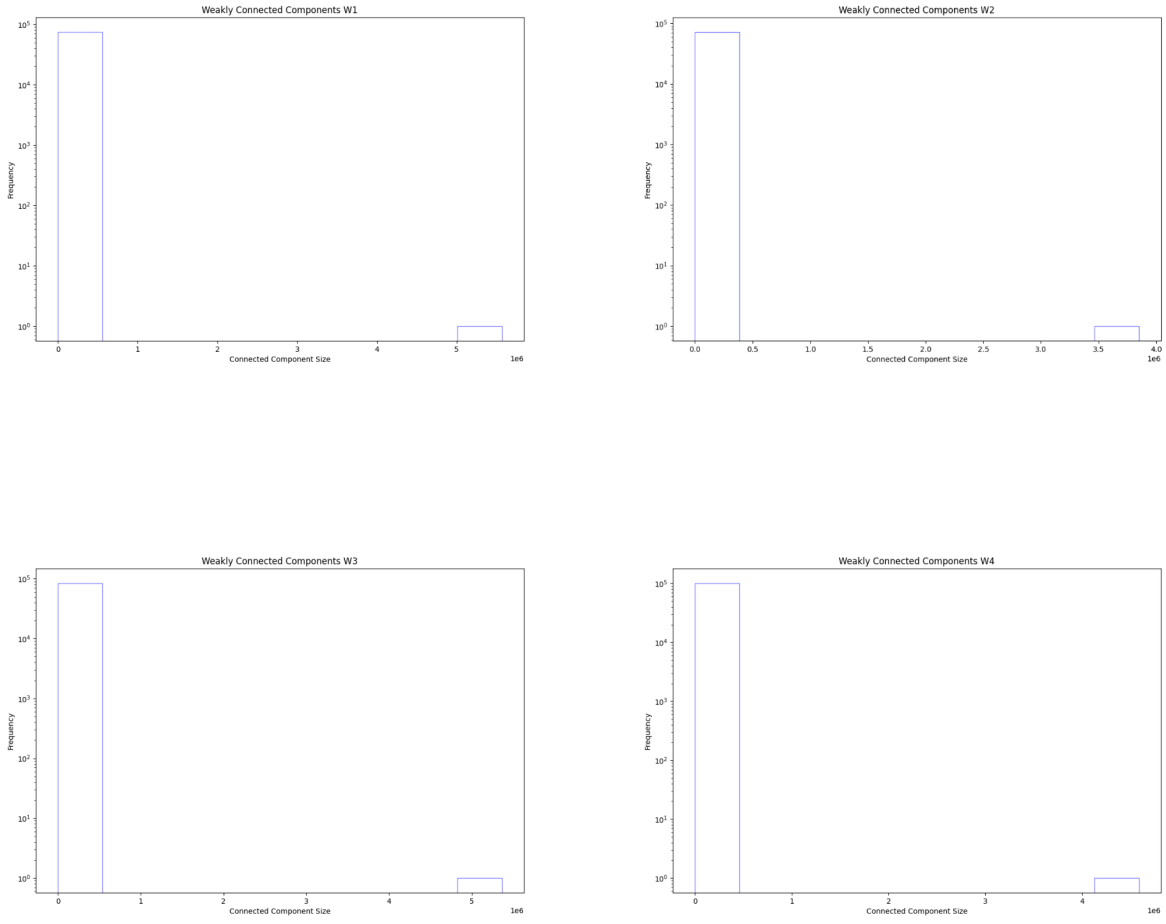


Figure 4.5: Out Degrees

4.3.2 Connected Components

Figure 4.6 and 4.7 includes the histograms of weakly/strongly connected components in all weeks. In general, it is observed for the weakly connected components in all weeks that there are nearly 10^5 components that include less than 10^6 nodes, whereas only one component has up to about 5×10^6 nodes (Figure 4.6).



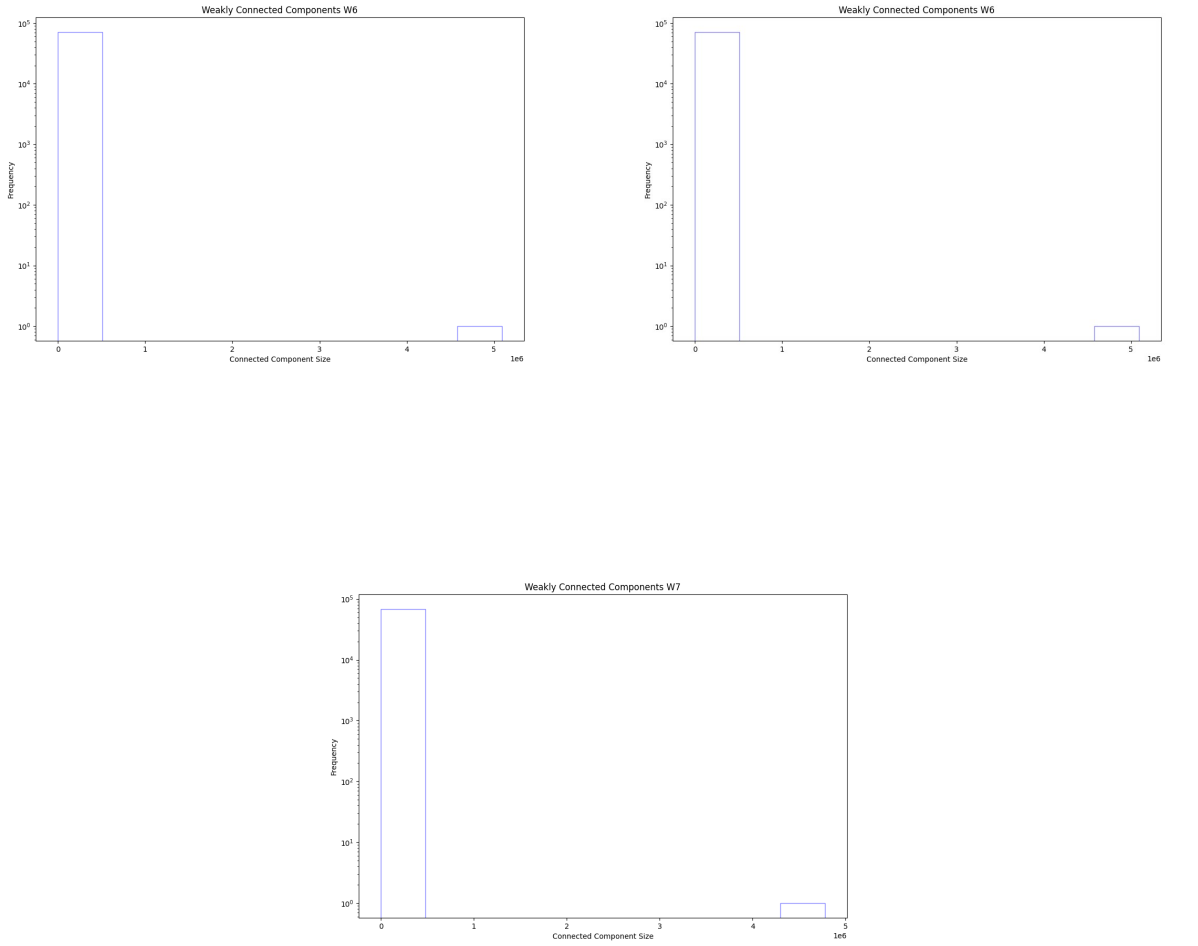
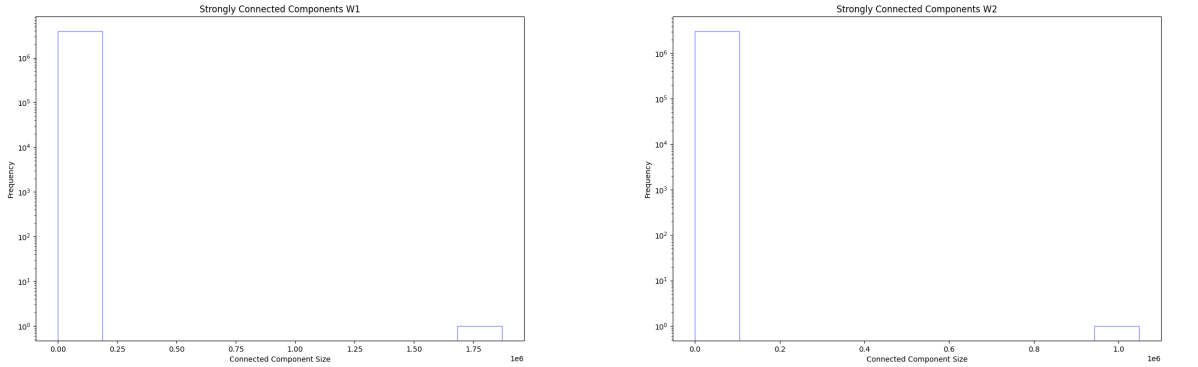


Figure 4.6: Weakly Connected Components

With the strongly connected component, the similar trend of the frequency concerning connected component size would be interpreted, but with different numbers. Overall, there are more than 10^6 connected components that have less than a quarter of 10^6 nodes while there is one connected component that includes up to more than 10^6 nodes. The higher transactions that a graph contains, the higher nodes within its largest connected component.



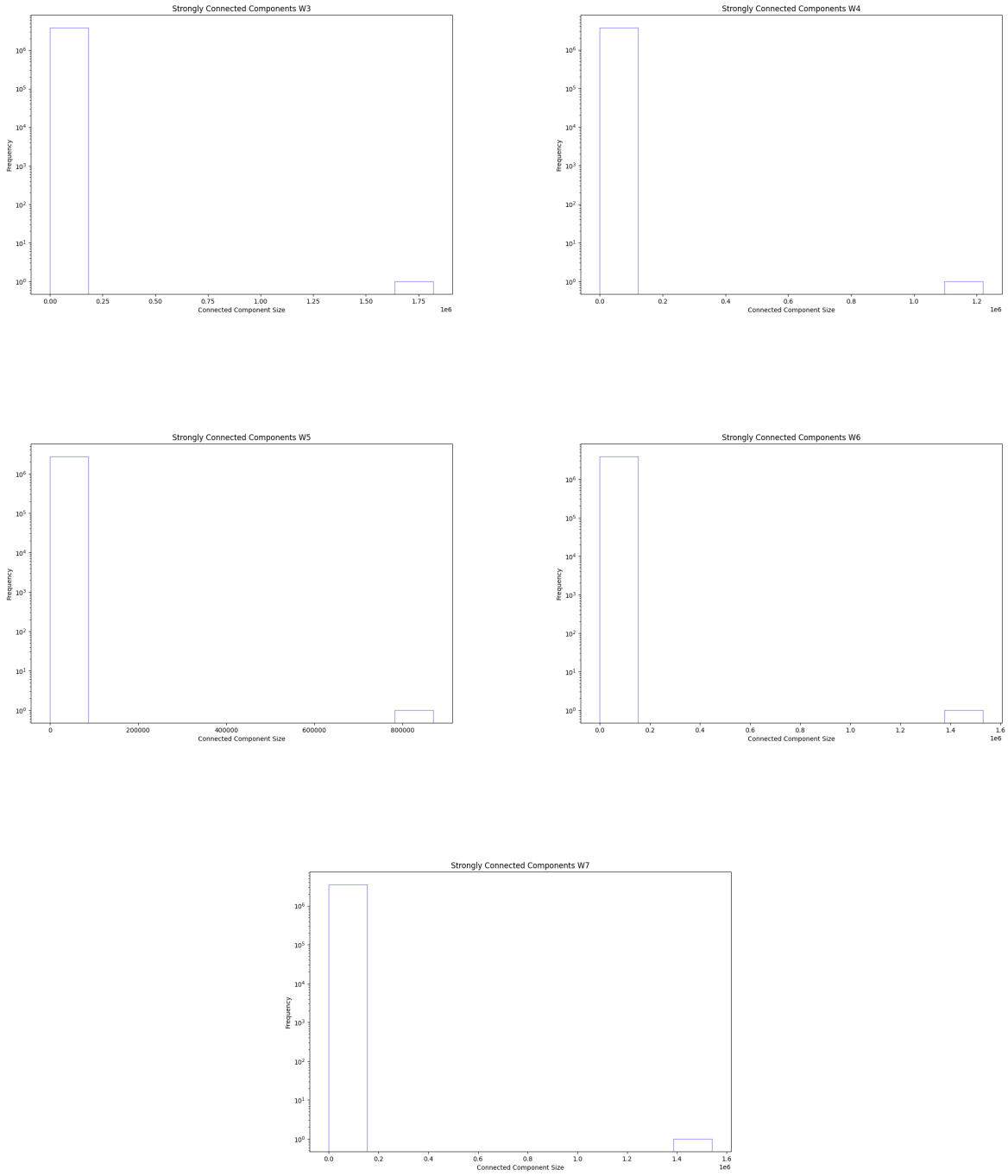


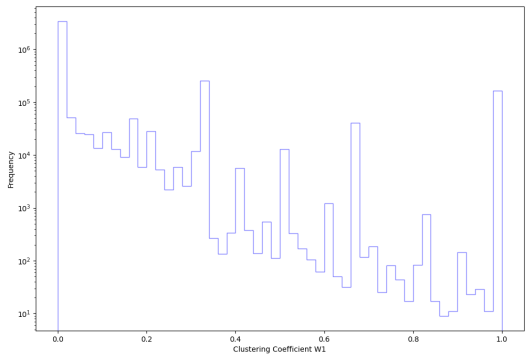
Figure 4.7: Strongly Connected Components

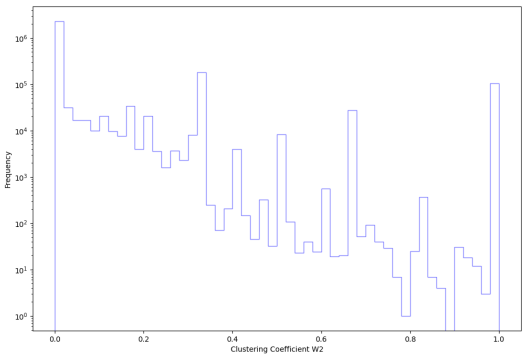
4.3.3 Clustering Coefficients

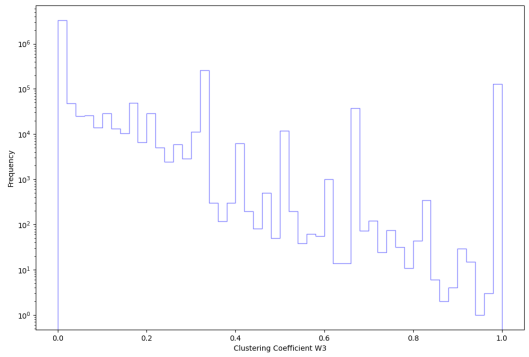
Clustering coefficient [33] of node i (C_i) represents how close its neighbours are to being a complete graph, $C_i \in [0,1]$. If the neighbourhood of i is fully connected, then the clustering coefficient is 1. Otherwise, the value of C_i close to or equal to 0 tells that there hardly/no any connections in the neighbourhood of node i.

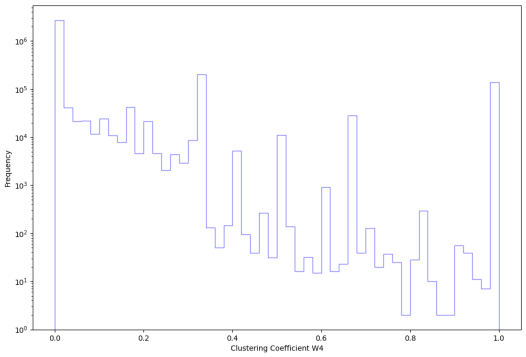
Figure 4.8 depicts clustering coefficients frequency of all nodes in each weekly transaction graph. Generally, What the histograms offer is even though a downward trend in the frequency with respect

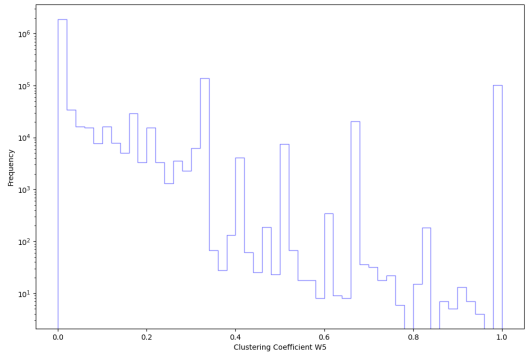
to the clustering coefficient in each transaction graph could be observed, the clustering coefficient with values of either 0.0, 0.3, 0.7 and 1.0 appear mostly in each transaction graph with the frequency from around 10^4 (for $C_i = 0.3$) to more than 10^6 (for $C_i = 0.0$).

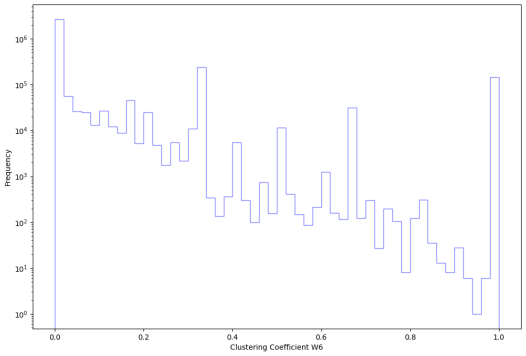












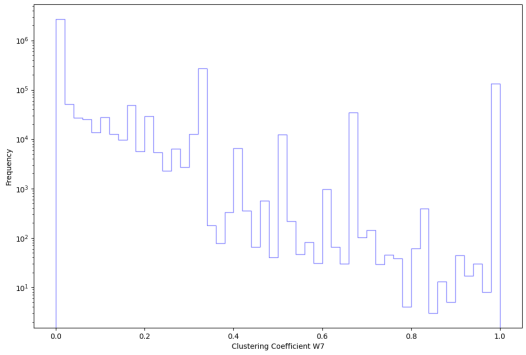
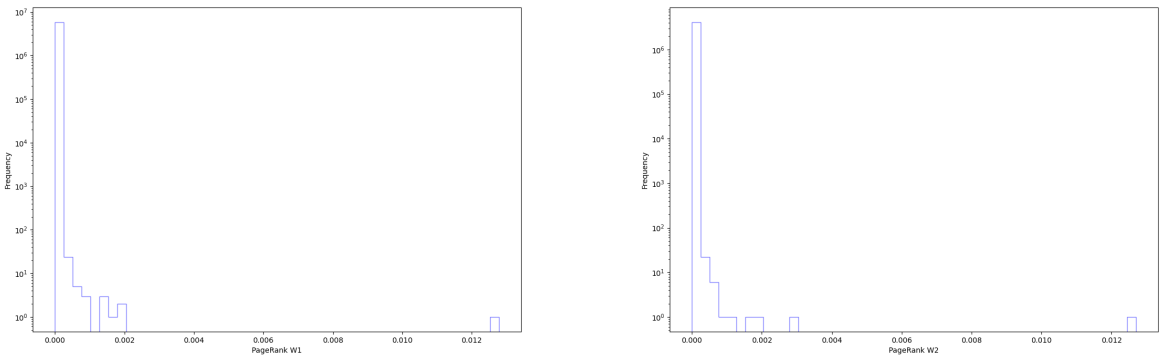


Figure 4.8: Clustering Coefficients

4.3.4 PageRanks

PageRank (PR) [34] is an algorithm used by Google Search to rank the web pages resulting in its search engine. The algorithm returns a probability distribution that illustrates the likelihood that a person could click on links that would arrive at any particular page. Therefore, PR has a value between 0 and 1.

In Graph Theory, the PR value of a node represents its importance within a network. Figure 4.9 includes the histograms of PR distribution in each weekly transaction graph. In each graph, mostly all of the nodes have a PR value less than 0.006, especially for the PR value between 0.0 - 0.001, there contain more than 10^6 nodes.



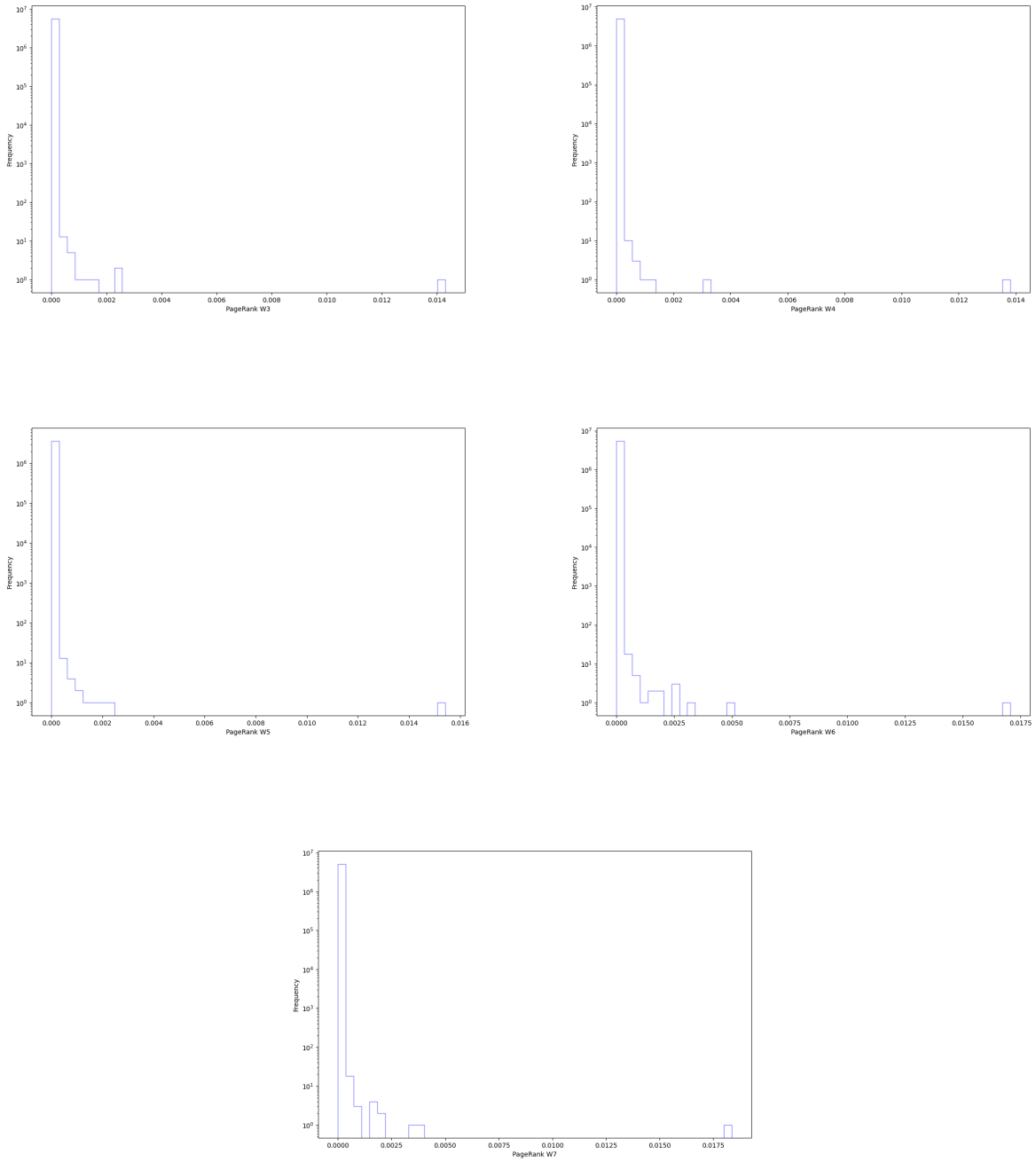


Figure 4.9: PageRanks

Time Duration	Node with highest PageRank	Value
Week 1 (March 24^{th} - March 31^{st})		0.028468823908
Week 2 (April 1^{st} - April 7^{th})		0.030221649974
Week 3 (April 8^{th} - April 14^{th})		0.031947221756
Week 4 (April 15^{th} - April 21^{st})	1NDyJtNTjmwk5xPNhjgAMu4HDHigtobu1s	0.036513643030
Week 5 (April 22^{nd} - April 28^{th})		0.036513643030
Week 6 (April 29^{th} - May 5^{th})		0.034731498170
Week 7 (May 6^{th} - May 13^{th})		0.037921098429

Table 4.3: Highest PageRank on each weekly transaction graph.

There is a common feature in all weekly transaction graph is that there is a single node with a significant higher PR value (more than 0.02) compared to all the other nodes. By sorting the PR in descendent order, we could index the node that has the highest PR value in all graphs has the same Bitcoin address, which is `1NDyJtNTjmwk5xPNhjgAMu4HDHigtobu1s` (Table 4.3). That address is also located in the largest weakly/strongly connected component in each weekly transaction graph. To get the Bitcoin address information, by referencing the API provided by Blockchain.com [31], there is an interesting finding that the identified address has conducted 1,173,096 transactions in the whole Blockchain history with the confirmed balance of 35267.06649700 BTC (counted at 23:00 pm November 4^{th} AEST).

Chapter 5

Node Embeddings Visualisation

This chapter describes the input node features, the method to train large graphs via stochastic minibatch training as well as the progress of using Graph Neural Networks to generate and visualise the node embeddings.

5.1 Input node features

As is discussed in the Background (section 2.5), the input of the GNN requires the graph structure and the node features. For the graph structure, we have generated seven graph objects with respect to each week duration (mentioned in Section 4).
For the input node features, we could consider it as a vector of Bitcoin address characteristics (e.g., number of transactions, confirmed balanced). However, within the scope of the weekly transaction graph, to get those values, for each address node, we need to iterate all the transactions to count the transactions they conducted and compute their balance). Besides, the APIs to get address information provided by Blockchain.com [31] or Sochain [15] gives the transactions count or the confirmed balance in the whole Blockchain history instead of the predefined block range.
As a result, we would extract the node features based on the graph structure, including the in/out degrees of a node, or its PageRank value. The reason why we chose the node degrees or PageRank was by referencing the paper [35], it was shown that the graph with the PageRank or node features outperforms the others ones (e.g., random, one-hot, shared) in a different dataset. Moreover, the accuracy of the GNN model with mean aggregator and degree node features surpasses the model with the same configuration but with the real features (being collected manually) in the MUTAG [36] dataset (Table 3, [35]). In addition, we would compare it with the feature of constant value (e.g., 1-dimension vector with the value of 1 for all nodes).
To sum up, there are four main node features that are used as the input to the GNN, including Constant, In/Out Degrees and PageRank. The implementation for extracting the features are mentioned as a part of Appendix A.3.

5.2 Stochastic training on graphs

5.2.1 Overview

Because we would deal with massive graphs of millions of nodes or edges, the full-graph training (where the whole graph, as well as all its node features, can fit into GPU memory) could not work. For instance, concerning a graph with N nodes is passed through a K-layer Graph Convolutional Networks with the hidden state size H. Processing the intermediate hidden state requires $O(NLH)$ memory, which would exceed the GPU memory resources with large N. This section discusses an approach of performing stochastic minibatch training via neighbourhood sampling, where instead of fitting the features of all nodes into GPU, we would fit the features of sub-graph (minibatch) only.

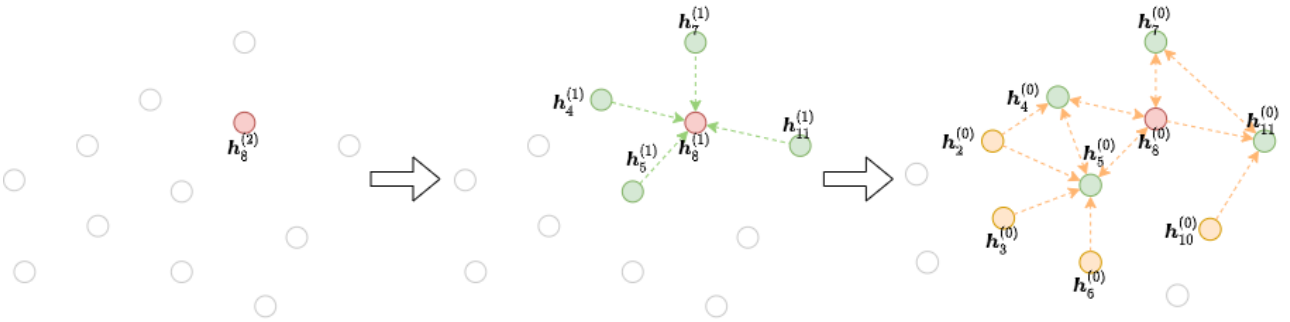


Figure 5.1: Neighbourhood sampling approach [8].

With the neighbourhood sampling approach (Figure 5.1), for each gradient descent step, instead of computing the node features of all nodes, we would take a minibatch of nodes whose features are to be computed at K-th layer. We then select **some** or **all** their neighbours at K-1 layer. This would continue until the input node features are reached (node features at layer 0). This repeated process forms the subgraph starting from the output nodes at K-th layer to the input nodes at layer 0.
With this method, the memory requirements would be significantly reduced when training a GNN on a large graph.

5.2.2 DGL implementation

DGL provides several neighbourhood sampler classes. `MultiLayerFullNeighborSampler` [37] is the most simple one that gathers the messages from **all** the neighbours of a node. To use a DGL `sampler`, we need to integrate it with the `NodeDataLoader` (Listing 2), which iterates over all the nodes in the minibatches [37].

Listing 2: Neighbourhood sampler and dataloader [38].

```
1 sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
2
```

```
dataloader = dgl.dataloading.NodeDataLoader(
    g,
    train_nids,
    sampler,
    batch_size=1024)
```

In Listing 2, the `sampler` is declared with 2-layer GNN to be sampled and the `dataloader` is initialised with the DGL graph object `g` [39], node ids to be trained, the predefined sampler above, size of the minibatch (`batch_size`).
Looping through the DataLoader (Listing 3) will return a list of three items at a time. `input_nodes` represents the nodes that are needed to compute the features of the `output_nodes` while `blocks` specifies which nodes are considered as input, which node representations are computed as output in each GNN layer.

Listing 3: Dataloader iteration [38].

```
input_nodes, output_nodes, blocks = next(iter(dataloader))
print(blocks)
```

5.3 Stochastic 2-layer GraphSage

Listing 4: Stochastic 2-layer GraphSage.

```
class StochasticTwoLayerGCN(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.sage1 = dglnn.SAGEConv(in_features, hidden_features, 'mean')
        self.sage2 = dglnn.SAGEConv(hidden_features, out_features, 'mean')

    def forward(self, blocks, x):
        x = F.relu(self.sage1(blocks[0], x))
        x = self.sage2(blocks[1], x)
        return x
```

Listing 4 defines 2-layer GraphSage model with DGL. Each layer (either `self.sage1` or `self.sage2`) is called with the `dglnn.SAGEConv()` function specified with the input, output features and the `mean` agrregator method.
To train a minibatch, instead of fitting the whole DGL graph object, we would pass the `blocks` resulting by the `dataloader` in Listing 3. For any GNN layers excepts the last one, the non-linearity activation function (e.g., ReLU) would be applied.

5.4 Embedding visualisation pipeline

To generate the node embeddings, we would initially create the DGL dataset based on the graph structure and the node features resulting from section 4.3 and 5.1 respectively (Detailed code implementation is referred in Appendix A.3). Following, the dataset would be passed through a two-layer GraphSage model with a mean aggregator, minibatch training and multi-layer full neighbours sampler. As a consequence, for such a giant transaction graph, it would reduce the training complexity to lower the memory usage during the training progress.
Between the GraphSage model, there would be a non-linearity activation function (e.g, ReLU). The whole process was trained with 10 epochs and a learning rate of 0.01.
As a result, the node embeddings were generated by the last GraphSage model. We would then use the dimensionality reduction techniques, including PCA, t-SNE and UMAP to decrease the feature dimensions to 2 for visualisation purposes (Figure 5.2). The detailed code implementation for generating the node embeddings and its visualisation are included in Appendix A.4 and A.5 sequentially.
Sections 5.5, 5.5.2 and 5.5.3 below represents the visualisation corresponding to each Dimensionality reduction method applied for each weekly transaction graph with predefined node features.

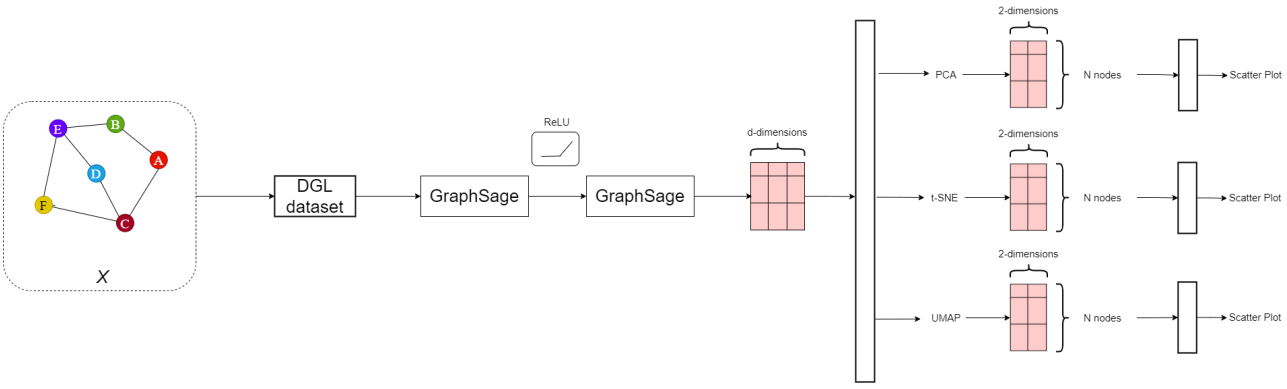


Figure 5.2: Node embeddings progress.

5.5 Embeddings visualisation

5.5.1 PCA

Constants

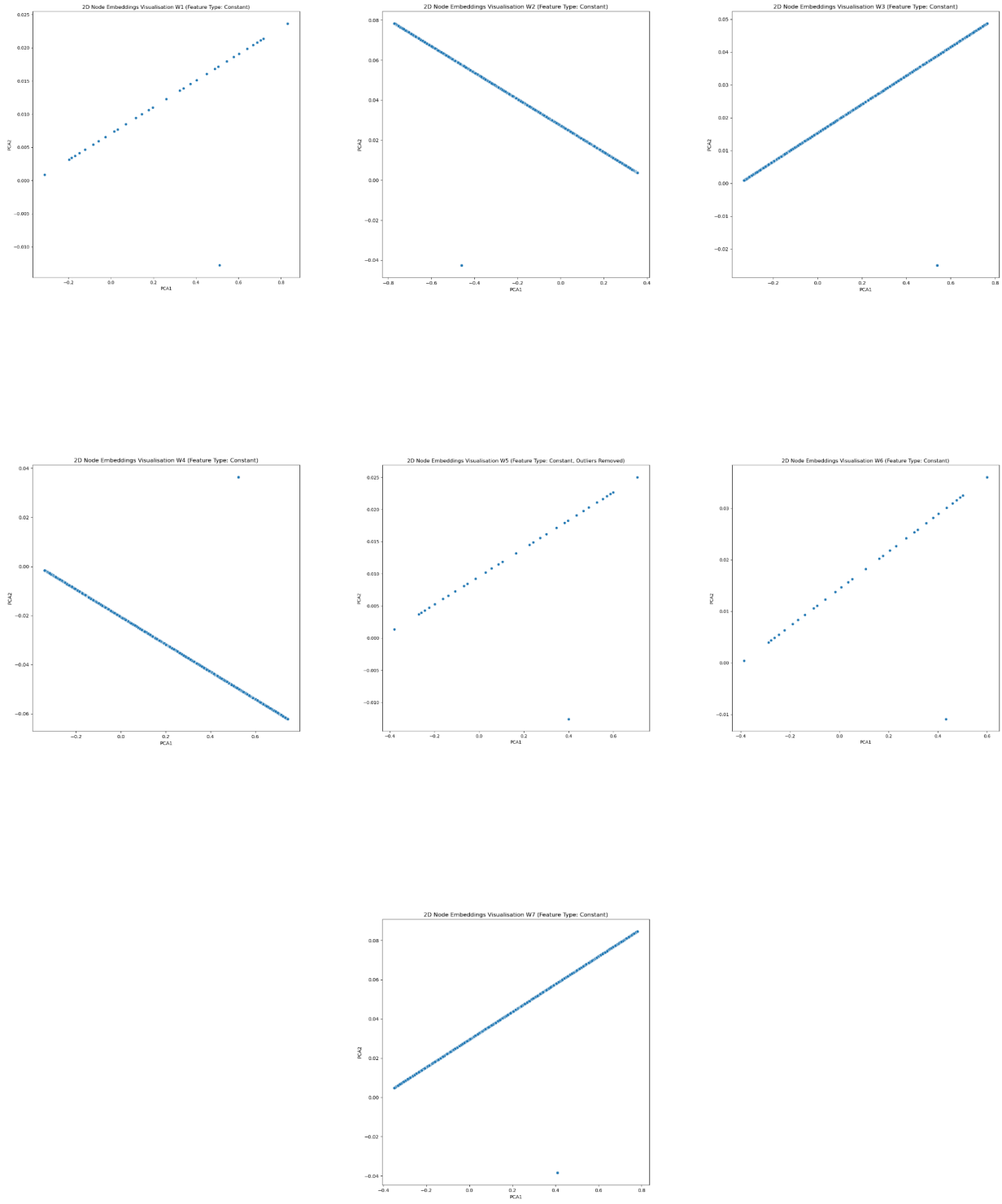


Figure 5.3: PCA Constants

In Degrees

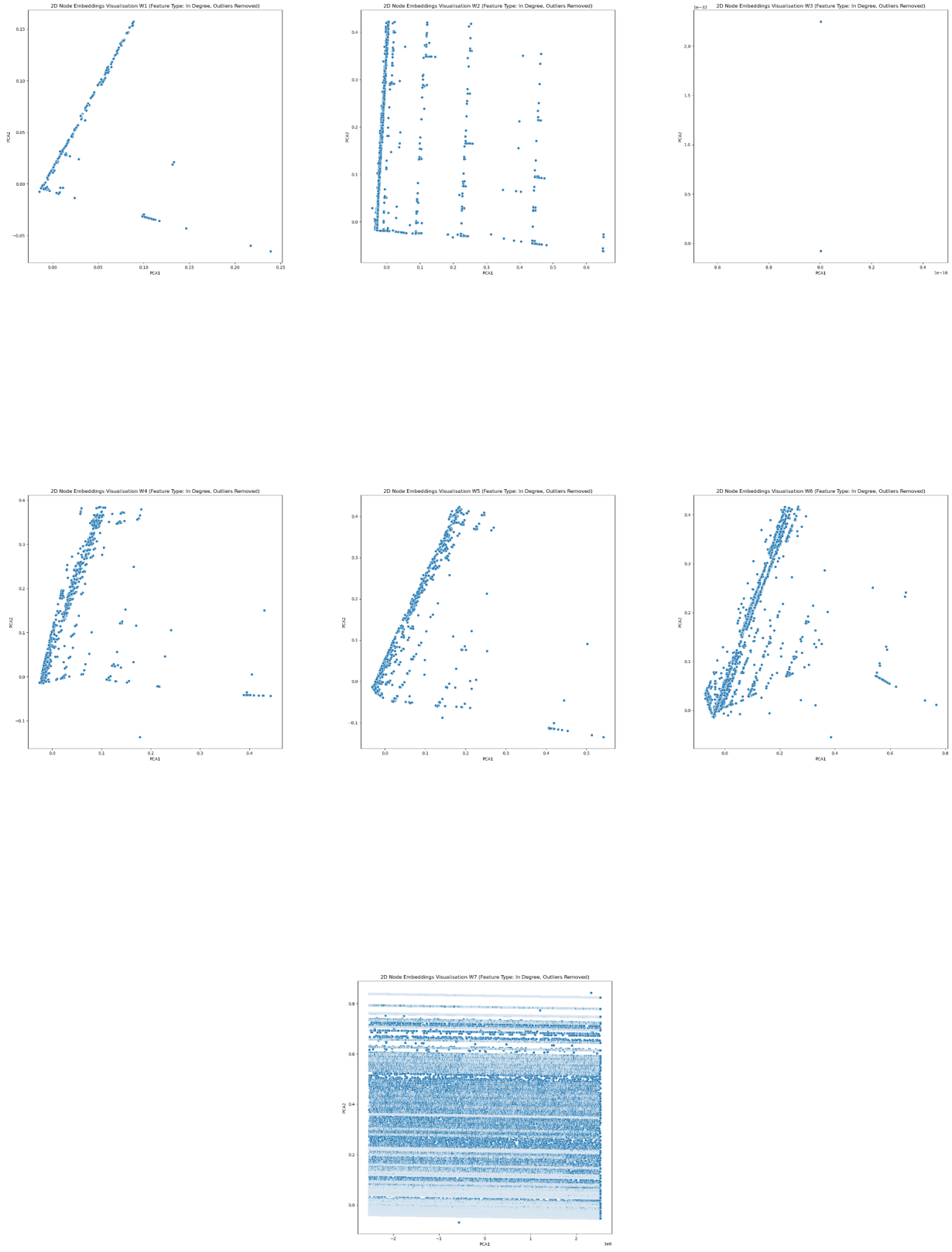


Figure 5.4: PCA In Degrees

Out Degrees

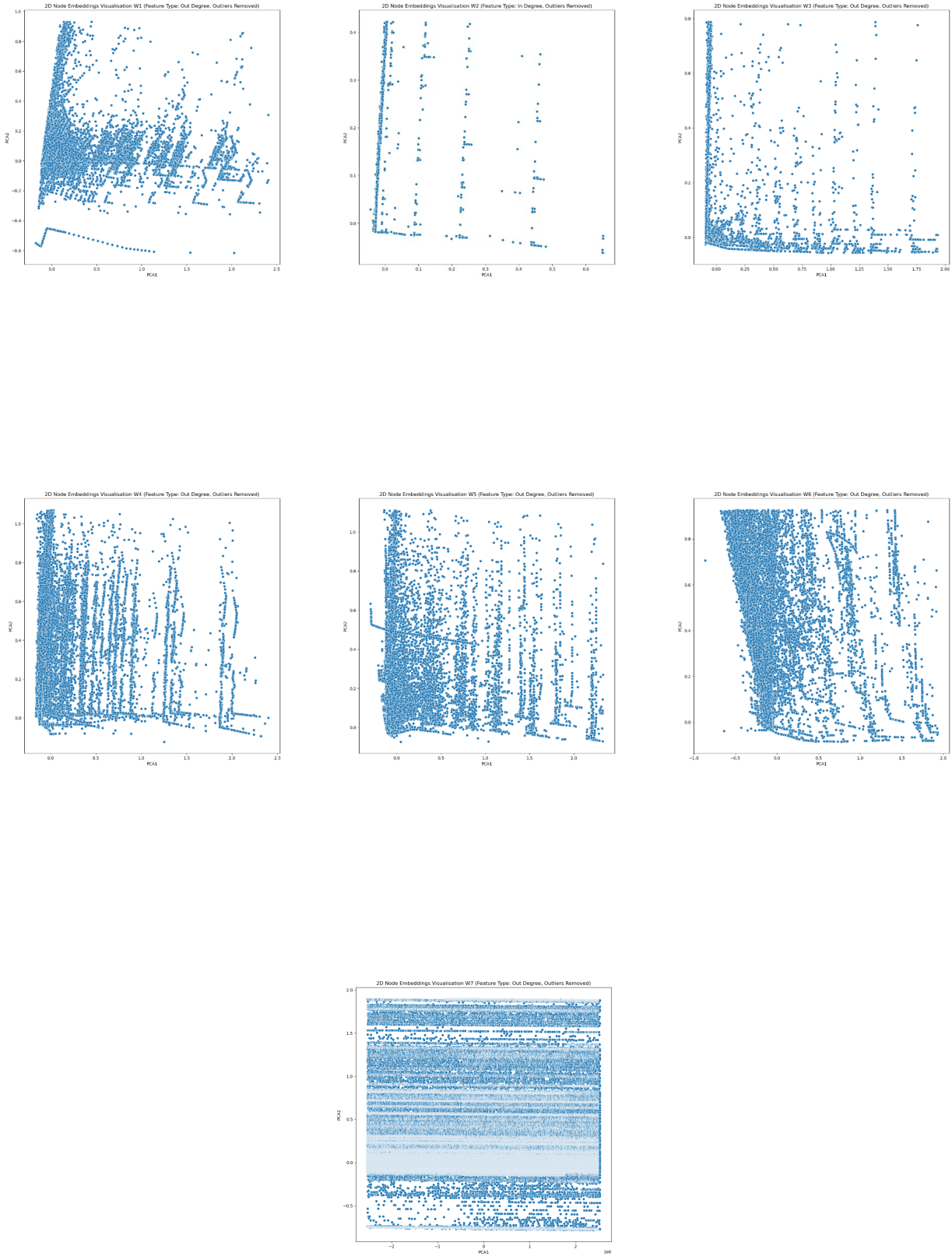


Figure 5.5: PCA Out Degrees

PageRanks

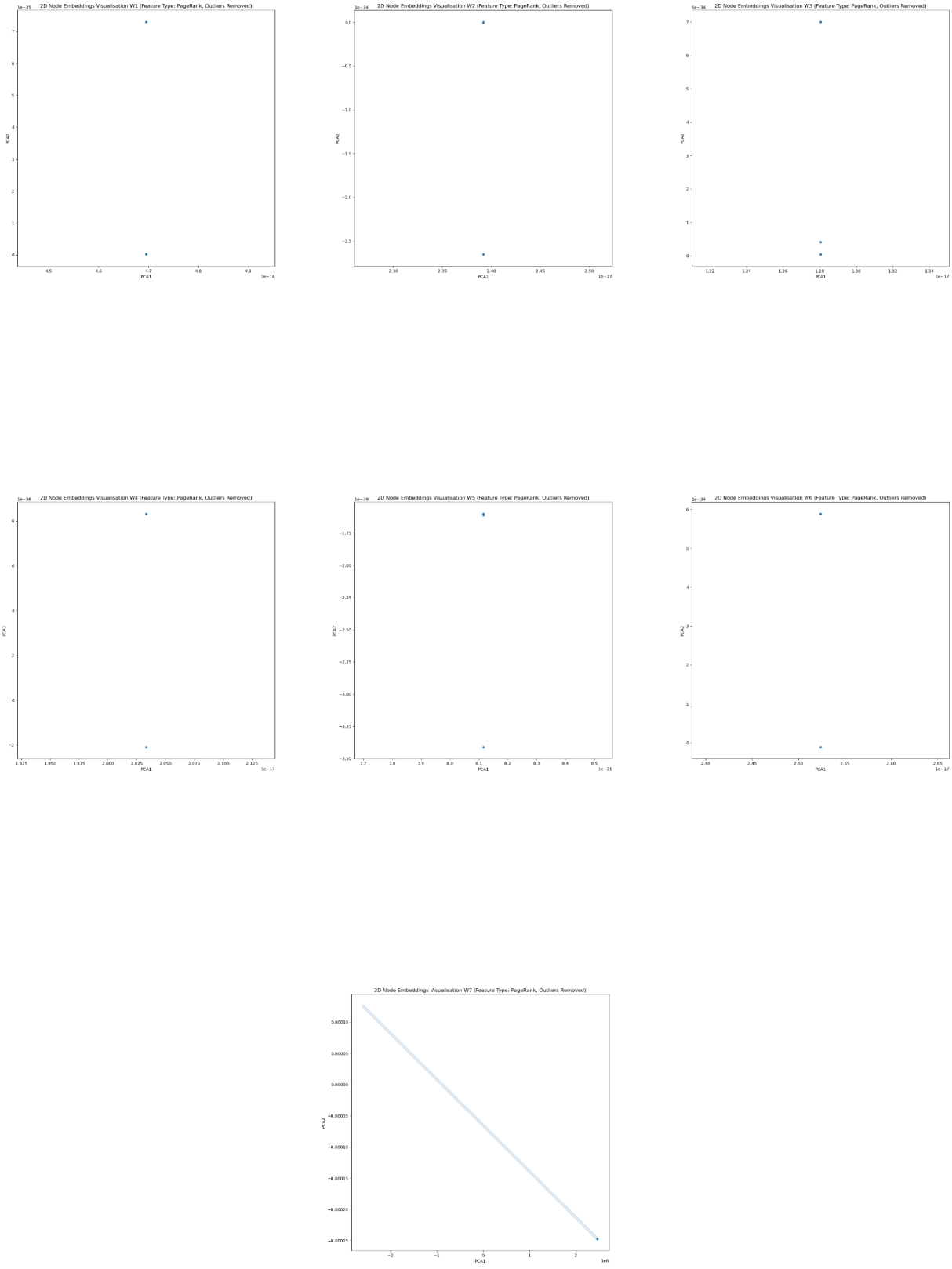


Figure 5.6: PCA PageRanks

5.5.2 t-SNE

Constants

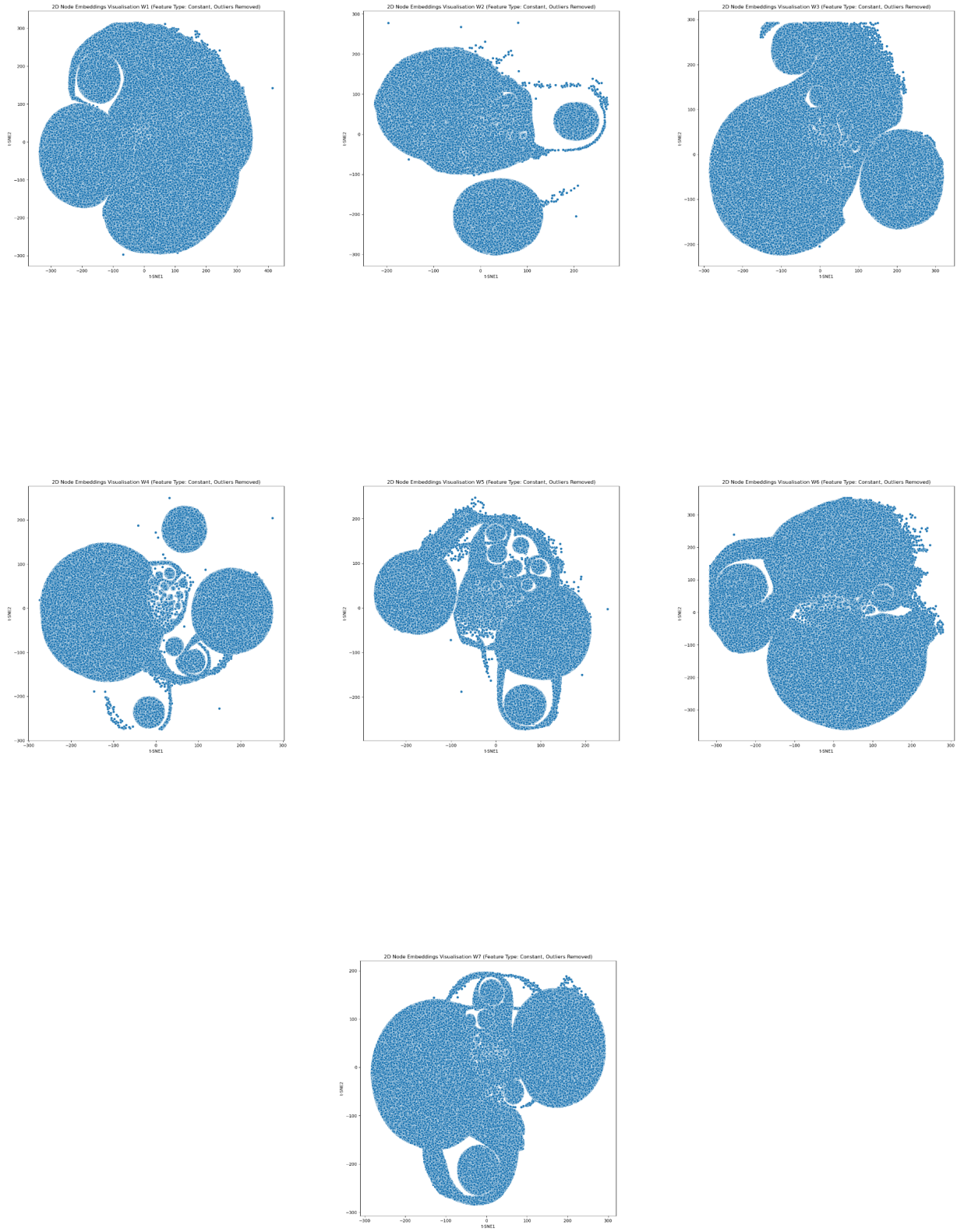


Figure 5.7: t-SNE Constants

In Degrees

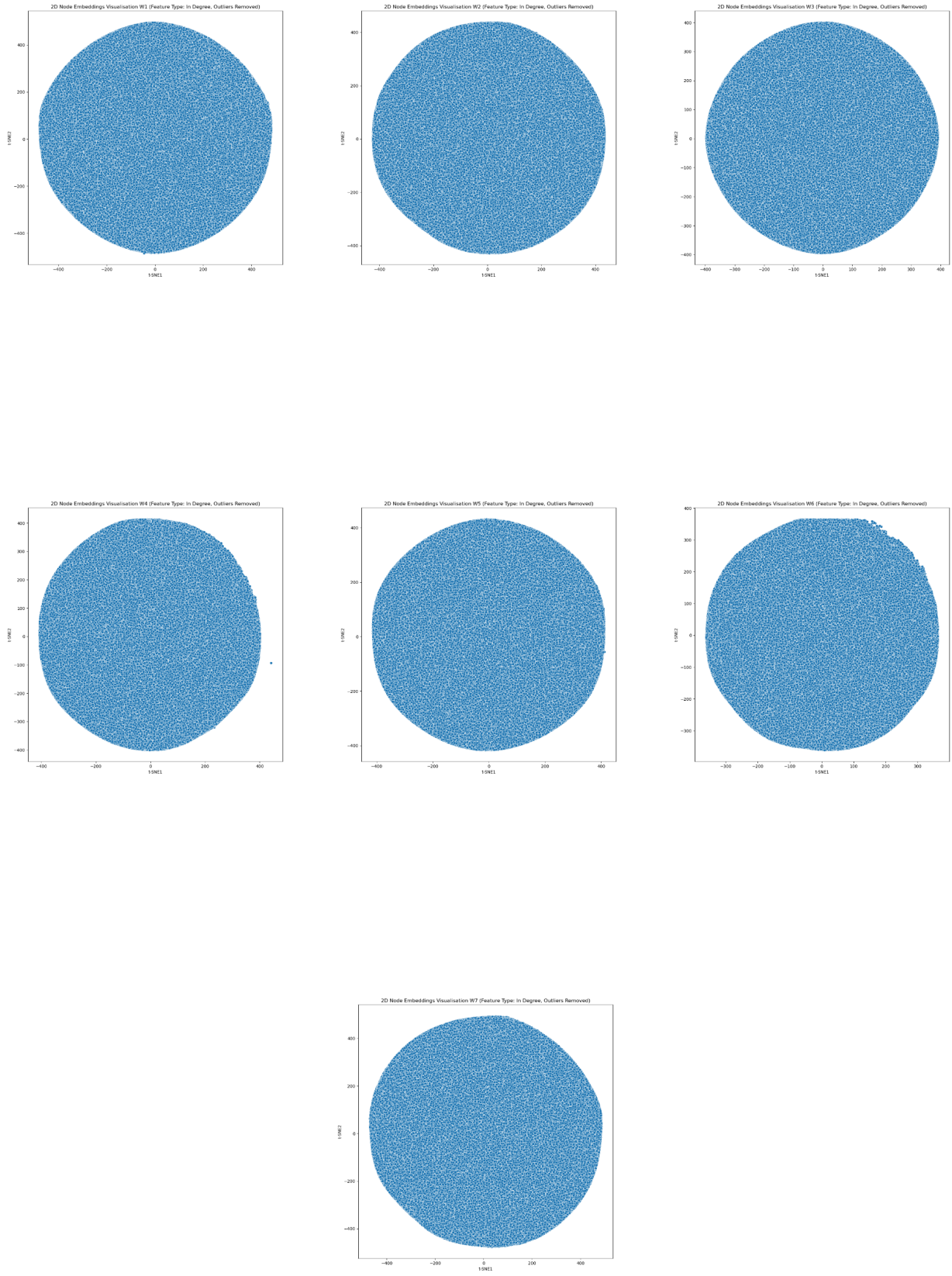


Figure 5.8: t-SNE In Degrees

Out Degrees

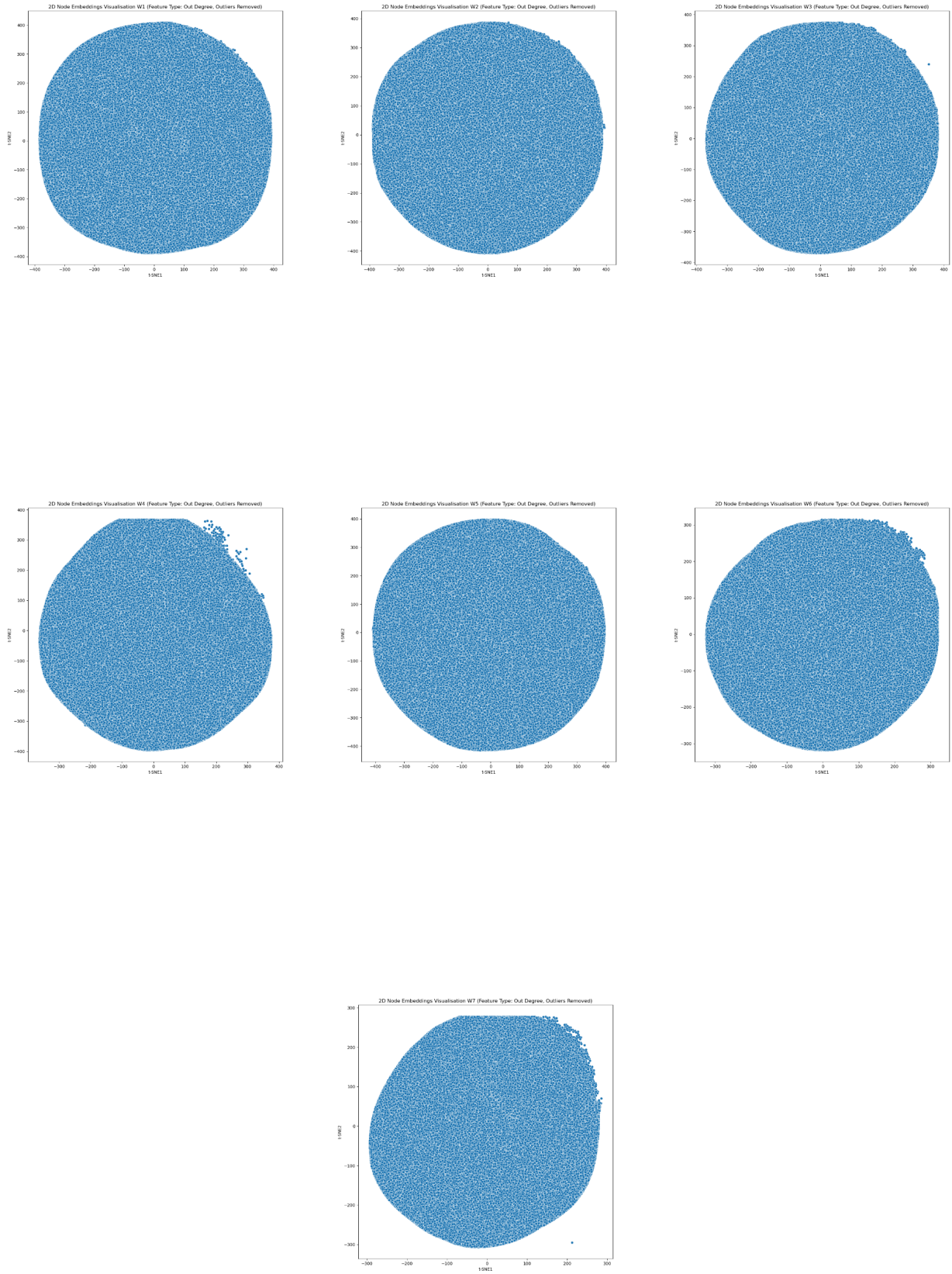


Figure 5.9: t-SNE Out Degrees

PageRanks

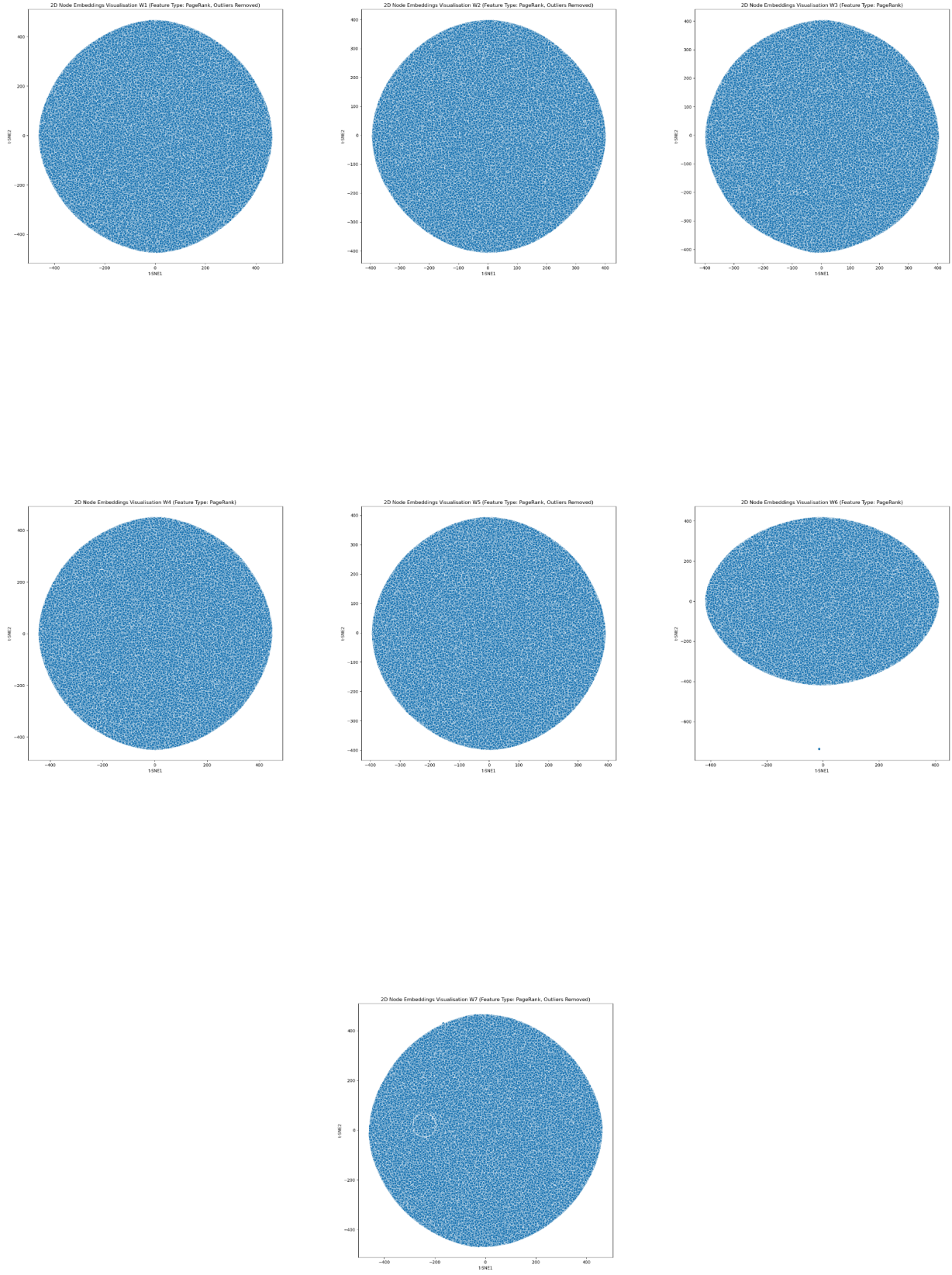


Figure 5.10: t-SNE PageRanks

5.5.3 UMAP

Constants

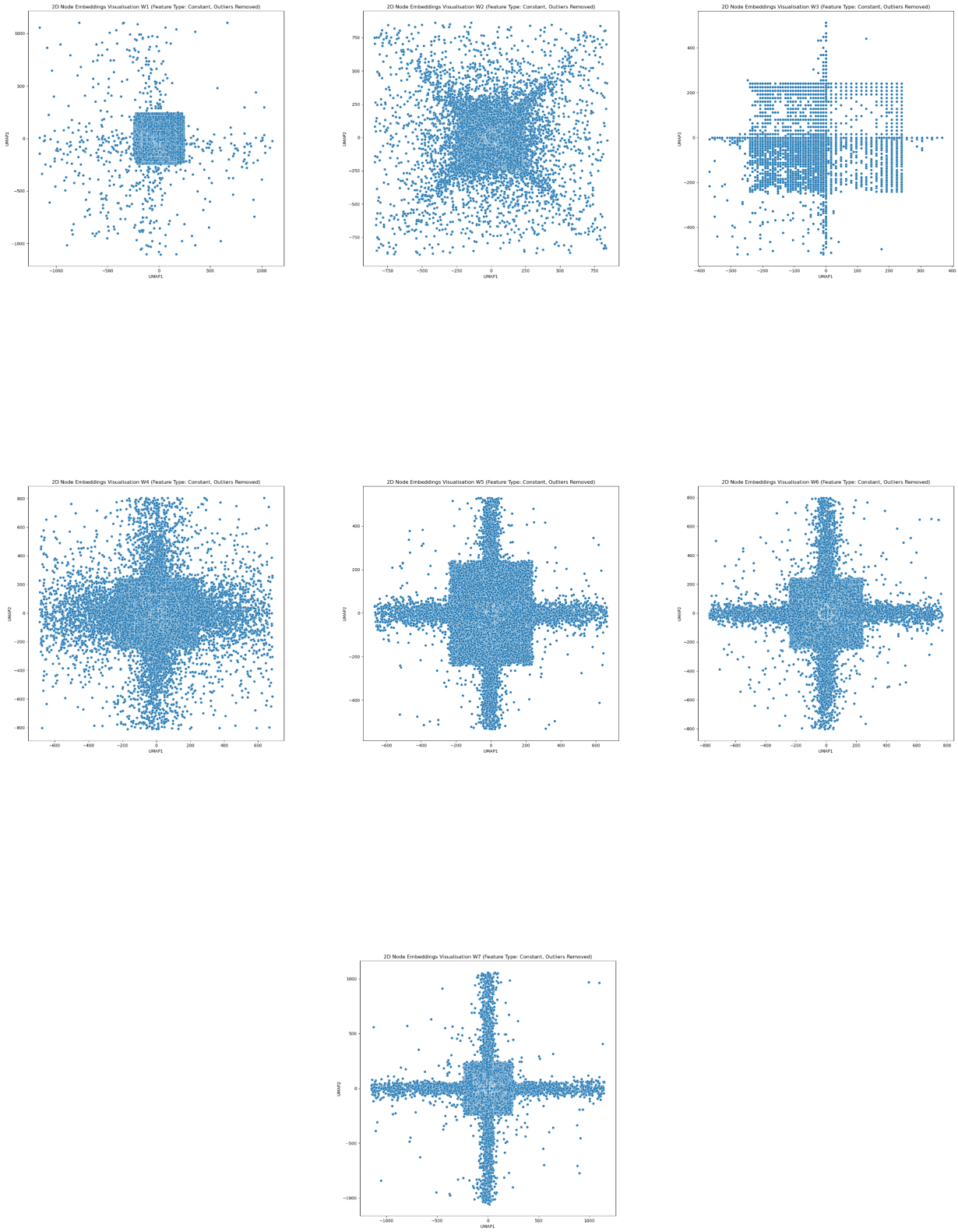


Figure 5.11: UMAP PageRanks

In Degrees

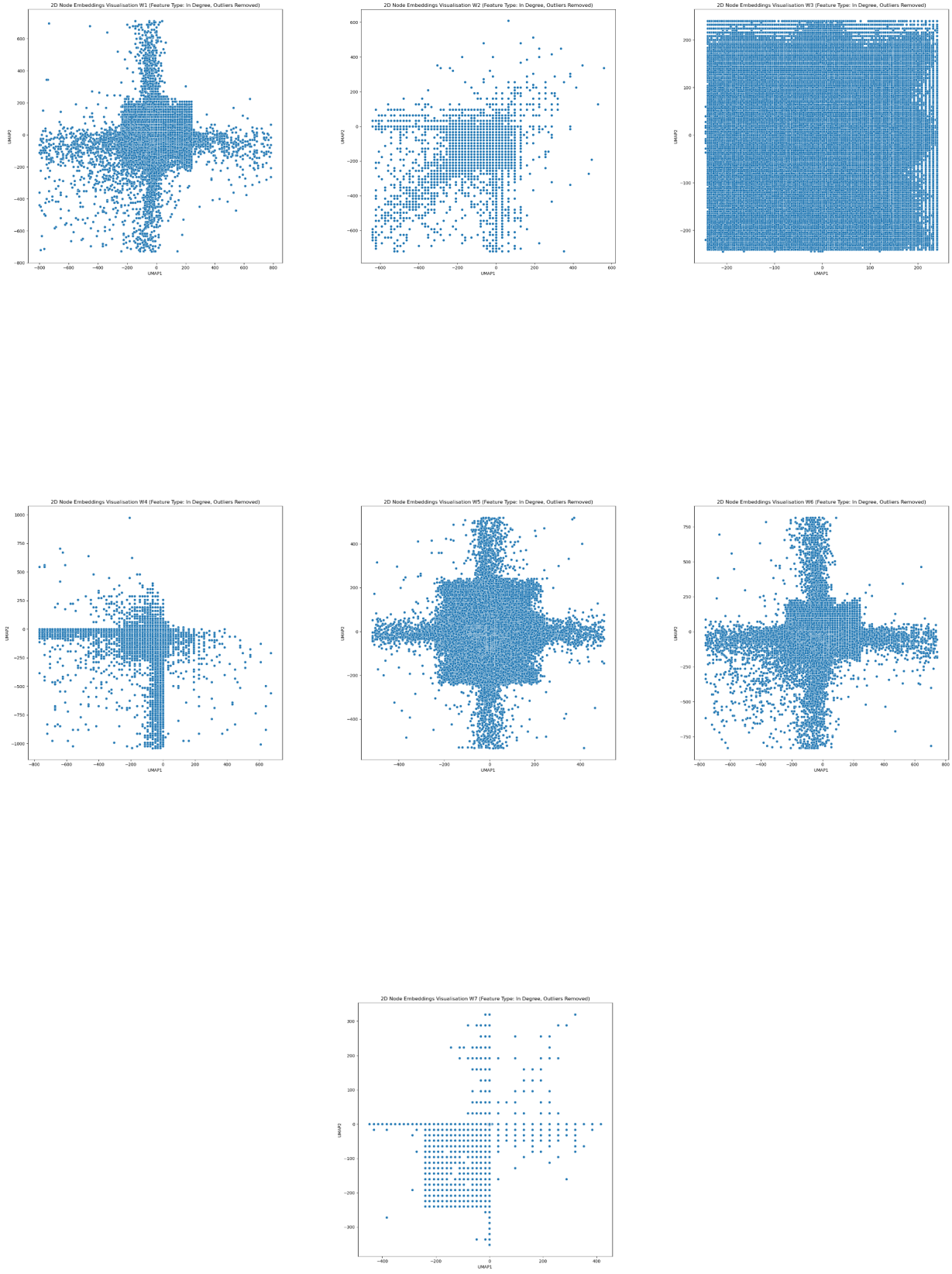


Figure 5.12: UMAP In Degrees

Out Degrees

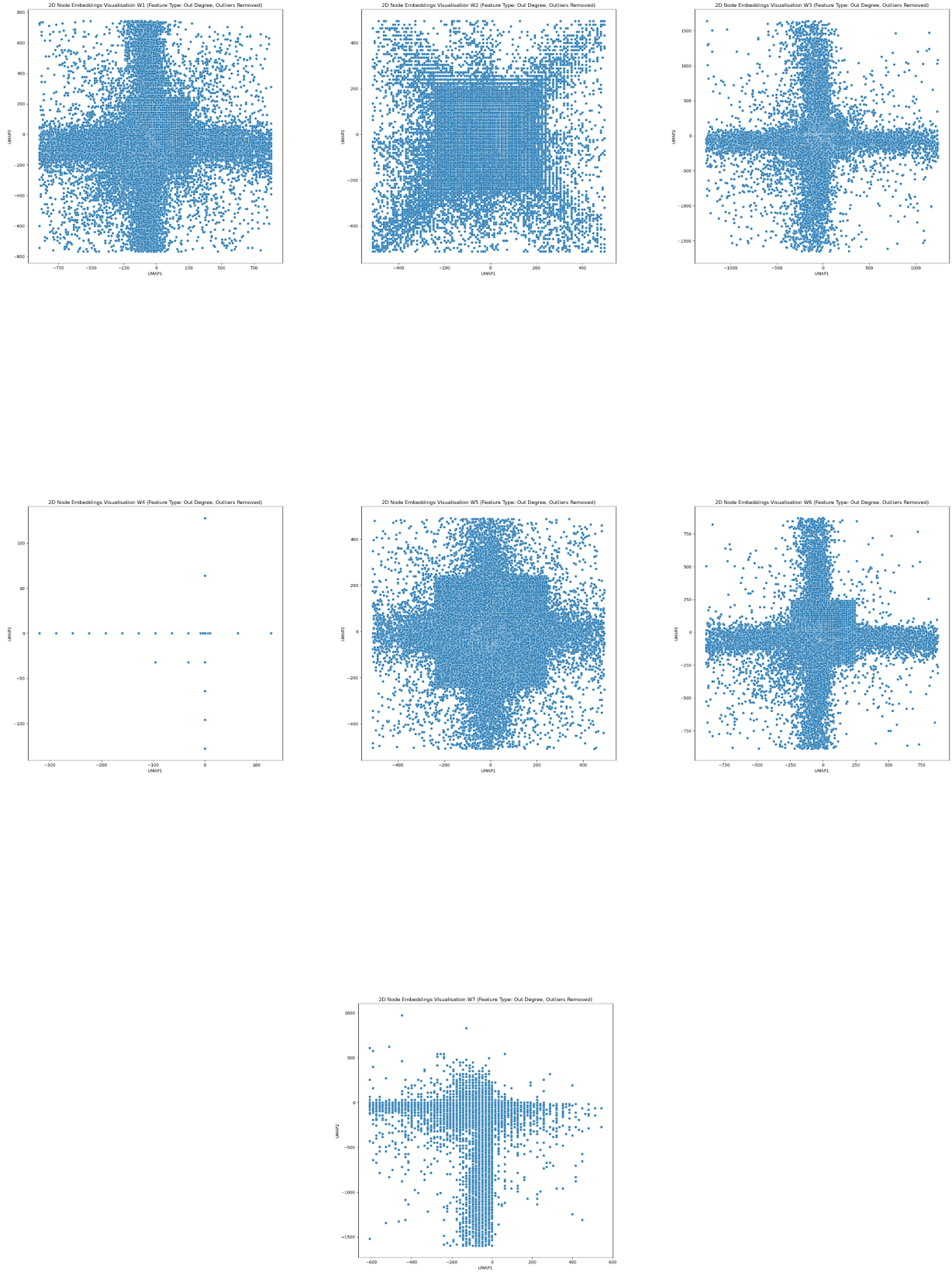


Figure 5.13: UMAP Out Degrees

PageRanks

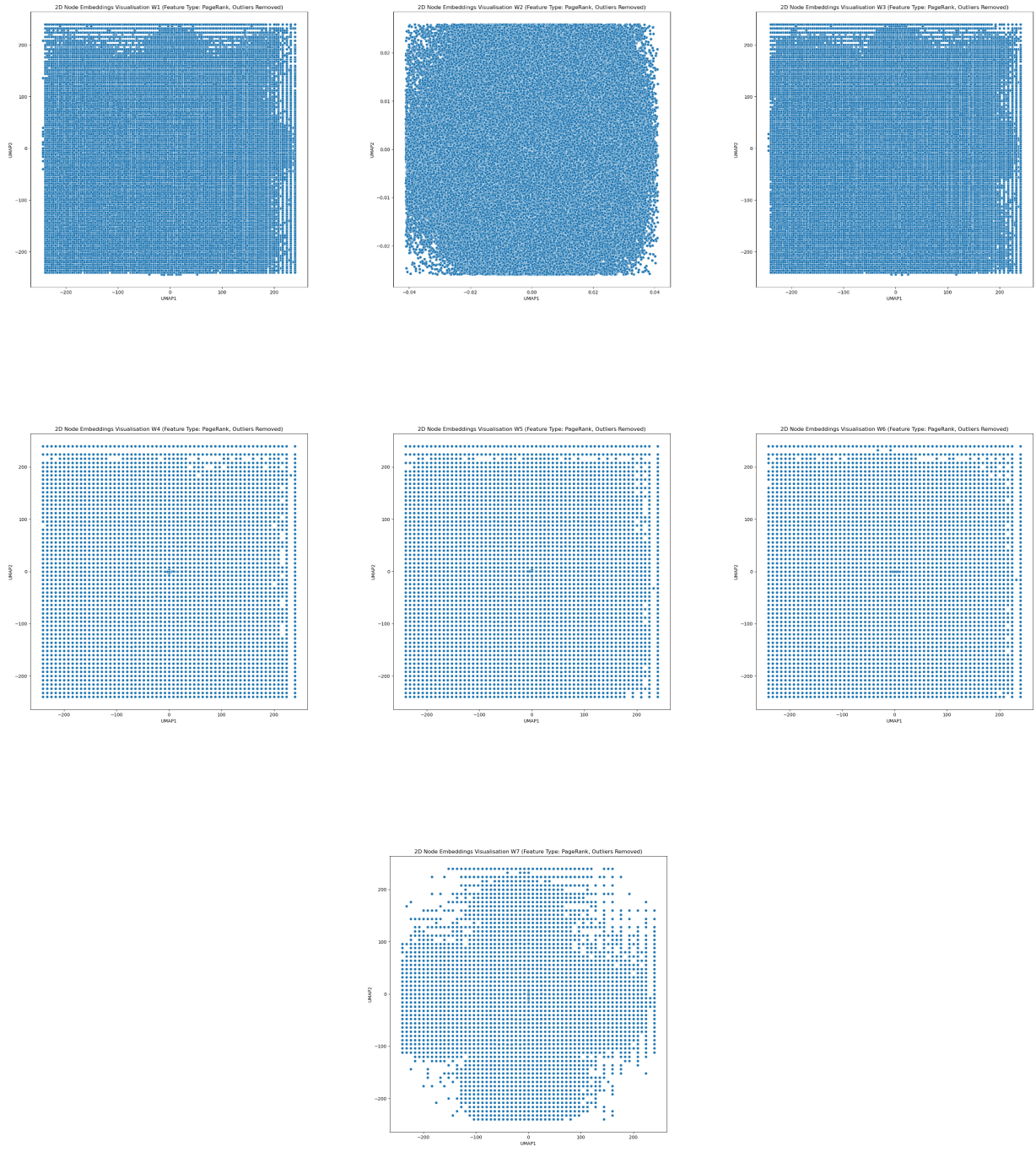


Figure 5.14: UMAP PageRanks

5.6 Discussion

Although the PCA and t-SNE did not provide expressive visualisations of the embeddings corresponding to all predetermined node features, UMAP is the dimensionality reduction technique that appeared to generate the most vivid and meaningful visualisation with the reflection of the highly dense cluster in the centred surrounded by scattering node entities.

Chapter 6

Link Prediction

This chapter details how the link prediction task of each GNN instantiation (e.g, GCN, GraphSage) was performed on the all-week transaction graph as well as its results.

6.1 Dataset in use

Instead of using each weekly transaction graph to train the link prediction model and compare the results similar to the approach of node embeddings visualisation in Chapter 5, we would perform the prediction task on the dataset of all 7 weeks, beginning from March 24^{th}. This combining graph includes more than 4,000 blocks (Table 6.1), with over 14 million transactions and millions of nodes/edges (Table 6.2).

Time Duration	Block Heights	Blocks Count
All 7 weeks (March 24^{th} - May 13^{th})	678,021 - 682,069	4,048

Table 6.1: Block heights of all weeks dataset [9]

Time Duration	#Nodes	#Edges	#Transactions
All 7 weeks (March 24^{th} - May 13^{th})	25,414,409	339,737,672	14,023,786

Table 6.2: Graph statistics.

In addition, the input node features for the following GNN models are the node's PageRank value.

6.2 Link prediction overview

A link prediction model represents the connectivity likelihood between two nodes u and v as a function $\phi(\cdot)$ of $\boldsymbol{h}_u^{(K)}$ and $\boldsymbol{h}_v^{(K)}$, their node features computed from the Kth-layer GNN.

$$y_{u,v} = \phi(\boldsymbol{h}_u^{(K)}, \boldsymbol{h}_v^{(K)})$$

where $y_{u,v}$ is the score between two nodes u and v.
Training a link prediction model is the task of learning the $\phi(\cdot)$ function to compute the connection score between two node features. Supposing that there is a link between two nodes u and v, it is expected that the score $y_{u,v}$ is larger than $y_{u,v'}$ where v' does not exist on the graph and is sampled from an arbitrary noise distribution $v' \sim P_n(v)$. Such methodology is called **negative sampling**. One of the proposed loss functions to be minimized in order to achieve the behaviour above is margin loss:

$$\mathcal{L} = \sum_{v_i \sim P_n(v), i=1,\ldots,k} \max(0, M - y_{u,v} + y_{u,v_i})$$

where M is a constant hyperparameter.

Here are the examples of using dot product to compute the scores on edges (Listing 5) and the loss function (Listing 6) implemented with DGL [5].

Listing 5: Compute the scores on edges with the dot product [40]

```
class DotPredictor(nn.Module):
    def forward(self, edge_subgraph, x):
        with edge_subgraph.local_scope():
            edge_subgraph.ndata['x'] = x
            edge_subgraph.apply_edges(
                dgl.function.u_dot_v('x', 'x', 'score'))
            return edge_subgraph.edata['score']
```

As the link prediction model works on graphs, the negative samplers as another graph contains all negative node pairs as edges is the `edge_subgraph`. This graph is passed to the `forward` function as a parameter in Listing 5 to compute the connection score between `x` ($\boldsymbol{h}_u^{(K)}$) and all nodes v' in the `edge_subgraph`.

Listing 6: Loss function [40]

```
def compute_loss(pos_score, neg_score):
    # Margin loss
    n_edges = pos_score.shape[0]
    return (1 - pos_score.unsqueeze(1) + neg_score.view(n_edges, -1)).clamp(min=0).mean()
```

6.3 Minibatch training

6.3.1 Neighbourhood sampler

To train the prediction model for such a massive transaction graph, we also need to apply the neighbourhood sampler (introduced in section 5.2). However, we could not use the `MultiLayerFullNeighborSampler`

method similar to the approach with the node embeddings tasks because now, we have to deal with a considerably larger graph. For every node in a GNN layer, the `MultiLayerFullNeighborSampler` approach would aggregate the messages from all its neighbour nodes to fit into the memory. As a result, it would exceed the computational resources before the training process completes.
However, with `MultiLayerNeighborSampler`, each node could uniformly gather a fixed number of neighbours in each GNN layer. In Listing 7, the sampler methodology would select uniformly 10 and 25 nodes in the first and second GNN layer respectively.

Listing 7: Multi-layer neighbours sampler.

```
sampler = dgl.dataloading.MultiLayerNeighborSampler([[10, 25])
```

6.3.2 Data loader with negative sampling

In DGL, EdgeDataLoader [41] supports making negative samples for link prediction tasks via providing the `negative_sample` as a parameter in the `dataloader` (Listing 8).

Listing 8: Data loader with negative sampling.

```
dataloader = dgl.dataloading.EdgeDataLoader(
    g,
    train_seeds,
    sampler,
    negative_sampler=dgl.dataloading.negative_sampler.Uniform(5),
    batch_size=args.batch_size)
```

In Listing 8, for each source node of an edge, the `dataloader` would uniformly pick 5 destination nodes in the noise distribution to sample a negative graph via the Uniform function [42].

6.3.3 Stochastic 2-layer GCN/GraphSage

In general, the stochastic 2-layer GCN/GraphSage in Listing 9, 10 is initialised with the number of `in_features`, `hidden_features` and `out_features`. To forward the data, we need to specify the `blocks` returned from the `dataloader` in Listing 8 together with the input node features `x`. The features aggregation resulted by each GNN layer would be passed through ReLU activation function.

Listing 9: Stochastic 2-layer GCN [43]

```
class StochasticTwoLayerGCN(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
```

```
        self.conv1 = dgl.nn.GraphConv(in_features, hidden_features)
        self.conv2 = dgl.nn.GraphConv(hidden_features, out_features)

    def forward(self, blocks, x):
        x = F.relu(self.conv1(blocks[0], x))
        x = F.relu(self.conv2(blocks[1], x))
        return x
```

Listing 10: Stochastic 2-layer GraphSage.

```
class StochasticTwoLayerSAGE(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.conv1 = SAGEConv(in_features, hidden_features, 'mean')
        self.conv2 = SAGEConv(hidden_features, out_features, 'mean')

    def forward(self, blocks, x):
        x = F.relu(self.conv1(blocks[0], x))
        x = F.relu(self.conv2(blocks[1], x))
        return x
```

6.3.4 Link prediction model

When an `EdgeDataloader` is provided with the `negative_sampler` (Listing 8), it then returns three items for each minibatch, including:

- `positive_graph` contains minibatch with all the sampled edges.
- `negative_graph` contains the negative sampler with non-existing edges.
- A list of `blocks` generated by sampling the neighbourhood.

Those three items would then be passed to a link prediction model (Listing 11) which includes either stochastic 2-layer GCN or GraphSage in Listing 9, 10 to compute the `positive_score` and `negative_score`. Those two scores will then be used to calculate and minimize the loss function specified by Listing 6 during the training process.

Listing 11: Link prediction model.

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        # self.gnn = StochasticTwoLayerGCN(
        #     in_features, hidden_features, out_features)

```

```
        self.gnn = StochasticTwoLayerSAGE(
            in_features, hidden_features, out_features)

    def forward(self, positive_graph, negative_graph, blocks, x):
        x = self.gnn(blocks, x)
        pos_score = self.predictor(positive_graph, x)
        neg_score = self.predictor(negative_graph, x)
        return pos_score, neg_score
```

The complete implementation of the link prediction task by taking advantage of multiple GPUs is mentioned in Appendix A.6.

6.4 Results

We conducted the experiment on both GNN variants, including GCN and GraphSage. As expected, the GraphSage model would behave much better compared to the GCN model in large graph representation learning.

After 100 epochs training with the learning rate of 0.01, the loss of GraphSage model reached lower than 0.7 while that value for GCN remained stable at about higher than 0.9 (Figure 6.1)

When evaluating the performance of two models by drawing the ROC curve, the AUC score of GraphSage reached 0.695 while the score for GCN was 0.576 (Figure 6.2).

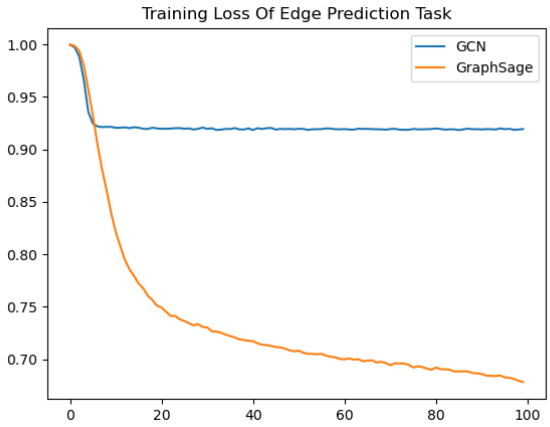


Figure 6.1: Training loss.

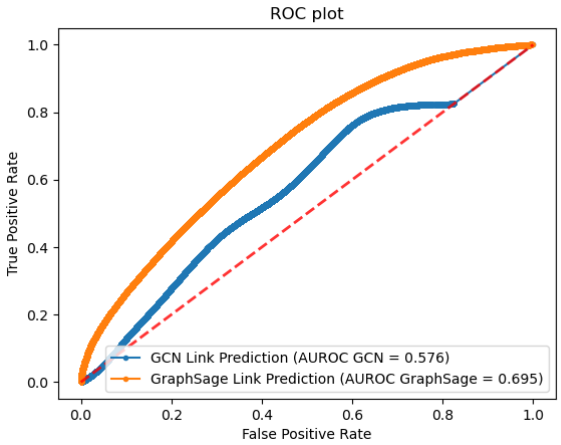


Figure 6.2: ROC plot.

Chapter 7

Conclusion and Future Work

Applying Graph Neural Networks to analyse the blockchain transaction graph is considered a fairly new approach. It is open research with various purposes depending on the outcome that a researcher would want to achieve. In this project, we identify the block range in interests and build the transaction graph without presumed node/edge features. Moreover, classes of nodes/edges have not been labelled. Therefore, we aim at analysing the transaction graph by heavily depending on the graph structure.

Initially, they did observe the Bitcoin transaction graphs, in general, have a relatively similar structure with a massive large connected component (this represents a large group of Bitcoin addresses that have a connection with each other) is surrounded by a large number of nodes/clustering of nodes. It is noticed that each weekly transaction graph had the correlative distribution in the node statistics such as degrees, connected components, clustering coefficient and PageRanks. In addition, in each weekly transaction graph, there is the same Bitcoin address that is considered as a centre of the largest connected component and also has the highest PageRank value compared to all the other node addresses.

By using Graph Neural Networks, we could learn the embeddings of nodes in low dimensions that reflect the original graph structure. With the predetermined node features, we did visualise the node embeddings with different dimensionality reduction methods. Even though there were not much information that we could extract from those graphical results, we could observe that UMAP was a methodology of choice as it could illustrate the structural behaviour of the original transaction graph with most of the predefined node features. In a link prediction task, GraphSage showed a significant performance when being compared to the GCN with higher AUC score after a hundred of epochs training.

There are much more further approaches that we could investigate in using Graph Neural Networks to analyse the blockchain transaction graph instead of being limited to the Bitcoin network only. What we are aiming to do is collect the real features for the nodes/edges in a transaction graph to build more robust Graph Neural Networks models via capturing the money flows of each node. In addition, there are various studies and works in partially labelling a Bitcoin address corresponding to their group of communities or their licit/illicit behaviours in a blockchain transaction network. Therefore, with those

node labels, combined with the graph structure learning, we would research on advancing the GNN model to detect the illegitimate actions and predict their money flows. Besides, GCN and GraphSage, there are more modern GNN instantiations that we need to investigate and compare the performance among them.

Bibliography

[1] Daily Bitcoin transactions 2017-2021.
URL https://www.statista.com/statistics/730806/daily-number-of-bitcoin-transactions

[2] J. Wang, Q. Wang, N. Zhou, Y. Chi, A Novel Electricity Transaction Mode of Microgrids Based on Blockchain and Continuous Double Auction, Energies 10 (12) (2017) 1971, number: 12 Publisher: Multidisciplinary Digital Publishing Institute. doi:10.3390/en10121971.
URL https://www.mdpi.com/1996-1073/10/12/1971

[3] Bitcoin and Cryptocurrency Technologies.
URL https://bitcoinbook.cs.princeton.edu/

[4] W. L. Hamilton, R. Ying, J. Leskovec, Inductive Representation Learning on Large Graphs, arXiv:1706.02216 [cs, stat]ArXiv: 1706.02216 (Sep. 2018).
URL http://arxiv.org/abs/1706.02216

[5] Deep Graph Library.
URL https://www.dgl.ai/

[6] I. Alarab, S. Prakoonwit, M. I. Nacer, Competence of Graph Convolutional Networks for Anti-Money Laundering in Bitcoin Blockchain, in: Proceedings of the 2020 5th International Conference on Machine Learning Technologies, ICMLT 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 23–27. doi:10.1145/3409073.3409080.
URL https://doi.org/10.1145/3409073.3409080

[7] Bitcoin price today, BTC to USD live, marketcap and chart | CoinMarketCap.
URL https://coinmarketcap.com/currencies/bitcoin/

[8] Chapter 6: Stochastic Training on Large Graphs — DGL 0.6.1 documentation.
URL https://docs.dgl.ai/en/0.6.x/guide/minibatch.html

[9] Blockchain Explorer - Search the Blockchain | BTC | ETH | BCH.
URL https://www.blockchain.com/btc/blocks

[10] Genesis block - Bitcoin Wiki.
URL https://en.bitcoin.it/wiki/Genesis_block

[11] Blockchain Explorer - Search the Blockchain | BTC | ETH | BCH.
URL https://www.blockchain.com/charts/blocks-size

[12] D. Ron, A. Shamir, Quantitative Analysis of the Full Bitcoin Transaction Graph, in: D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, A.-R. Sadeghi (Eds.), Financial Cryptography and Data Security, Vol. 7859, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 6–24, series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-642-39884-1_2.
URL http://link.springer.com/10.1007/978-3-642-39884-1_2

[13] I. Alarab, S. Prakoonwit, M. I. Nacer, Competence of Graph Convolutional Networks for Anti-Money Laundering in Bitcoin Blockchain, in: Proceedings of the 2020 5th International Conference on Machine Learning Technologies, ACM, Beijing China, 2020, pp. 23–27. doi:10.1145/3409073.3409080.
URL https://dl.acm.org/doi/10.1145/3409073.3409080

[14] Blockchain Explorer - Search the Blockchain | BTC | ETH | BCH.
URL https://www.blockchain.com/api/blockchain_api

[15] SoChain | Bitcoin API and More.
URL https://sochain.com/api/

[16] M. Nofer, P. Gomber, O. Hinz, D. Schiereck, Blockchain, Business & Information Systems Engineering 59 (3) (2017) 183–187. doi:10.1007/s12599-017-0467-3.
URL https://doi.org/10.1007/s12599-017-0467-3

[17] G. O. Karame, E. Androulaki, S. Capkun, Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin 17.

[18] S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System 9.

[19] L. Wang, Y. Liu, Exploring Miner Evolution in Bitcoin Network, in: J. Mirkovic, Y. Liu (Eds.), Passive and Active Measurement, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2015, pp. 290–302. doi:10.1007/978-3-319-15509-8_22.

[20] M. Szydlo, Merkle Tree Traversal in Log Space and Time, in: C. Cachin, J. L. Camenisch (Eds.), Advances in Cryptology - EUROCRYPT 2004, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2004, pp. 541–554. doi:10.1007/978-3-540-24676-3_32.

[21] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, M. Sun, Graph Neural Networks: A Review of Methods and Applications, arXiv:1812.08434 [cs, stat]ArXiv: 1812.08434 (Oct. 2021).
URL http://arxiv.org/abs/1812.08434

[22] T. N. Kipf, M. Welling, Semi-Supervised Classification with Graph Convolutional Networks, arXiv:1609.02907 [cs, stat]ArXiv: 1609.02907 (Feb. 2017).
URL http://arxiv.org/abs/1609.02907

[23] J. L. Elman, Finding structure in time, Cognitive Science 14 (2) (1990) 179–211. doi:10.1016/0364-0213(90)90002-E.
URL https://www.sciencedirect.com/science/article/pii/036402139090002E

[24] PyG Documentation — pytorch_geometric 2.0.2 documentation.
URL https://pytorch-geometric.readthedocs.io/en/latest/

[25] Graph Representation Learning Book.
URL https://www.cs.mcgill.ca/~wlh/grl_book/

[26] Spektral.
URL https://graphneural.network/

[27] Elliptic Data Set | Kaggle.
URL https://www.kaggle.com/ellipticco/elliptic-data-set

[28] M. Weber, G. Domeniconi, J. Chen, D. K. I. Weidele, C. Bellei, T. Robinson, C. E. Leiserson, Anti-Money Laundering in Bitcoin: Experimenting with Graph Convolutional Networks for Financial Forensics, arXiv:1908.02591 [cs, q-fin]ArXiv: 1908.02591 (Jul. 2019).
URL http://arxiv.org/abs/1908.02591

[29] Tesla Now Accepts Bitcoin as Payment for Cars, Musk Says, Bloomberg.com (Mar. 2021).
URL https://www.bloomberg.com/news/articles/2021-03-24/you-can-now-buy-a-tesla-with-bitcoin-elon-musk-says

[30] Elon Musk backtracks, says Tesla won't accept bitcoin.
URL https://www.nbcnews.com/tech/tech-news/elon-musk-backtracks-says-tesla-wont-acc

[31] Blockchain Explorer - Search the Blockchain | BTC | ETH | BCH.
URL https://www.blockchain.com/api/blockchain_api

[32] . G. T. Ltd, Graphia.
URL https://graphia.app/

[33] D. J. Watts, S. H. Strogatz, Collective dynamics of 'small-world' networks 393 (1998) 3.

[34] L. Page, S. Brin, R. Motwani, T. Winograd, The pagerank citation ranking: Bringing order to the web., Technical Report 1999-66, Stanford InfoLab, previous number = SIDL-WP-1999-0120 (November 1999).
URL http://ilpubs.stanford.edu:8090/422/

[35] H. Cui, Z. Lu, P. Li, C. Yang, On Positional and Structural Node Features for Graph Neural Networks on Non-attributed Graphs, arXiv:2107.01495 [cs]ArXiv: 2107.01495 (Jul. 2021).
URL http://arxiv.org/abs/2107.01495

[36] Papers with Code - MUTAG Dataset.
URL https://paperswithcode.com/dataset/mutag

[37] dgl.dataloading — DGL 0.6.1 documentation.
URL https://docs.dgl.ai/en/0.6.x/api/python/dgl.dataloading.html#dgl.dataloading.neighbor.MultiLayerFullNeighborSampler

[38] 6.1 Training GNN for Node Classification with Neighborhood Sampling — DGL 0.6.1 documentation.
URL https://docs.dgl.ai/en/0.6.x/guide/minibatch-node.html#guide-minibatch-node-classification-sampler

[39] dgl.graph — DGL 0.6.1 documentation.
URL https://docs.dgl.ai/en/0.6.x/generated/dgl.graph.html

[40] 5.3 Link Prediction — DGL 0.6.1 documentation.
URL https://docs.dgl.ai/en/0.6.x/guide/training-link.html

[41] dgl.dataloading — DGL 0.6.1 documentation.
URL https://docs.dgl.ai/en/0.6.x/api/python/dgl.dataloading.html#dgl.dataloading.pytorch.EdgeDataLoader

[42] dgl.dataloading — DGL 0.6.1 documentation.
URL https://docs.dgl.ai/en/0.6.x/api/python/dgl.dataloading.html#dgl.dataloading.negative_sampler.Uniform

[43] 6.3 Training GNN for Link Prediction with Neighborhood Sampling — DGL 0.5.3 documentation.
URL https://docs.dgl.ai/en/0.5.x/guide/minibatch-link.html#

Appendix A

Appendix

A.1 Download Bitcoin transaction data of a given block range.

```
import csv
import sys
from time import sleep
import requests
import threading

# Predefined block height range
FROM_BLOCK = sys.argv[1]
TO_BLOCK = sys.arg[2]

# Number of blocks to be gotten concurrently
CONCURRENT_BLOCKS_TO_GET = 500


lock = threading.Lock()
threads = list()


def get_block(b_height):
    """Append the csv file with the edge list get from a block.

    Parameters
    ----------
    b_height: Block height
    string

    Returns
    -------
    void
    """
    while True:
        try:
```

```
            # Get block hash of given block height
            b_info = f'https://chain.so//api/v2/get_blockhash/BTC/{b_height}'
            b_hash = requests.get(b_info).json()["data"]["blockhash"]
            # Get full block data
            b_url = f'https://blockchain.info/rawblock/{b_hash}'
            b_data = requests.get(b_url).json()
            with lock:
                # For each transaction in a block
                for index, tx in enumerate(b_data["tx"]):
                    inputs = tx["inputs"]
                    outputs = tx["out"]
                    # Connect each input address to all output addresses
                    for i in inputs:
                        if i["prev_out"] and "addr" in i["prev_out"]:
                            i_address = i["prev_out"]["addr"]
                        else:
                            i_address = "Coinbase"
                        for o in outputs:
                            if o["spent"] and "addr" in o:
                                o_address = o["addr"]
                                # Each csv_row contains a link
                                # between one input address to one output address
                                csv_row = [i_address, o_address]
                                # Append that row to csv file
                                writer_object = csv.writer(f_object)
                                writer_object.writerow(csv_row)
            break
        # Exception from the server
        except:
            # Sleep fo 5 seconds and resend the request
            sleep(5)


with open(f'txs_{FROM_BLOCK}_{TO_BLOCK}.csv', 'a') as f_object:
    # For each block within the predefined block height range
    for b_height in range(FROM_BLOCK, TO_BLOCK - 1, -1):
        # Concurrently fetch the block transaction data and append it to the csv file
        x = threading.Thread(target=get_block, args=(b_height,))
        threads.append(x)
        x.start()
        if len(threads) % CONCURRENT_BLOCKS_TO_GET == 0 and len(threads) > 0:
            for thread in threads:
                thread.join()
    f_object.close()
```

A.2 Extract and draw distributions of basic graph statistics

```python
import csv
import sys

import igraph as ig
import matplotlib.pyplot as plt
import networkx as nx
import pandas as pd
import pylab as pl
from igraph import Graph


def get_time_duration(file_name):
    """Get weekly time duration as a string beginning with 'W'.
    e.g., W1, W2, ..., W7

    Parameters
    ----------
    file_name : string

    Returns
    -------
    duration : string
    """
    duration = ''
    blocks_list = ['677146_676105', '678021_677147', '679186_678022',
                   '680019_679187', '680981_680020', '682068_680982',
                   '683344_682069']
    for index, blocks in enumerate(blocks_list):
        if file_name == f'txs_{blocks}.csv':
            duration = f'W{index+1}'
            return duration
    return duration


file_name = sys.argv[1]
duration = get_time_duration(file_name)


# Read CSV
header_list = ['src', 'dst']
graph_df = pd.read_csv(file_name, names=header_list)

# Init Networkx object
Graphtype = nx.DiGraph()
G = nx.from_pandas_edgelist(graph_df, 'src', 'dst', create_using=Graphtype)

# Export the Networkx object
nx.write_gml(G, f'{duration}.gml')
```

```
49
50 # Connected components list
51 weak_cc_list = [
52     G.subgraph(c).number_of_nodes() for c in nx.weakly_connected_components(G)]
53 strong_cc_list = [
54     G.subgraph(c).number_of_nodes() for c in nx.strongly_connected_components(G)]
55
56 # Init igraph object
57 h = ig.Graph.from_networkx(G)
58
59 # Clustering Coefficient list
60 clustering_co_list = Graph.transitivity_local_undirected(h)
61
62 # PageRank list
63 pagerank_list = Graph.pagerank(h)
64
65 """
66 Get basic graph info
67 """
68 # Nodes, edges
69 num_nodes = G.number_of_nodes()
70 num_edges = G.number_of_edges()
71
72 # Get the node with highest PageRank value
73 max_pr = max(pagerank_list)
74 most_imp_index = pagerank_list.index(max_pr)
75 nodes_list = list(G.nodes)
76 most_imp = nodes_list[most_imp_index]
77
78 # Export basic graph info
79 with open(f'{duration}_info.csv', 'w') as f_object:
80     info_row = [num_nodes, num_edges, most_imp]
81     writer_object = csv.writer(f_object)
82     writer_object.writerow(info_row)
83
84
85 """
86 Draw and export charts
87 """
88 # PageRank
89 df_pr = pd.DataFrame(pagerank_list)
90 pl.figure(figsize=(12, 8))
91 label = f'PageRank {duration}'
92 pl.xlabel(label)
93 pl.ylabel('Frequency')
94 plt.hist(pagerank_list, bins=50, log=True,
95          alpha=0.5, color='b', histtype='step')
96 plt.savefig(label)
97
98 # Clustering Coefficient
99 co_df = pd.DataFrame(clustering_co_list)
```

```
100 label = f'Clustering Coefficient {duration}'
101 pl.figure(figsize=(12, 8))
102 pl.xlabel(label)
103 pl.ylabel('Frequency')
104 plt.hist(clustering_co_list, bins=50, log=True,
105          alpha=0.5, color='b', histtype='step')
106 plt.savefig(label)
107
108 # Connected Components
109 strong_cc_df = pd.DataFrame(strong_cc_list)
110 weak_cc_df = pd.DataFrame(weak_cc_list)
111 pl.figure(figsize=(12, 8))
112 strong_cc_title = f'Strongly Connected Components {duration}'
113 plt.title(strong_cc_title)
114 pl.xlabel('Connected Component Size')
115 pl.ylabel('Frequency')
116 plt.hist(strong_cc_list, bins=10,
117          log=True, alpha=0.5, color='b', histtype='step')
118 plt.savefig(strong_cc_title)
119
120 pl.figure(figsize=(12, 8))
121 weak_cc_title = f'Weakly Connected Components {duration}'
122 plt.title(weak_cc_title)
123 pl.xlabel('Connected Component Size')
124 pl.ylabel('Frequency')
125 plt.hist(weak_cc_list, bins=10,
126          log=True, alpha=0.5, color='b', histtype='step')
127 plt.savefig(weak_cc_title)
128
129
130 # In, Out Degrees
131 def degree_histogram_directed(G, in_degree=False, out_degree=False):
132     """Return a list of the frequency of each degree value.
133
134     Parameters
135     ----------
136     G : Networkx graph
137     in_degree : bool
138     out_degree : bool
139
140     Returns
141     -------
142     freq : list
143        A list of frequencies of degrees.
144     """
145     nodes = G.nodes()
146     if in_degree:
147         in_degree = dict(G.in_degree())
148         degseq = [in_degree.get(k, 0) for k in nodes]
149     elif out_degree:
150         out_degree = dict(G.out_degree())
```

```
        degseq = [out_degree.get(k, 0) for k in nodes]
    else:
        degseq = [v for k, v in G.degree()]
    dmax = max(degseq)+1
    freq = [0 for d in range(dmax)]
    for d in degseq:
        freq[d] += 1
    return freq


in_degree_freq = degree_histogram_directed(G, in_degree=True)
out_degree_freq = degree_histogram_directed(G, out_degree=True)

plt.figure(figsize=(12, 8))
plt.loglog(range(len(in_degree_freq)),
           in_degree_freq, 'go-', label='in-degree')
plt.xlabel('In Degree {duration}')
plt.ylabel('Frequency')
plt.savefig(f'In Degree {duration}')

plt.figure(figsize=(12, 8))
plt.loglog(range(len(out_degree_freq)),
           out_degree_freq, 'bo-', label='out-degree')
plt.xlabel('Out Degree {duration}')
plt.ylabel('Frequency')
plt.savefig(f'Out Degree {duration}')
```

A.3 Create DGL dataset for each transaction graph with predefined node features.

```
import os
import sys

import dgl
import igraph as ig
import networkx as nx
import torch
from dgl import load_graphs, save_graphs
from dgl.data import DGLDataset
from igraph import Graph

# Import the Networkx object
gml_file = sys.argv[1]
G = nx.read_gml(gml_file)

# Extract duration from gml_file
# (e.g., gml_file = 'W1.gml' -> duration = 'W1')
```

```
duration = gml_file.split('.')[0]

# DGL graph init
g = dgl.from_networkx(G)

# DGL dataset init


class BlockchainDataset(DGLDataset):
    def __init__(self):
        super().__init__(
            name=f'blockchain_{duration}_{feat_type}', save_dir=f'./')

    def process(self):
        self.graph = g

    def save(self):
        graph_path = os.path.join(
            self.save_path + f'blockchain_{duration}_{feat_type}.bin')
        save_graphs(graph_path, self.graph)

    def load(self):
        # load processed data from directory `self.save_path`
        graph_path = os.path.join(
            self.save_path + f'blockchain_{duration}_{feat_type}.bin')
        print(graph_path)
        self.graph = load_graphs(graph_path)

    def has_cache(self):
        # check whether there are processed data in `self.save_path`
        graph_path = os.path.join(
            self.save_path + f'blockchain_{duration}_{feat_type}.bin')
        return os.path.exists(graph_path)

    def __getitem__(self, idx):
        return self.graph

    def __len__(self):
        return 1


feat_types = ['pr', 'const', 'indeg', 'outdeg', 'id']
for feat_type in feat_types:
    if feat_type == 'pr':
        # Init igraph object
        h = ig.Graph.from_networkx(G)
        # PageRank list
        pr_list = Graph.pagerank(h)
        g.ndata['feat'] = torch.tensor([[pv.item()] for pv in pr_list])
    elif feat_type == 'const':
        g.ndata['feat'] = torch.ones([G.number_of_nodes(), 1])
```

```
    elif feat_type == 'indeg':
        g.ndata['feat'] = torch.tensor([[pv.item()] for pv in g.in_degrees()])
    elif feat_type == 'outdeg':
        g.ndata['feat'] = torch.tensor([[pv.item()] for pv in g.out_degrees()])

    dataset = BlockchainDataset()
```

A.4 Generate node embeddings.

```
import os
import sys

import dgl
import dgl.function as fn
import dgl.nn.pytorch as dglnn
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
import tqdm
from dgl import load_graphs
import torch.optim as optim

duration = sys.argv[1]

# Params
in_features = 1
hidden_features = 128
out_features = 50
num_layers = 2
dropout = 0.5
learning_rate = 0.01
epoch = 10

# GraphSage model
class SAGE(nn.Module):
    def __init__(self,
                 in_feats,
                 n_hidden,
                 n_classes,
                 n_layers,
                 activation,
                 dropout):
        super().__init__()
        self.n_layers = n_layers
        self.n_hidden = n_hidden
        self.n_classes = n_classes
```

```
40          self.layers = nn.ModuleList()
41
42          self.layers.append(dglnn.SAGEConv(in_feats, n_hidden, 'mean'))
43          for i in range(1, n_layers - 1):
44              self.layers.append(dglnn.SAGEConv(n_hidden, n_hidden, 'mean'))
45          self.layers.append(dglnn.SAGEConv(n_hidden, n_classes, 'mean'))
46          self.dropout = nn.Dropout(dropout)
47          self.activation = activation
48
49      def forward(self, blocks, x):
50          h = x.float()
51          for l, (layer, block) in enumerate(zip(self.layers, blocks)):
52              h = layer(block, h)
53              if l != len(self.layers) - 1:
54                  h = self.activation(h)
55                  h = self.dropout(h)
56          return h
57
58      def inference(self, g, x, device):
59          x = x.float()
60          for l, layer in enumerate(self.layers):
61              y = torch.zeros(g.num_nodes(), self.n_hidden if l !=
62                              len(self.layers) - 1 else self.n_classes)
63
64              sampler = dgl.dataloading.MultiLayerNeighborSampler([10])
65              print('arrange ', torch.arange(g.num_nodes()))
66              dataloader = dgl.dataloading.NodeDataLoader(
67                  g,
68                  torch.arange(g.num_nodes()),
69                  sampler,
70                  batch_size=10240,
71                  shuffle=True,
72                  drop_last=False,
73                  num_workers=0)
74
75              for input_nodes, output_nodes, blocks in tqdm.tqdm(dataloader):
76                  block = blocks[0].int().to(device)
77
78                  h = x[input_nodes].to(device)
79                  h = layer(block, h)
80
81                  if l != len(self.layers) - 1:
82                      h = self.activation(h)
83                      h = self.dropout(h)
84
85                  y[output_nodes] = h.cpu()
86
87              x = y
88          return y
89
90
```

```
# Cross entropy loss
class CrossEntropyLoss(nn.Module):
    def forward(self, block_outputs, pos_graph, neg_graph):
        with pos_graph.local_scope():
            pos_graph.ndata['h'] = block_outputs
            pos_graph.apply_edges(fn.u_dot_v('h', 'h', 'score'))
            pos_score = pos_graph.edata['score']
        with neg_graph.local_scope():
            neg_graph.ndata['h'] = block_outputs
            neg_graph.apply_edges(fn.u_dot_v('h', 'h', 'score'))
            neg_score = neg_graph.edata['score']

        score = torch.cat([pos_score, neg_score])
        label = torch.cat([torch.ones_like(pos_score),
                           torch.zeros_like(neg_score)]).long()
        loss = F.binary_cross_entropy_with_logits(score, label.float())
        return loss


def main():
    device = torch.device('cuda')
    feat_types = ['const', 'pr', 'indeg', 'outdeg']
    for feat_type in feat_types:
        graph_path = os.path.join(f'blockchain_{duration}_{feat_type}.bin')
        graphs = load_graphs(graph_path)
        g = graphs[0][0]

        # Dataloader
        cuda_g = g.to('cuda')
        n_edges = cuda_g.num_edges()
        train_seeds = np.arange(n_edges)
        sampler = dgl.dataloading.MultiLayerNeighborSampler([10, 25])

        dataloader = dgl.dataloading.EdgeDataLoader(
            g, train_seeds, sampler,
            negative_sampler=dgl.dataloading.negative_sampler.Uniform(5),
            batch_size=10240,
            shuffle=True,
            drop_last=False,
            num_workers=0,
            device=device
        )

        # Get emb
        nfeat = g.ndata['feat']
        print(nfeat)
        model = SAGE(in_features, hidden_features,
                     out_features, num_layers, F.relu, dropout)
        model = model.to(device)
        loss_fcn = CrossEntropyLoss()
        opt = optim.Adam(model.parameters(), lr=learning_rate)
```

```
142
143         iter = 1
144         for j in range(epoch):
145             print(f"--------------EPOCH {iter}--------------")
146             iter += 1
147             for input_nodes, pos_graph, neg_graph, blocks in dataloader:
148                 batch_inputs = nfeat[input_nodes].to(
149                     torch.device('cuda'))
150                 pos_graph = pos_graph.to(torch.device('cuda'))
151                 neg_graph = neg_graph.to(torch.device('cuda'))
152                 blocks = [block.int().to(torch.device('cuda'))
153                           for block in blocks]
154                 # Compute loss and prediction
155                 batch_pred = model(blocks, batch_inputs)
156                 loss = loss_fcn(batch_pred, pos_graph, neg_graph)
157                 opt.zero_grad()
158                 loss.backward()
159                 opt.step()
160         model.eval()
161         with torch.no_grad():
162             if isinstance(model, SAGE):
163                 node_emb = model.inference(g, g.ndata['feat'], device)
164         nump_emb = node_emb.cpu().detach().numpy()
165         node_emb_df = pd.DataFrame(nump_emb)
166         node_emb_df.to_csv(
167             f"emb_{duration}_{feat_type}.csv", index=False)
168
169
170 if __name__ == '__main__':
171     main()
```

A.5 Node embeddings visualisation.

```
1  import sys
2
3  import matplotlib.pyplot as plt
4  import numpy as np
5  import pandas as pd
6  import seaborn as sns
7  import tqdm
8  from sklearn.decomposition import PCA
9  from cuml import UMAP
10 from cuml.manifold import TSNE
11 from scipy import stats
12
13 duration = sys.argv[1]  # either 'w1', 'w2', ... or 'w7'
14 type = sys.argv[2]  # either 'PCA', 'TSNE' or 'UMAP'
15
```

```
16
17  def get_feat_name(ft):
18      if ft == 'const':
19          return 'Constant'
20      elif ft == 'pr':
21          return 'PageRank'
22      elif ft == 'indeg':
23          return 'In Degree'
24      elif ft == 'outdeg':
25          return 'Out Degree'
26
27
28  feat_types = ['const', 'pr', 'indeg', 'outdeg']
29
30  for feat_type in feat_types:
31      print(feat_type)
32      # Read emb
33      file_name = f"emb_{duration}_{feat_type}.csv"
34      node_emb_df = pd.read_csv(file_name)
35      if type == 'PCA':
36          e_pca = PCA(n_components=2)
37          edges_emb = e_pca.fit_transform(node_emb_df)
38          df_pca = pd.DataFrame(edges_emb, columns=['PCA1', 'PCA2'])
39          df_pca.to_csv(
40              f"{type}_{feat_type}_{duration}", index=False)
41          # Plot PCA
42          plt.figure(figsize=(11, 11))
43          plt.title(
44              f'2D Node Embeddings Visualisation {duration} (Feature Type: {get_feat_name(feat_type)})')
45          sns.scatterplot(x=f'{type}1',
46                          y=f'{type}2', data=df_pca)
47          plt.savefig(
48              f'2D {type} (Feature type: {get_feat_name(feat_type)}) {duration}.png')
49
50          # PCA with outliers removed
51          new_pca_df = df_pca[(np.abs(stats.zscore(df_pca)) < 3).all(axis=1)]
52          new_pca_df.head
53
54          plt.figure(figsize=(11, 11))
55          plt.title(
56              f'2D Node Embeddings Visualisation {duration} (Feature Type: {get_feat_name(feat_type)}, Ou
57          sns.scatterplot(x=f'{type}1', y=f'{type}2', data=new_pca_df)
58          plt.savefig(
59              f'2D {type} (Feature Type: {get_feat_name(feat_type)} {duration}, Outliers Removed)')
60
61      elif type == 'TSNE':
62          e_tsne = TSNE(n_neighbors=1000, perplexity=50,
63                        learning_rate=150, n_iter=3000, verbose=4)
64          tsne_node_emb = e_tsne.fit_transform(node_emb_df)
65
66          df_tsne = pd.DataFrame(tsne_node_emb, columns=['t-SNE1', 't-SNE2'])
```

```
        df_tsne.to_csv(
            f"{type}_{feat_type}_{duration}", index=False)

        plt.figure(figsize=(11, 11))
        plt.title(
            f'2D Node Embeddings Visualisation {duration} (Feature Type: {get_feat_name(feat_type)})')
        sns.scatterplot(x='t-SNE1', y='t-SNE2', data=df_tsne)
        plt.savefig(
            f'2D {type} (Feature type: {get_feat_name(feat_type)}) {duration}.png')

        # TSNE outliers removed
        new_tsne_df = df_tsne[(np.abs(stats.zscore(df_tsne)) < 3).all(axis=1)]
        plt.figure(figsize=(11, 11))
        plt.title(
            f'2D Node Embeddings Visualisation {duration} (Feature Type: {get_feat_name(feat_type)}, Ou
        sns.scatterplot(x='t-SNE1', y='t-SNE2', data=new_tsne_df)
        plt.savefig(
            f'2D {type} (Feature Type: {get_feat_name(feat_type)}, Outliers Removed) {duration}')

    elif type == 'UMAP':
        reducer = UMAP(n_epochs=250, n_neighbors=100)
        umap_emb = reducer.fit_transform(node_emb_df)
        df_umap = pd.DataFrame(umap_emb, columns=['UMAP1', 'UMAP2'])
        df_umap.to_csv(
            f"{type}_{feat_type}_{duration}", index=False)

        plt.figure(figsize=(11, 11))
        plt.title(
            f'2D Node Embeddings Visualisation {duration} (Feature Type: {get_feat_name(feat_type)})')
        sns.scatterplot(x=f'{type}1', y=f'{type}2', data=df_umap)
        plt.savefig(
            f'2D {type} (Feature Type: {get_feat_name(feat_type)}) {duration}')

        # UMAP outliters removed
        new_umap = umap_emb[(np.abs(stats.zscore(umap_emb)) < 3).all(axis=1)]
        df_umap = pd.DataFrame(new_umap, columns=[f'{type}1', f'{type}2'])

        plt.figure(figsize=(11, 11))
        plt.title(
            f'2D Node Embeddings Visualisation {duration} (Feature Type: {get_feat_name(feat_type)}, Ou
        sns.scatterplot(x=f'{type}1', y=f'{type}2', data=df_umap)
        plt.savefig(
            f'2D {type} (Feature Type: {get_feat_name(feat_type)}, Outliers Removed) {duration}')
```

A.6 Link prediction.

```
import torch as th
from torch.nn.parallel import DistributedDataParallel
```

```
import os
import sys
import dgl.multiprocessing as mp

import dgl
import dgl.function as fn
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.nn.functional as F
from dgl import load_graphs
from dgl.nn import SAGEConv
from sklearn.metrics import roc_auc_score, roc_curve

num_gpus = 2

in_features = 1
n_classes = 2
hidden_features = 32
out_features = 16
num_layers = 2
dropout = 0.5
learning_rate = 0.01
epoch = 100
batch_size = 100240
num_workers = 0
mul_layer_neigh_sampler = [10, 25]
num_negs = 1
neg_share = 0
num_hidden = 16
log_every = 20
eval_every = 100

location = './'
feat_types = ['pr']
file_name = sys.argv[1]

# Distributed training


def init_process_group(world_size, rank):
    dist.init_process_group(
        backend='nccl',
        init_method='tcp://127.0.0.1:12345',
        world_size=world_size,
        rank=rank)


```

```
54  class DotPredictor(nn.Module):
55      def forward(self, edge_subgraph, x):
56          with edge_subgraph.local_scope():
57              edge_subgraph.ndata['x'] = x
58              edge_subgraph.apply_edges(
59                  dgl.function.u_dot_v('x', 'x', 'score'))
60              return edge_subgraph.edata['score']
61
62
63  def compute_loss(pos_score, neg_score):
64      # Margin loss
65      n_edges = pos_score.shape[0]
66      return (1 - pos_score.unsqueeze(1) + neg_score.view(n_edges, -1)).clamp(min=0).mean()
67
68
69  def compute_auc(pos_score, neg_score):
70      scores = torch.cat([pos_score, neg_score]).numpy()
71      labels = torch.cat(
72          [torch.ones(pos_score.shape[0]), torch.zeros(neg_score.shape[0])]).numpy()
73      return roc_auc_score(labels, scores)
74
75
76  def get_feat_name(ft):
77      if ft == 'const':
78          return 'Constant'
79      elif ft == 'pr':
80          return 'PageRank'
81      elif ft == 'indeg':
82          return 'In Degree'
83      elif ft == 'outdeg':
84          return 'Out Degree'
85
86  #  -------------------------GCN-------------------------
87
88  class StochasticTwoLayerGCN(nn.Module):
89      def __init__(self, in_features, hidden_features, out_features):
90          super().__init__()
91          self.conv1 = dgl.nn.GraphConv(in_features, hidden_features)
92          self.conv2 = dgl.nn.GraphConv(hidden_features, out_features)
93
94      def forward(self, blocks, x):
95          x = F.relu(self.conv1(blocks[0], x))
96          x = F.relu(self.conv2(blocks[1], x))
97          return x
98
99
100 class ModelGCN(nn.Module):
101     def __init__(self, in_features, hidden_features, out_features):
102         super().__init__()
103         self.gcn = StochasticTwoLayerGCN(
104             in_features, hidden_features, out_features)
```

```
105         self.predictor = DotPredictor()
106
107     def forward(self, positive_graph, negative_graph, blocks, x):
108         x = self.gcn(blocks, x)
109         pos_score = self.predictor(positive_graph, x)
110         neg_score = self.predictor(negative_graph, x)
111         return pos_score, neg_score
112
113
114 #  ----------------------------SAGE----------------------------
115 class StochasticTwoLayerSAGE(nn.Module):
116     def __init__(self, in_features, hidden_features, out_features):
117         super().__init__()
118         self.conv1 = SAGEConv(in_features, hidden_features, 'mean')
119         self.conv2 = SAGEConv(hidden_features, out_features, 'mean')
120
121     def forward(self, blocks, x):
122         x = F.relu(self.conv1(blocks[0], x))
123         x = F.relu(self.conv2(blocks[1], x))
124         return x
125
126
127 class ModelSAGE(nn.Module):
128     def __init__(self, in_features, hidden_features, out_features):
129         super().__init__()
130         self.sage = GraphSAGE(
131             in_features, hidden_features, out_features)
132         self.predictor = DotPredictor()
133
134     def forward(self, positive_graph, negative_graph, blocks, x):
135         x = self.sage(blocks, x)
136         pos_score = self.predictor(positive_graph, x)
137         neg_score = self.predictor(negative_graph, x)
138         return pos_score, neg_score
139
140
141 def main(rank, num_gpus):
142     init_process_group(num_gpus, rank)
143     device = torch.device('cuda:{:d}'.format(rank))
144     for feat_type in feat_types:
145         graph_path = os.path.join(
146             location + file_name)
147         graphs = load_graphs(graph_path)
148         dg = graphs[0][0]
149         g = dgl.add_reverse_edges(dg)
150
151         n_edges = g.num_edges()
152         train_seeds = np.arange(n_edges)
153         sampler = dgl.dataloading.MultiLayerNeighborSampler(
154             mul_layer_neigh_sampler)
155         dataloader = dgl.dataloading.EdgeDataLoader(
```

```
            g, train_seeds, sampler,
            negative_sampler=dgl.dataloading.negative_sampler.Uniform(5),
            batch_size=batch_size,
            shuffle=True,
            drop_last=False,
            device=device,
            num_workers=num_workers,
            use_ddp=True
        )

        print(f"data loader len: {len(dataloader)}")

        model = ModelGCN(in_features, hidden_features, out_features)
        model = model.to(device)
        model = DistributedDataParallel(
            model, device_ids=[device], output_device=device)
        opt = torch.optim.Adam(model.parameters())

        iter = 1

        losses = []
        auc_scores = []

        for j in range(epoch):
            print(f"--------------EPOCH {iter}--------------")
            iter += 1

            avr_loss = 0
            auc_score = 0

            for step, (input_nodes, positive_graph, negative_graph, blocks) in enumerate(dataloader):
                blocks = [b.to(device) for b in blocks]
                positive_graph = positive_graph.to(device)
                negative_graph = negative_graph.to(device)
                input_features = blocks[0].srcdata['feat']
                pos_score_gcn, neg_score_gcn = model(
                    positive_graph, negative_graph, blocks, input_features)
                loss = compute_loss(pos_score_gcn.to(
                    device), neg_score_gcn.to(device), device)

                opt.zero_grad()
                loss.backward()
                opt.step()

                if step % 100 == 0:
                    print(
                        f"Epoch: {j}, Step: {step}, Current loss: {loss.item()}")

                avr_loss += loss.item()

                with torch.no_grad():
```

```
                    auc_score += compute_auc(pos_score_gcn.cpu(),
                                             neg_score_gcn.cpu())

            avr_loss = avr_loss/len(dataloader)
            auc_score = auc_score/len(dataloader)
            losses.append(avr_loss)
            auc_scores.append(auc_score)
            print(f'current batch loss: {avr_loss}')
            print(f'AUC: {auc_score}')

        # GraphSage Model
        model = ModelSAGE(in_features, hidden_features, out_features)
        model = model.cuda()
        opt = torch.optim.Adam(model.parameters())

        iter = 1
        losses_sage = []
        auc_score_sages = []

        for j in range(epoch):
            print(f"--------------EPOCH {iter}--------------")
            iter += 1

            avr_loss_sage = 0
            auc_score_sage = 0

            for step, (input_nodes, positive_graph, negative_graph, blocks) in enumerate(dataloader):
                blocks = [b.to(device) for b in blocks]
                positive_graph = positive_graph.to(device)
                negative_graph = negative_graph.to(device)
                input_features = blocks[0].srcdata['feat']
                pos_score_sage, neg_score_sage = model(
                    positive_graph, negative_graph, blocks, input_features)
                loss = compute_loss(pos_score_sage.to(
                    device), neg_score_sage.to(device), device)

                opt.zero_grad()
                loss.backward()
                opt.step()

                avr_loss_sage += loss.item()

                with torch.no_grad():
                    auc_score_sage += compute_auc(pos_score_sage.cpu(),
                                                  neg_score_sage.cpu())

            avr_loss_sage = avr_loss_sage/len(dataloader)
            auc_score_sage = auc_score_sage/len(dataloader)
            losses_sage.append(avr_loss_sage)

            # --------------Save to csv--------------
```

```
258             loss_gcn_df = pd.DataFrame(losses, columns=["gcn_lost"])
259             loss_gcn_df.to_csv('gcn_lost.csv', index=False)
260             loss_sage_df = pd.DataFrame(losses_sage, columns=["sage_lost"])
261             loss_sage_df.to_csv('sage_lost.csv', index=False)
262
263             auc_score_sages.append(auc_score_sage)
264             print(f'current batch loss: {avr_loss_sage}')
265             print(f'AUC: {auc_score_sage}')
266         plt.figure()
267         plt.title(
268             f"Training Loss Of Edge Prediction Task")
269         plt.plot(losses, label="GCN")
270         plt.plot(losses_sage, label="GraphSage")
271         plt.legend()
272         plt.savefig(
273             f"Training Loss Of Edge Prediction Task")
274
275         with torch.no_grad():
276             auc_gcn = compute_auc(pos_score_gcn.cpu(), neg_score_gcn.cpu())
277             auc_sage = compute_auc(pos_score_sage.cpu(), neg_score_sage.cpu())
278             print('AUC GCN', auc_gcn)
279             print('AUC Sage', auc_sage)
280
281         def get_roc(pos_score, neg_score):
282             scores = torch.cat([pos_score, neg_score]).numpy()
283             labels = torch.cat(
284                 [torch.ones(pos_score.shape[0]), torch.zeros(neg_score.shape[0])]).numpy()
285             return roc_curve(labels, scores)
286
287         roc_result_gcn = get_roc(pos_score_gcn.detach().cpu(),
288                                  neg_score_gcn.detach().cpu())
289         fpr_gcn, tpr_gcn, threshold_gcn = roc_result_gcn
290
291         roc_result_sage = get_roc(pos_score_sage.detach().cpu(),
292                                   neg_score_sage.detach().cpu())
293         fpr_sage, tpr_sage, threshold_sage = roc_result_sage
294
295         # --------------save to csv--------------
296         fpr_gcn_df = pd.DataFrame(fpr_gcn, columns=["fpr_gcn"])
297         tpr_gcn_df = pd.DataFrame(tpr_gcn, columns=["tpr_gcn"])
298         fpr_gcn_df.to_csv('fpr_gcn.csv', index=False)
299         tpr_gcn_df.to_csv('tpr_gcn.csv', index=False)
300
301         fpr_sage_df = pd.DataFrame(fpr_sage, columns=["fpr_sage"])
302         tpr_sage_df = pd.DataFrame(tpr_sage, columns=["tpr_sage"])
303         fpr_sage_df.to_csv('fpr_sage.csv', index=False)
304         tpr_sage_df.to_csv('tpr_sage.csv', index=False)
305
306         plt.figure()
307         plt.title(
308             f'ROC plot')
```

```
309         plt.plot(fpr_gcn, tpr_gcn, marker='.',
310                  label='GCN Link Prediction (AUROC GCN = %0.3f)' % auc_gcn)
311         plt.plot(fpr_sage, tpr_sage, marker='.',
312                  label='GraphSage Link Prediction (AUROC GraphSage = %0.3f)' % auc_sage)
313         plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r', alpha=.8)
314 
315         plt.xlabel('False Positive Rate')
316         plt.ylabel('True Positive Rate')
317         plt.legend()
318         plt.savefig(f'ROC plot')
319 
320 
321 procs = []
322 if __name__ == '__main__':
323     for rank in range(num_gpus):
324         ctx = mp.get_context("spawn")
325         p = ctx.Process(target=main, args=(rank, num_gpus))
326         p.start()
327         procs.append(p)
328     for p in procs:
329         p.join()
```