UNIVERISTY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Advanced Informatics and Computing Systems - Software Development and Artificial Intelligence»

ΠΜΣ «Προηγμένα Συστήματα Πληροφορικής – Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Bitcoin Address Classification using Unsupervised Machine Learning** Ταξινόμηση διευθύνσεων Bitcoin χρησιμοποιώντας μη εποπτευόμενη μηχανική εκμάθηση
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**STAMATIOU ANGELOS** ΣΤΑΜΑΤΙΟΥ ΑΓΓΕΛΟΣ
Father's name: Πατρώνυμο:	**PANAGIS** ΠΑΝΑΓΗΣ
Student's ID No: Αριθμός Μητρώου:	**ΜΠΣΠ/19046**
Supervisor: Επιβλέπων:	**Constantinos Patsakis, Associate Professor** Κωνσταντίνος Πατσάκης, Αναπληρωτής Καθηγητής

Ιούλιος 2021/ July 2021

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Constantinos Patsakis
Associate Professor

Κωνσταντίνος Πατσάκης
Αναπληρωτής Καθηγητής

Efthmios Alepis
Associate Professor

Ευθύμιος Αλέπης
Αναπληρωτής Καθηγητής

Evangelos Sakkopoulos
Assistant Professor

Ευάγγελος Σακκόπουλος
Επίκουρος Καθηγητής

Abstract

Το Bitcoin είναι ένα ψηφιακό κρυπτονόμισμα, που παρουσιάστηκε το 2008 από τον Satoshi Nakamoto, παρέχοντας ψευδοανωνυμία στης χρήστες του. Εφόσον τα δεδομένα του Bitcoin Blockchain είναι διαθέσιμα δημοσίως, οι συναλλαγές του μπορούν να πάρουν τη μορφή ενός κατευθυνόμενου γράφου, για εις βάθος ανάλυση. Η παρούσα διατριβή παρουσιάζει μια νέα προσέγγιση για τη μείωση της ανωνυμίας που παρέχεται, χρησιμοποιώντας μη-επιβλεπόμενη μάθηση στον γράφο των συναλλαγών. Μέσω της εκμάθησης αναπαράστασης κόμβου, τα χαρακτηριστικά του κόμβου μπορούν να εξαχθούν και να χρησιμοποιηθούν από έναν Logistic Regression Classifier, για να προβλέψει την ετικέτα κάθε κόμβου του γράφου. Για να απλοποιηθεί η πρόσβαση στα δεδομένα, τα δεδομένα του blockchain εισήχθησαν σε μια βάση δεδομένων MySQL. Η απόδοση της πλήρους προτεινόμενης λύσης αξιολογήθηκε, εκτελώντας τον ταξινομητή σε ένα υποσύνολο των δεδομένων του blockchain, επιτυγχάνοντας μέγιστη ακρίβεια 76.39%.

Bitcoin is a decentralized digital cryptocurrency, introduced in 2008 by Satoshi Nakamoto, providing pseudonymity to its users. Since Bitcoin blockchain data is publicly available, transactions can be modeled to a directed graph, for further analysis. This dissertation presents a novel approach to reduce the anonymity provided, by using Unsupervised Machine Learning on the transactions graph. By using node representation learning, node features can be extracted and used by a Logistic Regression Classifier to predict the label of each graph node. To simplify data access, blockchain data was imported to a MySQL Database. Performance of the complete proposed solution was evaluated, by executing the classifier on a sub-set of the blockchain data, achieving a maximum accuracy of 76.39%.

Acknowledgments

First and foremost, I would like to express my gratitude to my supervisor, Dr. Konstantinos Patsakis, for his assistance and guidance during the process of writing this dissertation. In addition, I would like to express my gratitude to my family and friends, for their continued support and motivation as I pursue my academic goals.

Contents

1. Introduction

Bitcoin is a digital cryptocurrency introduced in 2008 [1], featuring a publicly accessible distributed ledger. Bitcoin has attracted the attention of researchers from a variety of fields, gaining widespread popularity due to its unique characteristics, such as the lack of a centralized authority [3] and high-level degree of anonymity.

Since its release in 2009 [2], Bitcoin blockchain has reached more than 300 GB in size [31], becoming a challenge for researchers to perform analytical tasks on its data. Bitcoin transactions can be modeled as a directed graph, on which graph analysis can be executed. By using an existing labeling system [28], the performance of Machine Learning tasks on the graph can be explored. This dissertation aims to provide a complete solution on how to perform a classification task using unsupervised Machine Learning algorithms on Bitcoin blockchain data stored in a MySQL Database.

The rest of the dissertation is structured as follows. Chapter 2 presents an overview of the Bitcoin blockchain, MySQL Databases and the StellarGraph [7] Python library. Chapter 3 propounds in detail the suggested solution implementation, describing the parsing of Bitcoin blockchain data, imported to the Database and performing the unsupervised Machine Learning task. Chapter 4 analyzes the performance evaluation conducted for the proposed solution and challenges raised during development. Chapter 5 discusses the evaluation results and considers future work.

2. Background

This chapter presents the technologies that were utilized in the development of the approach, such as the Bitcoin blockchain, MySQL and the StellarGraph Python library. These technologies are further described in the next sections.

2.1 Bitcoin

Bitcoin is a decentralized cryptocurrency, developed by an unknow person or group of people using the pseudonym of Satoshi Nakamoto [1]. The actual Bitcoin blockchain network implementation was released as an open-source software in 2009 [2], enabling users to perform peer-to-peer transactions, without the need for intermediaries [3]. Transactions are verified by the network nodes and are publicly available through a distributed ledger, called a blockchain. The following sub-sections include the technical background of the Bitcoin blockchain, required for the understanding of later chapters in this dissertation, based on the book "Mastering Bitcoin" [2].

2.1.1 Blockchain

The Bitcoin blockchain is a public ledger that stores all validated Bitcoin transactions. It is implemented as an ordered back-linked list of blocks. Each block is linked to the previous block by including the hash of the previous block in its header and must conform to a specific pattern of, e.g., trailing zeros. This inclusion affects the block hash of the current block. Changing a block requires changing all the following blocks, a task with enormous computation requirements as the blockchain grows. This concept provides the immutability of the ledger, a key feature of blockchain security. In fact, the immutability, decentralization, and auditability make blockchains an extremely attractive technology for building new solutions [34].

Block

The data structure of a block is described in Table 1. The *Block Header* field contains the metadata of the block, detailed in Table 2, used for the block hash calculation. The calculation of the hash is executed by hashing the *Block Header* twice, using the SHA256 algorithm, resulting in a 32-byte block hash, uniquely identifying the block in the blockchain. An additional identification method is its position in the blockchain, called the *block height*, indicating the distance of the block from the genesis block (first block of the chain). Since two or more blocks can compete for the same position in the blockchain during a fork in the chain, the block height is not a unique identifier.

Table 1: Bitcoin block structure

Field	Description	Size
Block Size	The size of the block	4 bytes
Block Header	Metadata of the block	80 bytes
Transaction Counter	How many transactions follow	1-9 bytes (VarInt)
Transactions	The transactions recorded in this block	Variable

Table 2: Bitcoin block header structure

Field	Description	Size
Version	A version number to track software/protocol upgrades	4 bytes
Previous Block Hash	A reference to the hash of the previous (parent) block in the chain	32 bytes
Merkle Root	A hash of the root of the merkle tree of this block’s transactions	32 bytes

Timestamp	The approximate creation time of this block (seconds from Unix Epoch)	4 bytes
Difficulty Target	The proof-of-work algorithm difficulty target for this block	4 bytes
Nonce	A counter used for the proof-of-work algorithm	4 bytes

Addresses

A Bitcoin address is an identifier of 27-34 alphanumeric characters, beginning with the number 1, 3 or bc1, acting in the same way as a bank account number. Users can share their address with other people to allow them to exchange Bitcoins. There are currently three address formats in use in Bitcoin *Mainnet*:

1. **Pay to Public Key Hash (P2PKH)** or Legacy Address Format, starting with the number **1.** Example: 17VZNX1SN5NtKa8UQFxwQbFeFc3iqRYhem
2. **Pay to Script Hash (P2SH)** or Compatibility Address Format, starting with the number **3**. Example: 3EktnHQD7RiAE6uzMj2ZifT9YgRrkSgzQX
3. **Bech32** or Segwit Address Format, starting with "bc1". Example: bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4

Transactions

A transaction is the data structure, described in Table 3, holding the information of a Bitcoin transfer form one or more source addresses to one or more destination addresses.

Table 3: Bitcoin transaction structure

Field	**Description**	**Size**
Version	Specifies which rules this transaction follows.	4 bytes
Input Counter	Defines how many inputs are included.	1–9 bytes (VarInt)
Inputs	One or more transaction inputs.	Variable
Output Counter	Defines how many outputs are included.	1–9 bytes (VarInt)
Outputs	One or more transaction outputs.	Variable
Locktime	A Unix timestamp or block number.	4 bytes

Each transaction consumes and produces spendable chunks of bitcoin, called unspent transaction outputs (UTXO)**.** Figure 1 depicts a simplified Bitcoin transaction example, including the fields relevant for this dissertation, described in Table 4.

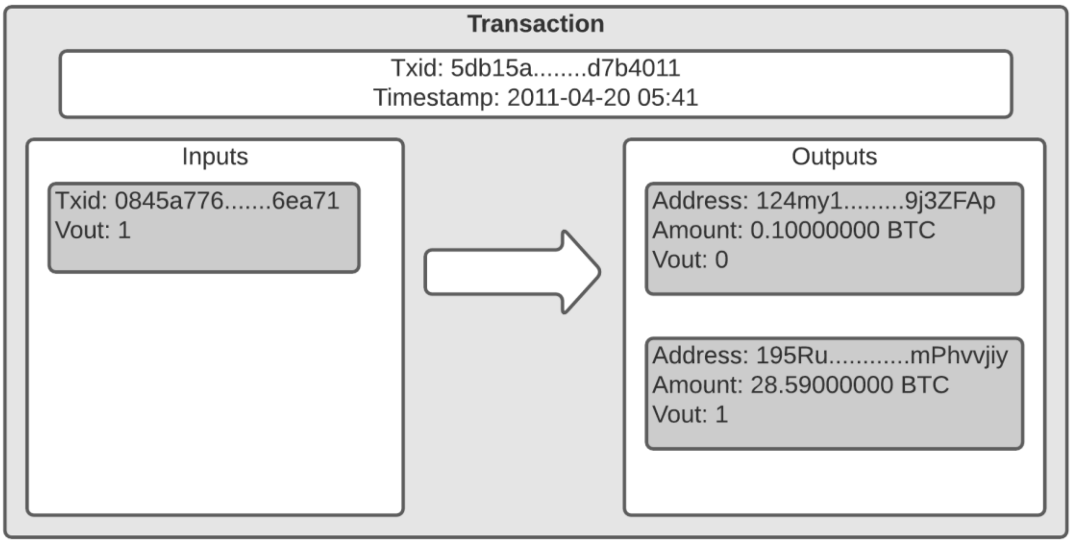

Figure 1: Simplified Bitcoin transaction example

Table 4: Simplified Bitcoin transaction structure

Field	**Description**	**Size**
Txid	Transaction hash	32 bytes
Timestamp	Creation time of block containing the transaction	4 bytes
Address	Bitcoin wallet address	32 bytes
Amount	Bitcoin value in satoshis (10-8 bitcoin)	8 bytes
Vout	Output index	4 bytes

2.1.2 Mining

The process of adding new coin generation, added to the money supply, is known as mining. Miners independently verify and include new transactions, propagated on the Bitcoin network, into new blocks during this process. Next step of the process is solving a computationally intensive cryptographic problem, known as Proof-of-Work (PoW), to verify and transmit the newly created block on the Bitcoin network. Mining secures the system and allows for network-wide consensus without the need of a central authority. A miner who successfully adds a block to the blockchain is rewarded with a block reward, along with all transaction fees included in the block.

Proof-of-Work Algorithm

A hashing algorithm converts a data input and into a fixed-length deterministic output, a digital fingerprint of the input. The output hash will always be the same, for any given input, and can be easily computed and verified by anyone using the same hash algorithm. Bitcoin's block hash is computed by hashing block's header data through SHA-256 repeatedly, while incrementing the nonce field, until the hash is smaller or equal to the target difficulty. For further details regarding the consensus mechanism of Bitcoin, the interested reader may refer to "The Bitcoin Backbone Protocol: Analysis and Application" [35].

2.1.3 Bitcoin Core Client

Bitcoin Core Client is an open-source project maintaining the Bitcoin client software, based on Satoshi Nakamoto's original Bitcoin client. It includes both "full-node" software, maintaining a full copy of the blockchain, as well as a Bitcoin wallet to execute transactions [4]. Client uses blk*.dat files, to store the blocks as soon as they are received. The data in blk.dat files is stored in binary format, with each new block being appended to the file's end. Figure 2 depicts the genesis block, by reading the first 293 bytes of blk00000.dat file. For detailed information refer to Bitcoin Wiki: Protocol Documentation [5].

```
f9beb4d91d010000010000000000000000000000000000000000000000000000000000000000000000000000003ba3edfd7a7b12b27ac72c3e67768f617f
c81bc3888a51323a9fb8aa4b1e5e4a29ab5f49ffff001d1dac2b7c0101000000010000000000000000000000000000000000000000000000000000000000
00000000ffffffff4d04ffff001d0104455468652054696d65732030332f4a616e2f32303039204368616e63656c6c6f72206f6e206272696e6b206f66
207365636f6e64206261696c6f757420666f722062616e6b73ffffffff0100f2052a01000000434104678afdb0fe5548271967f1a67130b7105cd6a828
e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5fac00000000
```

Figure 2: Bitcoin genesis block

2.2 MySQL

MySQL is an open-source relational database management system (RDBMS), developed, distributed and supported by Oracle Corporation [6]. Relational databases organize data to tables containing data types related to each other, structuring the data. Using the SQL language, programmers can build, modify, and extract data from the relational database. Users have the option of using MySQL as a free open-source product under the GNU General Public License or purchasing a standard commercial license from Oracle. Main features provided are as follows:

- **Ease of Management** – The software is simple to install and uses an event scheduler to automatically schedule activities.
- **Robust Transactional Support** – Holds the ACID property (Atomicity, Consistency, Isolation, and Durability), as well as allowing distributed multi-version support.
- **Comprehensive Application Development** – MySQL comes with plugin libraries that allow you to integrate the database into any program. For application creation, it also supports stored procedures, triggers, functions, views, and many other features.
- **High Performance** – With distinct memory caches and table index partitioning, it provides quick load utilities.
- **Low Total Cost of Ownership** – This reduces licensing costs and hardware expenditures.
- **Secure Data Protection** – MySQL has robust processes in place to ensure that only authorized users have access to databases.
- **High Availability** – MySQL supports high-performance master/slave replication as well as server clustering.
- **Scalability & Flexibility** – MySQL allows you to run deeply embedded applications and create data warehouses that can store massive amounts of data.

2.3 StellarGraph

StellarGraph is a Python library for solving machine learning tasks on graphs and networks, like Representation learning, Classification and Link prediction [7]. The library provides cutting-edge graph machine learning algorithms, simplifying pattern discovery on graph-structured data, supporting analysis for many kinds of graphs (e.g., homogenous, heterogenous, etc.). StellarGraph is built on TensorFlow2 and its Keras high-level API, along with Pandas and NumPy, resulting in a user-friendly, flexible and expandable library. Current version includes the graph machine learning algorithms described in Table 5.

Table 5: StellarGraph supported algorithms

Algorithm	**Description**
GraphSAGE	Supports supervised as well as unsupervised representation learning, node classification/regression, and link prediction for homogeneous networks [χ8].
HinSAGE	Extension of GraphSAGE algorithm for heterogeneous networks [33].
attri2vec	Supports node representation learning, node classification, and out-of-sample node link prediction for homogeneous graphs with node attributes [9].
Graph Attention Network (GAT)	The GAT algorithm supports representation learning and node classification for homogeneous graphs [10].
Graph Convolutional Network (GCN)	The GCN algorithm supports representation learning and node classification for homogeneous graphs [11].
Cluster Graph Convolutional Network (Cluster-GCN)	An extension of the GCN algorithm supporting representation learning and node classification for homogeneous graphs [12].
Simplified Graph Convolutional network (SGC)	The SGC network algorithm supports representation learning and node classification for homogeneous graphs [13].
(Approximate) Personalized Propagation of Neural Predictions (PPNP/APPNP)	The (A)PPNP algorithm supports fast and scalable representation learning and node classification for attributed homogeneous graphs [14].
Node2Vec	The Node2Vec and Deepwalk algorithms perform unsupervised representation learning for homogeneous networks, taking into account network structure while ignoring node attributes [15].
Metapath2Vec	The metapath2vec algorithm performs unsupervised, metapath-guided representation learning for heterogeneous networks, taking into account network structure while ignoring node attributes [16].
Relational Graph Convolutional Network	The RGCN algorithm performs semi-supervised learning for node representation and node classification on knowledge graphs [17].
ComplEx	The ComplEx algorithm computes embeddings for nodes (entities) and edge types (relations) in knowledge graphs, and can use these for link prediction [18].
GraphWave	GraphWave calculates unsupervised structural embeddings via wavelet diffusion through the graph [19].
Supervised Graph Classification	A model for supervised graph classification based on GCN [11] layers and mean pooling readout.
Watch Your Step	The Watch Your Step algorithm computes node embeddings by using adjacency powers to simulate expected random walks [20].
Deep Graph Infomax	Deep Graph Infomax trains unsupervised GNNs to maximize the shared information between node level and graph level features [21].
Continuous-Time Dynamic Network Embeddings (CTDNE)	Supports time-respecting random walks which can be used in a similar way as in Node2Vec for unsupervised representation learning [22].
DistMult	The DistMult algorithm computes embeddings for nodes (entities) and edge types (relations) in knowledge graphs, and can use these for link prediction [23].

DGCNN	The Deep Graph Convolutional Neural Network (DGCNN) algorithm for supervised graph classification [24].
TGCN	The GCN_LSTM model in StellarGraph follows the Temporal Graph Convolutional Network architecture proposed in the TGCN paper with a few enhancements in the layers architecture [25].

For the execution of the Address classification task using unsupervised Machine Learning, Deep Graph Infomax [21] with Graph Convolutional Network (GCN) [11] algorithm was utilized for the nod representation learning. Deep Graph Infomax makes use of graph convolutional network architectures, to maximize mutual information between patch representations and corresponding high-level summaries of graphs. The learned patch representations summarize subgraphs centered around nodes of interest, reused for downstream node-wise learning tasks.

3. Processing Bitcoin Data

This chapter describes the design and implementation of the proposed solution, allowing the execution of the Bitcoin Address classification task. Section 3.1 describes the complete solution, including an overview of the architecture, database, and generated Address graph. The implementation of the proposed solution's various components is presented in detail in Section 3.2.

3.1 Solution Design

3.1.1 Solution Architecture

The solution architecture high-level view is illustrated in Figure 3. Script *parser.py* parses blk*.dat files of the Bitcoin blockchain and produces files containing the fields relevant to this dissertation, as described in Table 4. Script *reader.py* parses the output files of *parser.py* script and imports retrieved information to the Database. This streamline method was chosen, to simplify parsing and importing steps, and enabling batch execution. Imported data are then processed by *transactions_retriever.py* script, which generates the execution dataset for the *analyzer.py* script. Finally, *analyzer.py* script performs the Address classification task using unsupervised Machine Learning.

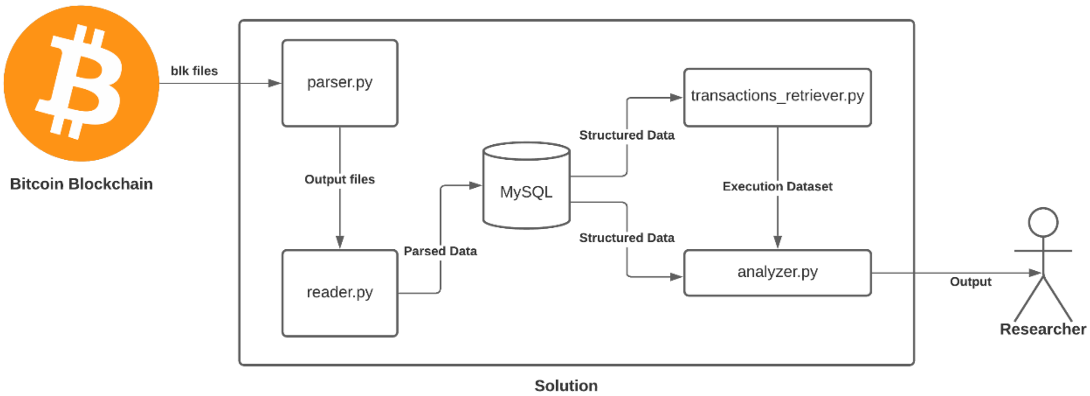

Figure 3: Solution architecture

3.1.2 Database

MySQL offers a free and open-source relational database management system, allowing full control and customization to satisfy the solution's requirements. Due to the massive dataset, row compression was enabled, reducing the total disk size as presented in Table 6. Additionally, field indexing was included, for faster query execution, drastically improving data retrieval time. Figure 4 depicts the Database schema, populated by the parsed data of *parser.py* script.

Table 6: Database row compression

Table	Original Size	Compressed Size	Compression Ratio
\`tx\`	119 GB	77.2 GB	1.54
\`tx_in\`	530 GB	291 GB	1.82
\`tx_out\`	480 GB	284 GB	1.69
Overall	1129 GB	652 GB	1.73

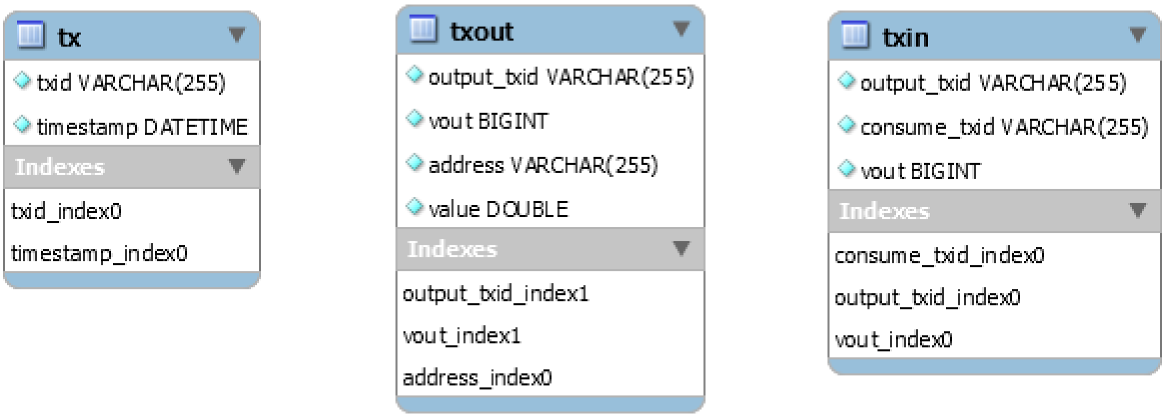

Figure 4: Database schema

3.1.3 Address Graph

To perform the Address classification task, a graph is generated to represent the relations between Database records. Each transaction can be described as a graph node, connected with the inputs and outputs addresses. An input address sends some Bitcoin to the transaction and an output address receives it. This results to a Directed Graph, depicted in Figure 5, with two node types, *transaction* in yellow and *address* in orange. As link weight, the amount transferred between the nodes is used. Additionally, each link holds the transaction timestamp.

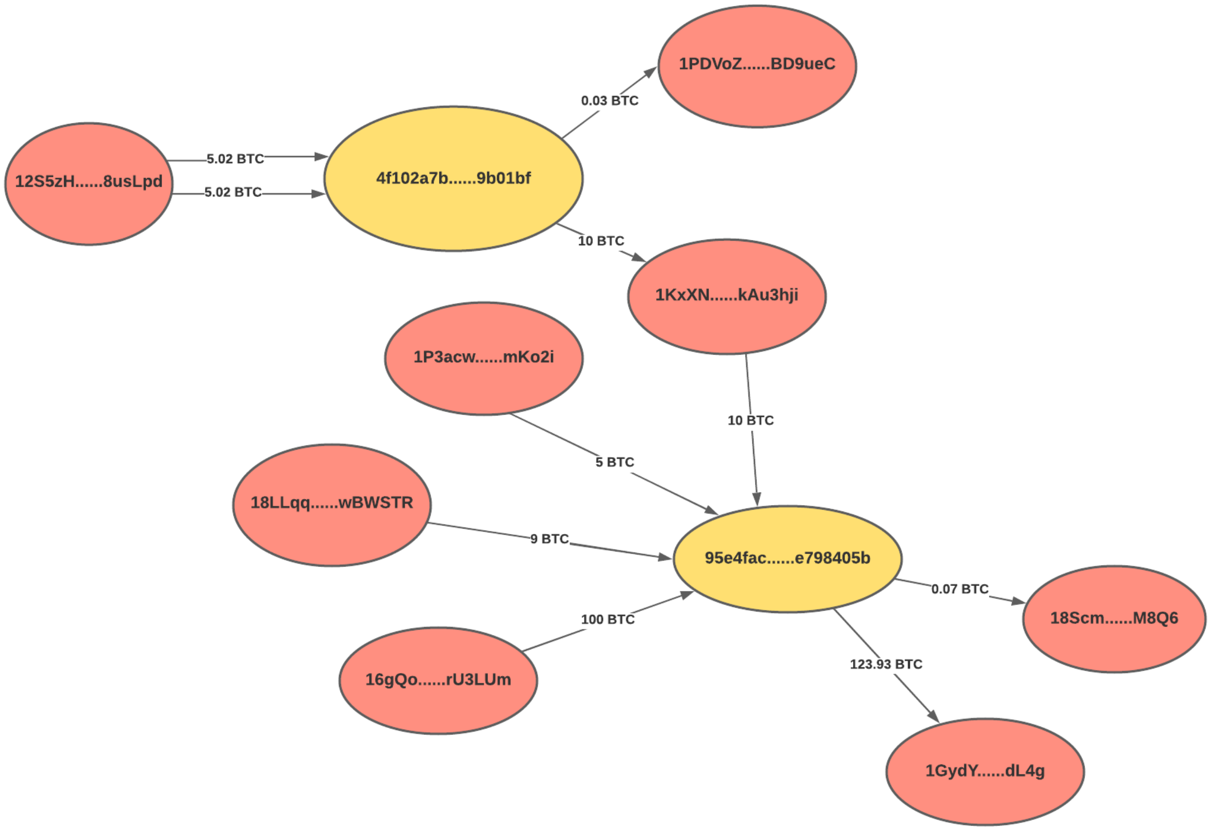

Figure 5: Bitcoin transactions graph

3.2 Scripts implementation

3.2.1 parser.py

To retrieve the Bitcoin transactions information required for this dissertation, a modified version of Blockchain parser by Denis Leonov [26] was created. In this alteration, default output was replaced by the information of the transactions included in the file. This was achieved by employing btcpy [27], a python library providing tools to handle Bitcoin data structures. After extracting each transaction's RawTX, the hex string of the transaction, function create_record deserializes the transaction and appends the extracted information to the output, as shown in Figure 6.

```
def create_record(resList, RawTX, Timestamp):
    tx = Transaction.unhexlify(RawTX)
    resList.append('tx,' + str(tx.txid) + ',' + Timestamp + ';')
    for txin in tx.ins:
        resList.append('txin,' + str(txin.txid)
                            + ',' + str(tx.txid)
                            + ',' + str(txin.txout) + ';')
    for txout in tx.outs:
        resList.append('txout,' + str(tx.txid)
                            + ',' + str(txout.n) + ','
                            + str(txout.address()) + ','
                            + str(Decimal(txout.value) * Constants.get('from_unit')) + ';')
```

Figure 6: parser.py/create_record

This script produces a CSV file for each blk*.dat file in parses. Output files consist of lines containing the entities extracted from each deserialized transaction, described in Table 4. Each line starts with the entity type, followed by the information described in Table 7. An output example is depicted in Figure 7.

Table 7: parser.py output format

Entity type	Information
tx	txid, timestamp
txin	output_txid, consume_txid, vout
txout	consume_txid, vout, address, value

```
tx,84c9e6cb8a20e56fb8cc1ff29cb904114df76ba1f8fa8c9951c2a06b9de76dd4,2012-03-06T18:13:30;
txin,0000000000000000000000000000000000000000000000000000000000000000,84c9e6cb8a20e56fb8cc1ff29cb904114df76ba1f8fa8c9951c2a06b9de76dd4,4294967295;
txout,84c9e6cb8a20e56fb8cc1ff29cb904114df76ba1f8fa8c9951c2a06b9de76dd4,0,None,50.00750000;
tx,59a6642e1815268a4ddfdc8ebb83dd8cb2568e60e44d20447cf8f923d1450ca0,2012-03-06T18:13:30;
txin,252ceda6f57efdf9fa06dfe189fc5e91583e29feff8390822813fc20902c3f30,59a6642e1815268a4ddfdc8ebb83dd8cb2568e60e44d20447cf8f923d1450ca0,1;
txin,3896a8b26d9ed3ceb3e1d6d5c7f57e00af3938cc373a9a8987b54e69b6b16f63,59a6642e1815268a4ddfdc8ebb83dd8cb2568e60e44d20447cf8f923d1450ca0,0;
txout,59a6642e1815268a4ddfdc8ebb83dd8cb2568e60e44d20447cf8f923d1450ca0,0,1GdYw7eqdzs5ACqta8EoR3qmevA4c5TnRf,1.36750000;
txout,59a6642e1815268a4ddfdc8ebb83dd8cb2568e60e44d20447cf8f923d1450ca0,1,1GQxHZth5UjcAswSvBsTiPR9RiknfB3Q9A,38.00000000;
tx,ee46d607880e493b84a4ef2f22563254ec600ba30c7ce45d03534e02b37de25a,2012-03-06T18:13:30;
txin,ec6e00dd893a8396ad7bd251bd4328f4be2029fbd643cdcb4e1e2576858da273,ee46d607880e493b84a4ef2f22563254ec600ba30c7ce45d03534e02b37de25a,1;
txout,ee46d607880e493b84a4ef2f22563254ec600ba30c7ce45d03534e02b37de25a,0,1MsrUrxskCwtamxFSPcdK5qSKzfZaqiyjn,674.76866771;
txout,ee46d607880e493b84a4ef2f22563254ec600ba30c7ce45d03534e02b37de25a,1,1JCs3Zwj2WVQaJHS3TYq6aSZ7BPGJJTXth,825.23133229;
tx,da2efaabc15130bd7e34d9c9195b39bfaf3ba2d7f8a6a3d328bee668e7f1cec2,2012-03-06T18:13:30;
txin,492034702dbe876ceb5a59b398baa58b8c5f152ef9e775ae0f514ac1e668bfcc,da2efaabc15130bd7e34d9c9195b39bfaf3ba2d7f8a6a3d328bee668e7f1cec2,1;
txin,b3bd1faf7a50c933fa9e018e54a7820f58ccd24cbee8dde4d780bd1a1832a8d8,da2efaabc15130bd7e34d9c9195b39bfaf3ba2d7f8a6a3d328bee668e7f1cec2,1;
txout,da2efaabc15130bd7e34d9c9195b39bfaf3ba2d7f8a6a3d328bee668e7f1cec2,0,1ASXzYFiHysNaE496ovpxapz6Z7qtCmEEs,0.01000000;
txout,da2efaabc15130bd7e34d9c9195b39bfaf3ba2d7f8a6a3d328bee668e7f1cec2,1,1GfsndtFrQobiJfcGszqkCg8Qt3EqJrr5G,150.00000000;
tx,8a9566eaf4f96bf76743fc35640001e9959626150083cb66da985df1f3710207,2012-03-06T18:13:30;
txin,996b4fb1dbfef039d82ea2d6eeb77cd0585e7f46cf1f8713bb48b86028a5af0b,8a9566eaf4f96bf76743fc35640001e9959626150083cb66da985df1f3710207,0;
txin,4445dc8ce0301655450b62975f49580b964fc4af70cf980a6f3a00baa7eb9cb2,8a9566eaf4f96bf76743fc35640001e9959626150083cb66da985df1f3710207,0;
txout,8a9566eaf4f96bf76743fc35640001e9959626150083cb66da985df1f3710207,0,14Jfa9TaErKiuVRBu8z7rUkq2XeQWsDH3S,15.36000000;
txout,8a9566eaf4f96bf76743fc35640001e9959626150083cb66da985df1f3710207,1,1DgMxZfBWUPjwd2pgJskC4AQb5o4Ek36Kr,4.35949026;
tx,43620a0ba50e02786d7a8d49d237e2bb4a6e1668f7809653628eba4d0ebab250,2012-03-06T18:13:30;
txin,cd83928a3eb4db415e411aa398d69173874d48afa21b7452bea3a18fcc84cd32,43620a0ba50e02786d7a8d49d237e2bb4a6e1668f7809653628eba4d0ebab250,1;
txout,43620a0ba50e02786d7a8d49d237e2bb4a6e1668f7809653628eba4d0ebab250,0,19b3ixXe4BGUB3P4tsQTcL6T3pqAxDP8pC,5.00000000;
tx,398825c31fd12b5146e0f13bf269e1b1dd083c51929a1283eaf0e5bb46acf44e,2012-03-06T18:13:30;
txin,c5f330a8fac1c230a8ba55e766555b2f0639f60c649e77123f637c0eb50427fe,398825c31fd12b5146e0f13bf269e1b1dd083c51929a1283eaf0e5bb46acf44e,1;
txin,cfa2b9a1348643fc6767c135aac3a603a01374603d0f4a517a183a843e03c0a9,398825c31fd12b5146e0f13bf269e1b1dd083c51929a1283eaf0e5bb46acf44e,1;
txout,398825c31fd12b5146e0f13bf269e1b1dd083c51929a1283eaf0e5bb46acf44e,0,13V2RdW8WLTkbrYBx8VoPwJ7bGVr4e2J44,275.05000000;
tx,c492c340b6a3546546a4f2fc16c988e9d46e9be6998d8eadec7bf2a1c04152d6,2012-03-06T18:13:30;
txin,be7cabb98b6f4745cb912665299ab60bde5fb7ca5a813c7ae495b53911e7852f,c492c340b6a3546546a4f2fc16c988e9d46e9be6998d8eadec7bf2a1c04152d6,0;
txout,c492c340b6a3546546a4f2fc16c988e9d46e9be6998d8eadec7bf2a1c04152d6,0,1DfkjPbTGh6nEwZDDLTjyT7ykfYjFpD4pf,1.87000000;
txout,c492c340b6a3546546a4f2fc16c988e9d46e9be6998d8eadec7bf2a1c04152d6,1,1JL6ocVmsqRJwhpM8hEiLwGngCQmqQPBaL,0.03000000;
tx,69501ff91b2dd12a18d3e43b7094a0c27183e6c006acb4bdb14c2f42930b2bd1,2012-03-06T18:13:30;
txin,ae323c45dcc9a3e53617957a5e32eb36bc14589d16593f552a0f6678676ab860,69501ff91b2dd12a18d3e43b7094a0c27183e6c006acb4bdb14c2f42930b2bd1,1;
txin,889af92748fbf3e4daa2fb76cc59151694716928ba0c5d8a932c476a33605d92,69501ff91b2dd12a18d3e43b7094a0c27183e6c006acb4bdb14c2f42930b2bd1,1;
txin,ae29d5102f617e1c346b73fadc36cf9d08e79a39c05153a5acfd539929a74d19,69501ff91b2dd12a18d3e43b7094a0c27183e6c006acb4bdb14c2f42930b2bd1,1;
txin,78e0891f359472014d32d83ee1674e91555b2e8692840e69f69f215616ca8144,69501ff91b2dd12a18d3e43b7094a0c27183e6c006acb4bdb14c2f42930b2bd1,1;
txin,2ef7702c31772b1e18be01b7437165c36422564da54b556e638a0a26393cf21f,69501ff91b2dd12a18d3e43b7094a0c27183e6c006acb4bdb14c2f42930b2bd1,0;
txin,c3f1da84e1990eaf9daae5db5df13cb8e1df2b8b19df58793b26a2efb43441eb,69501ff91b2dd12a18d3e43b7094a0c27183e6c006acb4bdb14c2f42930b2bd1,0;
txout,69501ff91b2dd12a18d3e43b7094a0c27183e6c006acb4bdb14c2f42930b2bd1,0,1CxXZZXaezP2EGvscSjpFtQALpi3V8zYJ1,0.01147407;
txout,69501ff91b2dd12a18d3e43b7094a0c27183e6c006acb4bdb14c2f42930b2bd1,1,1FxkP2JXY5DZpxwBa3L21g5BfkuYAnCVZj,73.00000000;
```

Figure 7: parser.py output example

3.2.2 reader.py

As *parser.py* execution has been concluded, output files must be parsed to import extracted information to the Database. Python script *reader.py* parses the output files and generates the MySQL Database records, further described in the following sections.

main script

When executing the script, a Database connection is created using the utility function init_database. Then, each file with index in specific range is parsed by function parse_file. Finally, the Database connection is terminated using the utility function close_database**.**

```python
dir = 'e:/Blockchain Analysis/blockchain-parser-master/results/'
db = init_database()
for x in range(2364, 2400):
    file = dir + 'blk' + f"{x:05d}" + '.txt'
    parse_file(db, file)
close_database(db)
```

Figure 8: reader.py main

parse_file function

This function parses a file and maps extracted data to TX, TXIN and TXOUT class objects. These classes represent the corresponding Database tables, including the record creation query for each one, depicted in Figure 10. After parsing is concluded, all parsed records are inserted to the Database, using batching commit for optimization.

```python
def parse_file(db, file):
    start_time = time.time()
    print ('Start reading file ' + str(file) + ' at: ' + time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime(start_time)))
    with open(file, newline='') as f:
        reader = csv.reader(f)
        records = list(reader)

    tx_list = []
    txin_list = []
    txout_list = []
    for record in records:
        if record[0] == 'tx':
            tx = TX(record[1], record[2].replace(';', ''))
            tx_list.append(tx)
        elif record[0] == 'txin':
            txin = TXIN(record[1], record[2], record[3].replace(';', ''))
            txin_list.append(txin)
        else:
            txout = TXOUT(record[1], record[2], record[3], record[4].replace(';', ''))
            txout_list.append(txout)

    commit_counter = 0;
    for tx in tx_list:
        tx.insert_record(db)
        if (commit_counter == 10000):
            db.commit();
            commit_counter = 0
        else:
            commit_counter += 1
    db.commit()

    for txin in txin_list:
        txin.insert_record(db)
        if (commit_counter == 10000):
            db.commit();
            commit_counter = 0
        else:
            commit_counter += 1
    db.commit()

    for txout in txout_list:
        txout.insert_record(db)
        if (commit_counter == 10000):
            db.commit();
            commit_counter = 0
        else:
            commit_counter += 1
    db.commit()

    print ('Finished reading file ' + str(file) + '! Elapsed time: ' + time.strftime("%H:%M:%S", time.gmtime(time.time() - start_time)))
```

Figure 9: reader.py/parse_file

```python
# Class mapping `tx` DB records.
class TX:
    def __init__(self, txid, timestamp):
        self.txid = txid
        self.timestamp = timestamp

    def __str__(self):
        return 'TX=[txid={0}, timestamp={1}]'.format(self.txid, self.timestamp)

    def insert_record(self, db):
        cursor = db.cursor()
        cursor.execute('INSERT INTO tx VALUES(\'{0}\', \'{1}\')'.format(self.txid, self.timestamp))

# Class mapping `txin` DB records.
class TXIN:
    def __init__(self, output_txid, consume_txid, vout):
        self.output_txid = output_txid
        self.consume_txid = consume_txid
        self.vout = vout

    def __str__(self):
        return 'TXIN=[output_txid={0}, consume_txid={1}, vout={2}]'.format(self.output_txid, self.consume_txid, self.vout)

    def insert_record(self, db):
        cursor = db.cursor()
        cursor.execute('INSERT INTO txin VALUES(\'{0}\', \'{1}\', \'{2}\')'.format(self.output_txid, self.consume_txid, self.vout))

# Class mapping `txout` DB records.
class TXOUT:
    def __init__(self, output_txid, vout, address, value):
        self.output_txid = output_txid
        self.vout = vout
        self.address = address
        self.value = value

    def __str__(self):
        return 'TXOUT=[output_txid={0}, vout={1}, address={2}, value={3}]'.format(self.output_txid, self.vout, self.address, self.value)

    def insert_record(self, db):
        cursor = db.cursor()
        cursor.execute('INSERT INTO txout VALUES(\'{0}\', \'{1}\', \'{2}\', \'{3}\')'.format(self.output_txid, self.vout, self.address, self.value))
```

Figure 10: reader.py/classes

init_database function

This utility function initializes a connection with the MySQL Database and creates the DB schema in case it is not present.

```
def init_database():
    db = mysql.connect(host='localhost', user='root', password='root')
    cursor = db.cursor()
    cursor.execute("CREATE DATABASE IF NOT EXISTS btc")
    db = mysql.connect(host='localhost', user='root', password='root', database='btc')
    cursor = db.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS tx (txid VARCHAR(255) NOT NULL, timestamp DATETIME NOT NULL)")
    cursor.execute("CREATE TABLE IF NOT EXISTS txin (output_txid VARCHAR(255) NOT NULL, consume_txid VARCHAR(255) NOT NULL, vout BIGINT NOT NULL)")
    cursor.execute("CREATE TABLE IF NOT EXISTS txout (output_txid VARCHAR(255) NOT NULL, vout BIGINT NOT NULL, address VARCHAR(255) NOT NULL, value DOUBLE NOT NULL)")
    return db;
```

Figure 11: reader.py/init_database

close_database function

This utility function closes an active connection to the Database.

```
def close_database(db):
    if db is not None and db.is_connected():
        db.close()
```

Figure 12: reader.py/close_database

3.2.3 Transactions_retriever.py

This Python script generates the execution dataset for the *analyzer.py* script. A random address sample is retrieved from the Entity-address dataset for 2010-2018 Bitcoin transactions [28], used in papers Characterizing Entities in the Bitcoin Blockchain [29] and A Probabilistic Model of the Bitcoin Blockchain [30]. Additionally, a dataset containing malicious addresses was kindly provided by the authors of “An Analysis of Bitcoin Laundry Services” [32], further enriching the dataset variety. For each address in the sample, all their transaction ids are retrieved from the Database, to create the execution dataset output files, further described in the following sections.

main script

When executing the script, each file in the original dataset is parsed using function read_csv_file, producing the random sample and the full addresses list. A database connection is initialized using the utility function init_database. For each produced sample, function execute_query retrieves all their transactions, appending them to the transaction set. After finishing records fetching, Database connection is terminated using utility function close_database. Finally, a CSV file containing the retrieved transactions is generated, along with a CSV file containing the full addresses list for each original dataset file, by utility function generate_csv_file.

```python
total_time = time.time()
logging.info('Retrieving address records...')
random_exchanges_addresses, exchanges_addresses = read_csv_file(EXCHANGES_ADDRESSES_FILE)
random_gambling_addresses, gambling_addresses = read_csv_file(GAMBLING_ADDRESSES_FILE)
random_historic_addresses, historic_addresses = read_csv_file(HISTORIC_ADDRESSES_FILE)
random_malicious_addresses, malicious_addresses = read_csv_file(MALICIOUS_ADDRESSES_FILE)
random_mining_addresses, mining_addresses = read_csv_file(MINING_ADDRESSES_FILE)
random_services_addresses, services_addresses = read_csv_file(SERVICES_ADDRESSES_FILE)

logging.info('Retrieving transaction records...')
db = init_database()
cursor = db.cursor()
transactions = set()
execute_query(transactions, cursor, TXIN_QUERY + str(random_exchanges_addresses).replace('[','(').replace(']',')'), 'TXIN_QUERY for random_exchanges_addresses...')
execute_query(transactions, cursor, TXOUT_QUERY + str(random_exchanges_addresses).replace('[','(').replace(']',')'), 'TXOUT_QUERY for random_exchanges_addresses...')
execute_query(transactions, cursor, TXIN_QUERY + str(random_gambling_addresses).replace('[','(').replace(']',')'), 'TXIN_QUERY for random_gambling_addresses...')
execute_query(transactions, cursor, TXOUT_QUERY + str(random_gambling_addresses).replace('[','(').replace(']',')'), 'TXOUT_QUERY for random_gambling_addresses...')
execute_query(transactions, cursor, TXIN_QUERY + str(random_historic_addresses).replace('[','(').replace(']',')'), 'TXIN_QUERY for random_historic_addresses...')
execute_query(transactions, cursor, TXOUT_QUERY + str(random_historic_addresses).replace('[','(').replace(']',')'), 'TXOUT_QUERY for random_historic_addresses...')
execute_query(transactions, cursor, TXIN_QUERY + str(random_malicious_addresses).replace('[','(').replace(']',')'), 'TXIN_QUERY for random_malicious_addresses...')
execute_query(transactions, cursor, TXOUT_QUERY + str(random_malicious_addresses).replace('[','(').replace(']',')'), 'TXOUT_QUERY for random_malicious_addresses...')
execute_query(transactions, cursor, TXIN_QUERY + str(random_mining_addresses).replace('[','(').replace(']',')'), 'TXIN_QUERY for random_mining_addresses...')
execute_query(transactions, cursor, TXOUT_QUERY + str(random_mining_addresses).replace('[','(').replace(']',')'), 'TXOUT_QUERY for random_mining_addresses...')
execute_query(transactions, cursor, TXIN_QUERY + str(random_services_addresses).replace('[','(').replace(']',')'), 'TXIN_QUERY for random_services_addresses...')
execute_query(transactions, cursor, TXOUT_QUERY + str(random_services_addresses).replace('[','(').replace(']',')'), 'TXOUT_QUERY for random_services_addresses...')
close_database(db, cursor);
logging.info('Finished retrieving transaction records! Total transactions: ' + str(len(transactions)))

generate_csv_file(TRANSACTIONS_CSV_FILE, 'txid', transactions)
generate_csv_file(EXCHANGES_ADDRESSES_CSV_FILE, 'address', exchanges_addresses)
generate_csv_file(GAMBLING_ADDRESSES_CSV_FILE, 'address', gambling_addresses)
generate_csv_file(HISTORIC_ADDRESSES_CSV_FILE, 'address', historic_addresses)
generate_csv_file(MALICIOUS_ADDRESSES_CSV_FILE, 'address', malicious_addresses)
generate_csv_file(MINING_ADDRESSES_CSV_FILE, 'address', mining_addresses)
generate_csv_file(SERVICES_ADDRESSES_CSV_FILE, 'address', services_addresses)
logging.info('Total Execution time: ' + time.strftime('%H:%M:%S', time.gmtime(time.time() - total_time)))
```

Figure 13: transactions_retriever.py/main

read_csv_file function

This function parses a CSV file, using the file configuration depicted in Figure 15, which identifies the file path, address position and address limit. Function produces a list containing all the parsed addresses, along with a random sample of them.

```python
def read_csv_file(file):
    logging.info('Retrieving addresses from csv: ' + file[0])
    logging.info('Addresses limit: ' + str(file[2]))
    addresses = set()
    with open(file[0]) as csv_file:
        csv_reader = csv.reader(csv_file)
        header = next(csv_reader)
        for row in csv_reader:
            if row[file[1]] not in addresses:
                addresses.add(row[file[1]])
    logging.info('Addresses found: ' + str(len(addresses)))
    random_addresses = random.sample(addresses, file[2]) if len(addresses) > file[2] else list(addresses)
    return random_addresses, addresses
```

Figure 14: transactions_retriever.py/read_csv_file

```python
# Original dataset files configuration.
# file = ['file_path', address_position, address_limit]
EXCHANGES_ADDRESSES_FILE = ['Addresses/Exchanges_full_detailed.csv', 5, 10]
GAMBLING_ADDRESSES_FILE = ['Addresses/Gambling_full_detailed.csv', 4, 10]
HISTORIC_ADDRESSES_FILE = ['Addresses/Historic_full_detailed.csv', 4, 10]
MALICIOUS_ADDRESSES_FILE = ['Addresses/Malicious_addresses.csv', 0, 300]
MINING_ADDRESSES_FILE = ['Addresses/Mining_full_detailed.csv', 4, 10]
SERVICES_ADDRESSES_FILE = ['Addresses/Services_full_detailed.csv', 4, 10]
```

Figure 15: transactions_retriever.py/file configuration

execute_query function

This function executes the queries depicted in Figure 17, appending retrieved records to the transaction set.

```
def execute_query(transactions, cursor, query, label):
    logging.info('Executing: ' + label)
    query_time = time.time()
    cursor.execute(query)
    count = 0
    for result in cursor:
        if result[0] not in transactions:
            transactions.add(result[0])
        count += 1
    logging.info('Finished executing query (' + str(count) + ' records) ! Elapsed time: ' + time.strftime('%H:%M:%S', time.gmtime(time.time() - query_time)))
```

Figure 16: transactions_retriever.py/execute_query

```
# Database queries used to retrieve the dataset.
# Using this queries, all transactions related to given address list are retrieved.
TXIN_QUERY = 'SELECT DISTINCT(t1.txid) FROM btc.tx t1 JOIN btc.txin t2 ON (t1.txid = t2.consume_txid) JOIN btc.txout t3 ON (t2.output_txid = t3.output_txid AND t2.vout = t3.vout) WHERE t1.timestamp < \'2018-04-01\' and t3.address IN '
TXOUT_QUERY = 'SELECT DISTINCT(t1.txid) FROM btc.tx t1 JOIN btc.txout t2 ON (t1.txid = t2.output_txid) WHERE t1.timestamp < \'2018-04-01\' and t2.address IN '
```

Figure 17: transactions_retriever.py/queries

generate_csv_file function

This utility function produces a CSV file containing the input list. An output example is depicted in Figure 19.

```
def generate_csv_file(csv_file, header, records):
    logging.info('Generating file: ' + csv_file)
    with open(csv_file,'w') as file:
        file.write(header + '\n')
        for record in records:
            file.write(record + '\n')
    logging.info('File generated!')
```

Figure 18: transactions_retriever.py/generate_csv_file

```
txid
26a0168afabb02e75b6cae0c1f54116974cd4cb545497d719dd9a03c5155dd81
678fd88fe0df6ddd0e70a76cf9f3368ebba8c1a7b2a488c94f43ed6bfd4a6434
c46fd9a2fd2f7a137e8a1aebb2ea48ffa47c4d079330f423e7ef094c876601c6
dbe7c2d512d54d99d79c90dd9c61700e73b0e3a3eb1437ddafba38220d6bb9eb
04c08bc10a44981f36fd5fe9f9cdb0982578bb26c4858d7b0e55fc689f18f6ed
03e2344d38eb2c7c104ed2102d701a3aaca27ff846c51130c97ec88845298e70
d413270a1833d6add19c444d79821a1116185bdb51c5ebcfa90d40b077c24b23
21010f8f4f7cb8bbdde46cc031b81ee81a2dbf5aea8c5d9d0cf7bbd0e698e576
6a4b430cbe169be7edf922b6ce797413455234a41a76b63afcaaa620a5bacac7
aa47fd9fc4205d4c605784cec600fc24c0d3a6714374dcf2d05886d2b27c5697
6cdd208baf18bdf8d9a5ac9b9b5f390f07f95defc7488a67ce2f387f0ec1da0d
058f900de36e56764fd13909896a3dfc03d292eb71a9967d82a22a0392eeb19b
09dfdab53b8e74a22267849b787b005144a8c6cede45d9dbd976fe1b3c734d3d
5d1362506d68e4f2431689e51279303a75de9b147da6254847c4b231676c664d
044c1afd9ba71fc0ebb66803216ba202732e012e292bc284e91aaf25fa968a82
02ae406d3eed0eb75b4503c50d5fb740f30ce80749532ae79ac0380df1e08534
ff3ce06b5d1419f3748a44eb1687529af73fc0caf3e0e196a4013a304228fd3d
816cf659ea1d595d3b4395a6280967e2152d9183ff45214da5d3eb2cbd860496
c8827776187423a20cab2b18d361838827f13c37d503e5d3bdb0ebbac3aaeabc
c9be0250ed54ce0cc8ee1de196fd571d111a562272fa1094071e709ed3027ab4
5186a0965e27cd31beda72438184f8edbe7714d7d0cc55cd6ee275f22a9fa242
6af51fab4bb90f46ac78369c6e953b9ecffa36ccbc434213662412331de6491e
8c25c2716f3144ab54a727004c902bd8fede81fd00f6e8ddefae4c7d22a2a73d
4ba3f2d8a4d6a6fe82394d4db9f8e6369eff379c1446d1ce42378edad431aee0
8986d4b147d63adee54c2277fce80351842439b66468cb69919f3c0d03cae5db
60f7145e62b87e791a27cb2d6a4370d550886d71809b8f70ee26f59ef97a2f45
22e1c119c447c82f25ebb1145969890f5c24adc98caa77d3af88a1c6efe38347
4d539c8aa34ee36b3c1bdb951f2401c4744bc0b07a84ea2b5898a24c2123394e
55c3c68f1d5b19b40391b86b9e9619fd08f6052bbdb7016a1071865df6f04134
acf2da96d2cac005ecd41e71ecd8d0b7ff2a9db44057869169aa2463e8bb621b
c08593ae7e632bb209fc249f5a5736c0070e32bff62f84ff8488ab067825db16
6e6f9442d63ba69e27c5bd3fbe9b560df721e165768e0dd379e442511549abbd
b0f093a7d55d45a11811a9ebf2bec723c065ef53efb4f8d683b18686adce8ca1
0eeffbb25aa9acebabfd33ba08087137f818b3a431c2851955f5d73542f14222
59579700774c8016cdee99b1fdc9ebc45bfd931eadc8789967b2940df939f5d0
a809849a4c73126f8a6f6bb07419dfa7451522c765bbd592448e146763606da5
ebd4b4e342d483eea2c704da91f2471ab4669e592a4213e2591e3f68e73f6098
c301b241d213387a1b9e3d0e169d9a1e8387e796fab6b5d1a008a2142c2b69c6
cfc826546b9083a82e706c483f2f856ca14dc9cce4f978a88c67f541bdb46ed0
489f832b48e7ef8730215e962c8a240ab33a492e6234b634c744e0037aa85fbb
6b57da3a9513bd980ee10d09edd847edba92e2248585107c6d8c3b071ddc17b8
```

Figure 19: transactions_retriever.py/generage_csv_file output example

init_database function

This utility function initializes a connection with the MySQL Database.

```
def init_database():
    logging.info('Initializing Database connection...')
    db = mysql.connect(host='localhost', user='root', password='root', database='btc')
    logging.info('Database connection initialized!')
    return db;
```

Figure 20: transactions_retriever.py/init_database

close_database function

This utility function closes an active connection to the Database. "RESTART" command is used as to reset Database cache for memory optimization.

```
def close_database(db, cursor):
    logging.info('Closing Database connection...')
    if db is not None and db.is_connected():
        cursor.execute('RESTART;')
        db.close()
    logging.info('Database connection closed!')
```

Figure 21: transactions_retriever.py/close_database

3.2.4 Analyzer.py

This Python script performs an unsupervised Machine Learning task, using Deep Graph Infomax [21] and Graph Convolutional Network (GCN) [11] algorithms for node representation learning. After node features have been extracted, classification of each node on the temporal network graph for the Bitcoin transactions dataset is executed, using Logistic regression. Described functionality is further detailed in the following sections.

main script

During the script's execution, an output folder is created, using utility function create_output_folder. Execution records are retrieved by executing function retrieve_execution_records. StellarGraph object is created by executing function generate_graph. Finally, the ML task is performed using function execute_graph_ML.

```
total_time = time.time()
OUTPUT_FOLDER = create_output_folder()
execution_records_dict = retrieve_execution_records()
stellar_graph, node_flags = generate_graph(execution_records_dict)
execute_graph_ML(stellar_graph, node_flags)
logging.info('Total Execution time: ' + time.strftime('%H:%M:%S', time.gmtime(time.time() - total_time)))
```

Figure 22: analyzer.py/main

create_output_folder function

This utility function generates a folder that will contain all files generated in execution, distinguished by timestamp.

```
def create_output_folder():
    logging.info('Creating outputs folder...')
    output_folder = OUTPUT_FOLDER + datetime.now().strftime("%Y_%m_%d_%H_%M_%S") + '/'
    os.mkdir(output_folder)
    logging.info('Outputs folder ' + output_folder + ' created.')
    return output_folder
```

Figure 23: analyzer.py/create_output_folder

retrieve_execution_records function

This function parses the generated dataset of *transactions_retriever.py* script and builds the execution records dictionary, used for labeling graph nodes. Generated dictionary contains an address list for the address type contained in each file. Each file is parsed using utility function read_csv_file.

```
def retrieve_execution_records():
    logging.info('Retrieving execution records...')
    transactions = read_csv_file(TRANSACTIONS_CSV_FILE)
    exchanges_addresses = read_csv_file(EXCHANGES_ADDRESSES_CSV_FILE)
    gambling_addresses = read_csv_file(GAMBLING_ADDRESSES_CSV_FILE)
    historic_addresses = read_csv_file(HISTORIC_ADDRESSES_CSV_FILE)
    malicious_addresses = read_csv_file(MALICIOUS_ADDRESSES_CSV_FILE)
    mining_addresses = read_csv_file(MINING_ADDRESSES_CSV_FILE)
    services_addresses = read_csv_file(SERVICES_ADDRESSES_CSV_FILE)
    execution_records_dict = {'transactions':transactions, 'exchanges_addresses':exchanges_addresses,
                              'gambling_addresses':gambling_addresses, 'historic_addresses':historic_addresses, 'malicious_addresses':malicious_addresses,
                              'mining_addresses':mining_addresses, 'services_addresses':services_addresses}
    logging.info('Execution records retrieved!')
    return execution_records_dict
```

Figure 24: analyzer.py/retrieve_execution_records

read_csv_file function

This utility function parses a CSV file generated by *transactions_retriever.py* script, containing a list of records.

```
def read_csv_file(file):
    logging.info('Retrieving records from csv: ' + file)
    records = set()
    with open(file) as csv_file:
        csv_reader = csv.reader(csv_file)
        header = next(csv_reader)
        for row in csv_reader:
            if row[0] not in records:
                records.add(row[0])
    logging.info('Records found: ' + str(len(records)))
    #logging.info('Records: ' + str(records))
    return records
```

Figure 25: analyzer.py/read_csv_file

generate_graph function

This function generates the StellarGraph object, to execute the unsupervised Machine Learning task. A database connection is initialized using the utility function init_database. A networkx graph is created using the records retrieved by executing functions execute_txin_query and execute_txout_query, as described in Section 3.1.3. After finishing records fetching, Database connection is terminated using utility function close_database. For the generated graph, a graphML file is created for further visualization in external tools. Finally, the networkx graph is converted to a StellarGraph object. A generated graph example is depicted in Figure 27.

```python
def generate_graph(execution_records_dict):
    logging.info('Generating graph...')
    db = init_database()
    cursor = db.cursor()
    graph = nx.DiGraph()
    addresses = set()
    transactions = set()
    execute_txin_query(cursor, execution_records_dict, graph, addresses, transactions)
    execute_txout_query(cursor, execution_records_dict, graph, addresses, transactions)
    close_database(db, cursor)
    logging.info('Generating graph file...')
    nx.write_graphml_xml(graph, OUTPUT_FOLDER + 'graph.graphml')
    logging.info('Graph file generated! Generating StellarGraph object...')
    graph_dict = dict(graph.nodes())
    edge_matrix = nx.to_pandas_edgelist(graph)
    stellar_graph = StellarDiGraph(pd.DataFrame.from_dict(graph_dict, orient='index'), edge_matrix, dtype='float32')
    logging.info(stellar_graph.info())
    node_flags = pd.DataFrame.from_dict(graph.nodes, orient='index')['flag']
    logging.info(Counter(node_flags))
    logging.info('StellarGraph generated!')
    return stellar_graph, node_flags
```

Figure 26: analyzer.py/generate_graph

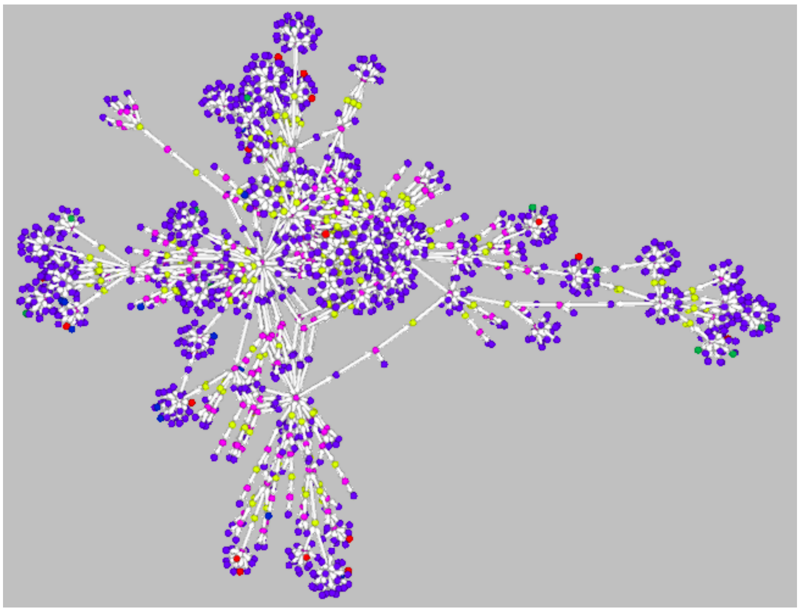

Figure 27: analyzer.py/generate graph example

execute_txin_query and execute_txout_query functions

These functions execute the TXIN_QUERY and TXOUT_QUERY, depicted in Figure 30, and convert retrieved data to networkx graph nodes and links. Utility function retrieve_address_flag is used to determine each address flag.

```python
def execute_txin_query(cursor, execution_records_dict, graph, addresses, transactions):
    logging.info('Fetching TXIN records and converting to graph data...')
    txin_query = TXIN_QUERY + str(execution_records_dict['transactions']).replace('{','(').replace('}',')')
    cursor.execute(txin_query)
    count = 0
    for result in cursor:
        if result[0] not in addresses:
            flag = retrieve_address_flag(execution_records_dict, result[0])
            graph.add_node(result[0], type=Node_Type.ADDRESS.value, flag=flag)
            addresses.add(result[0])
        if result[1] not in transactions:
            graph.add_node(result[1], type=Node_Type.TRANSACTION.value, flag=Node_Flag.TRANSACTION.value)
            transactions.add(result[1])
        graph.add_edge(result[0], result[1], weight=result[3], timestamp=datetime.timestamp(result[2]))
        count += 1
    logging.info('Finished TXIN records retriaval (' + str(count) + ') and conversion!')
```

Figure 28: analyzer.py/execute_txin_query

```python
def execute_txout_query(cursor, execution_records_dict, graph, addresses, transactions):
    logging.info('Fetching TXOUT records and converting to graph data...')
    txout_query = TXOUT_QUERY + str(execution_records_dict['transactions']).replace('{','(').replace('}',')')
    cursor.execute(txout_query)
    count = 0
    for result in cursor:
        if result[0] not in transactions:
            graph.add_node(result[0], type=Node_Type.TRANSACTION.value, flag=Node_Flag.TRANSACTION.value)
            transactions.add(result[0])
        if result[1] not in addresses:
            flag = retrieve_address_flag(execution_records_dict, result[1])
            graph.add_node(result[1], type=Node_Type.ADDRESS.value, flag=flag)
            addresses.add(result[1])
        graph.add_edge(result[0], result[1], weight=result[3], timestamp=datetime.timestamp(result[2]))
        count += 1
    logging.info('Finished TXOUT records retriaval (' + str(count) + ') and conversion!')
```

Figure 29: analyzer.py/execute_txout_query

```python
# Database queries used to retrieve the dataset.
TXIN_QUERY = 'SELECT t3.address, t1.txid, t1.timestamp, t3.value FROM btc.tx t1 JOIN btc.txin t2 ON (t1.txid = t2.consume_txid) JOIN btc.txout t3 ON (t2.output_txid = t3.output_txid AND t2.vout = t3.vout) WHERE t1.txid in '
TXOUT_QUERY = 'SELECT t1.txid, t2.address, t1.timestamp, t2.value FROM btc.tx t1 JOIN btc.txout t2 ON (t1.txid = t2.output_txid) WHERE t1.txid in '
```

Figure 30: analyzer.py/queries

retrieve_address_flag function

This utility function identifies an address flag based on the execution records dictionary. If an address doesn't exist in any of the known address type records, *unknown* flag is used.

```
def retrieve_address_flag(execution_records_dict, address):
    flag = Node_Flag.UNKNOWN.value
    if address in execution_records_dict['exchanges_addresses']:
        flag = Node_Flag.EXCHANGES.value
    elif address in execution_records_dict['gambling_addresses']:
        flag = Node_Flag.GAMBLING.value
    elif address in execution_records_dict['historic_addresses']:
        flag = Node_Flag.HISTORIC.value
    elif address in execution_records_dict['malicious_addresses']:
        flag = Node_Flag.MALICIOUS.value
    elif address in execution_records_dict['mining_addresses']:
        flag = Node_Flag.MINING.value
    elif address in execution_records_dict['services_addresses']:
        flag = Node_Flag.SERVICES.value
    return flag
```

Figure 31: analyzer.py/retrieve_address_flag

init_database function

This utility function initializes a connection with the MySQL Database.

```
def init_database():
    logging.info('Initializing Database connection...')
    db = mysql.connect(host='localhost', user='root', password='root', database='btc')
    logging.info('Database connection initialized!')
    return db;
```

Figure 32: analyzer.py/init_database

close_database function

This utility function closes an active connection to the Database. "RESTART" command is used as to reset Database cache for memory optimization.

```
def close_database(db, cursor):
    logging.info('Closing Database connection...')
    if db is not None and db.is_connected():
        cursor.execute('RESTART;')
        db.close()
    logging.info('Database connection closed!')
```

Figure 33: analyzer.py/close_database

execute_graph_ML function

This function performs the unsupervised Machine Learning task. Node representation model is generated for the given StellarGraph object, using function deep_graph_infomax. Graph nodes dataset is then split into K folds, to evaluate the classifier accuracy. Each fold follows a 70/30 train-test split of the original dataset, generated by StratifiedShuffleSplit function of scikit-learn library. For each fold, function train_and_evaluate is executed, to train the classifier with the fold's train set and evaluate its accuracy using the fold's test set. Each fold's predictions are extracted to a file, along with the general execution statistics and best fold predictions, for further analysis. Execution output example is depicted in Figure 35.

```python
def execute_graph_ML(stellar_graph, node_flags):
    logging.info('Executing graph Machine Learning using StellarGraph Deep Graph Infomax + GCN algorithm...')

    # Generating Deep Graph Infomax model.
    generator, model = deep_graph_infomax(stellar_graph)

    # Generating k-folds.
    logging.info('Generating ' + str(FOLDS) + ' folds...')
    folds = model_selection.StratifiedShuffleSplit(n_splits=FOLDS, test_size=0.3, random_state=42).split(node_flags, node_flags)
    logging.info('Folds generated!')

    # Train and evaluate each fold.
    accuracies = []
    best_fold = 0
    best_accuracy = 0
    best_fold_predictions = None
    best_fold_test_subjects = None
    for i, (train_subjects, test_subjects) in enumerate(folds):
        logging.info('Training and evaluating on fold ' + str(i) + '...')
        acc, pred = train_and_avaluate(generator, model, node_flags[train_subjects], node_flags[test_subjects])
        # Extract fold predictions to a file.
        df = pd.DataFrame({"Predicted": pred, "True": node_flags[test_subjects]})
        df.to_csv(OUTPUT_FOLDER + 'fold_' + str(i) + '_predictions.csv', sep=',')
        accuracies.append(acc)
        # Best fold check.
        if acc > best_accuracy:
            best_fold = i
            best_accuracy = acc
            best_fold_predictions = pred
            best_fold_test_subjects = node_flags[test_subjects]

    # Extract execution statistics to a file.
    statistics = 'K-Fold validation statistics:' + '\nBest fold: ' + str(best_fold) + '\nBest accuracy: ' + str(best_accuracy) + '\nMean accuracy: ' + str(np.mean(accuracies)) + '\nStandard deviation: ' + str(np.std(accuracies))
    logging.info(statistics)
    with open(OUTPUT_FOLDER + 'classification_statistics.txt', "w") as output_file:
        output_file.write(statistics)
    # Extract best fold predictions to a file.
    df = pd.DataFrame({"Predicted": best_fold_predictions, "True": best_fold_test_subjects})
    df.to_csv(OUTPUT_FOLDER + 'best_fold_predictions.csv', sep=',')
```

Figure 34: analyzer.py/execute_graph_ML

```
K-Fold validation statistics:
Best fold: 7
Best accuracy: 0.7616549242992056
Mean accuracy: 0.7506970469195022
Standard deviation: 0.005328798933638884
```

Figure 35: analyzer.py/execute_graph_ML output

deep_graph_infomax function

This function performs the unsupervised training for node representation learning, using Deep Graph Infomax and GCN algorithms, provided by the StellarGraph library. As per usual StallGraph workflow, data generators are created. Since this is an unsupervised task, all nodes are passed to the CorruptedGenerator. A GCN model is created, along with the DeepGraphInfomax model, which will execute the ML task. Generated model is trained to learn the node features and the final embeddings are extracted. When executing this function, a history file is generated, containing the plot of loss over each training epoch, depicted in Figure 37.

```python
def deep_graph_infomax(stellar_graph):
    logging.info('Generating Deep Graph Infomax model for node represation learning...')

    logging.info('Creating data generators...')
    fullbatch_generator = FullBatchNodeGenerator(stellar_graph, sparse=False)
    corrupted_generator = CorruptedGenerator(fullbatch_generator)
    gen = corrupted_generator.flow(stellar_graph.nodes())
    logging.info('Data generators created!')

    logging.info('Creating DeepGraphInfomax + GCN model...')
    gcn_model = GCN(layer_sizes=[128], activations=["relu"], generator=fullbatch_generator)
    infomax = DeepGraphInfomax(gcn_model, corrupted_generator)
    x_in, x_out = infomax.in_out_tensors()
    model = Model(inputs=x_in, outputs=x_out)
    model.compile(loss=tf.nn.sigmoid_cross_entropy_with_logits, optimizer=Adam(lr=1e-3))
    logging.info('DeepGraphInfomax + GCN model created!')

    logging.info('Training generated model to learn node representations...')
    es = EarlyStopping(monitor="loss", min_delta=0, patience=20)
    history = model.fit(gen, epochs=EPOCHS, verbose=0, callbacks=[es])
    logging.info('Generated model trained!')

    logging.info('Generating history plot file...')
    plot_history(history)
    plt.savefig(OUTPUT_FOLDER + 'history.png')
    logging.info('History plot file gemerated!')

    logging.info('Extracting Embeddings...')
    x_emb_in, x_emb_out = gcn_model.in_out_tensors()
    x_out = tf.squeeze(x_emb_out, axis=0)
    emb_model = Model(inputs=x_emb_in, outputs=x_out)
    logging.info('Embeddings extracted!')

    logging.info('Deep Graph Infomax model generated!')
    return fullbatch_generator, emb_model
```

Figure 36: analyzer.py/deep_graph_infomax

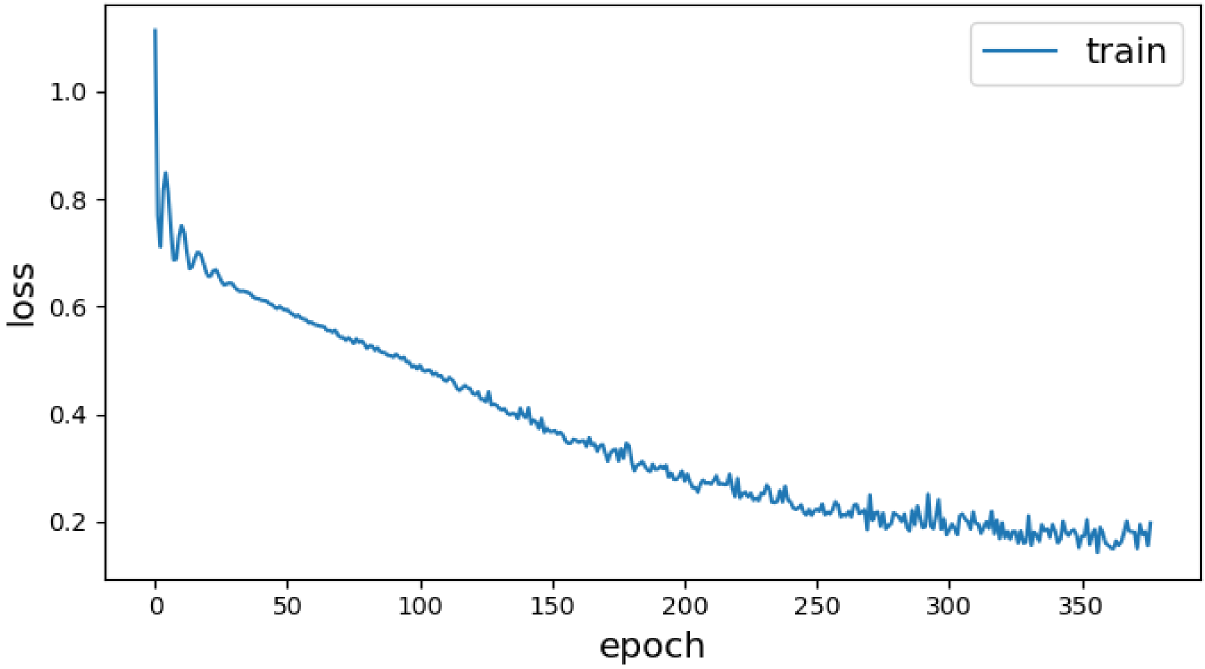

Figure 37: analyzer.py/deep_graph_infomax loss over epochs

train_and_evaluate function

This function performs a classification task using Logistic Regression, for a given features model of graph nodes. Logistic Regression classifier is created and trained on the provided train set and predicts the nodes class of the test set.

```
def train_and_evaluate(generator, model, train_subjects, test_subjects):
    logging.info('Training classifier and performing predictions using Logistic Regression...')
    train_gen = generator.flow(train_subjects.index)
    test_gen = generator.flow(test_subjects.index)
    train_embeddings = model.predict(train_gen)
    test_embeddings = model.predict(test_gen)
    lr = LogisticRegression(multi_class="auto", solver="lbfgs", max_iter=500)
    lr.fit(train_embeddings, train_subjects)
    y_pred = lr.predict(test_embeddings)
    gcn_acc = (y_pred == test_subjects).mean()
    logging.info('Test classification accuracy: ' + str(gcn_acc))
    return gcn_acc, y_pred
```

Figure 38: analyzer.py/train_and_evaluate

4. Implementation evaluation and challenges discussion

This chapter presents a performance evaluation of the proposed solution, detailed in Chapter 3, as well as challenges raised during the design and development process. The performance evaluation follows the solution workflow structure. After the Bitcoin blockchain is parsed and data are imported to the Database, an execution dataset is generated and the Address classification task using unsupervised Machine learning is executed. The solution is evaluated in terms of processing time, storage usage and classification accuracy of the corresponding components. Table 8 presents the hardware and Table 9 the various software and libraries used for the implementation and evaluation of the proposed solution.

Table 8: System hardware

Component	**Description**
CPU	Intel Core i7 6700K, 4C/8T @ 4,5 GHz
RAM	32 GB @ 3200 MHz
GPU	NVIDIA GeForce GTX 1070
Disks	1 x Samsung SSD 850 Evo 250 GB 1 x Samsung NVMe SSD 970 Evo Plus 1 TB 1 x Seagate ST2000DM006 Barracuda HDD 2 TB

Table 9: Software and libraries

Software/Library	**Version**
Windows 10 Pro	20H2, OS build 19042.964
NVIDIA Driver	466.11
MySQL Community Server - GPL	8.0.21
Python	3.8.5
chainside-btcpy	0.6.5
networkx	2.5
stellargraph	1.2.1
tensorflow	2.4.1
scikit-learn	0.24.0

4.1 Blockchain parsing

Using *parser.py* script, the Bitcoin blockchain was parsed until file blk02399.dat, sizing 298 GB in total. All blk*.dat files were parsed after 60 hours. Output file size averaged at 180 MB, with a total size of 426 GB. Figure 39 depicts the parsing time to reach each blk*.dat file, and Figure 40 demonstrates the size increase of the Bitcoin blockchain from 2009 until 2021. Since the parsing time of each blk*.dat file is almost identical, total elapsed time has a steady increase as the blockchain grows.

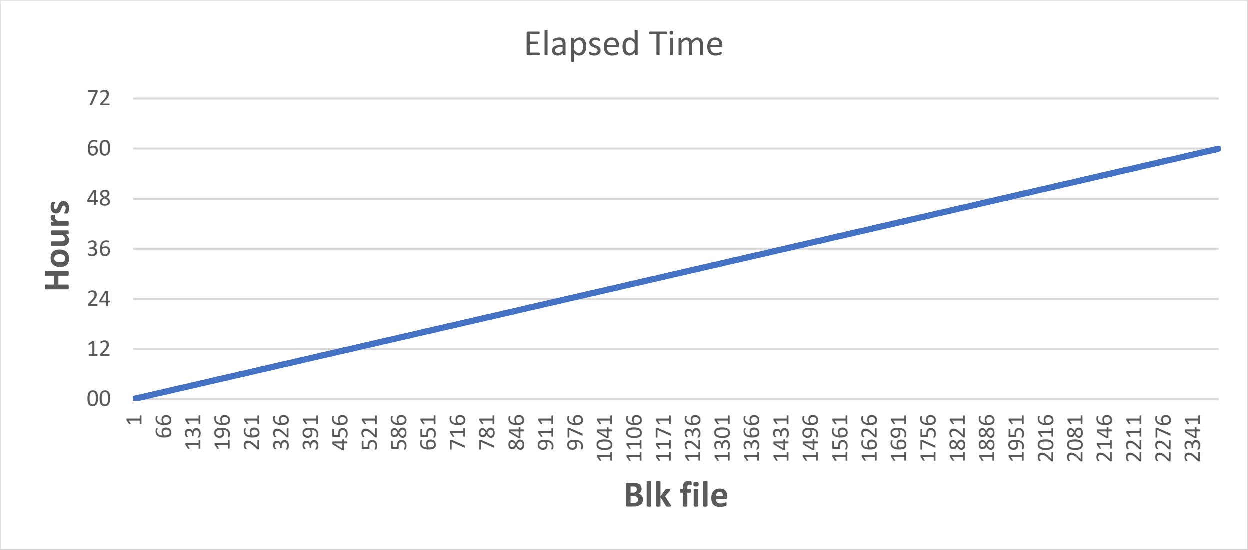

Figure 39: parser.py parsing time

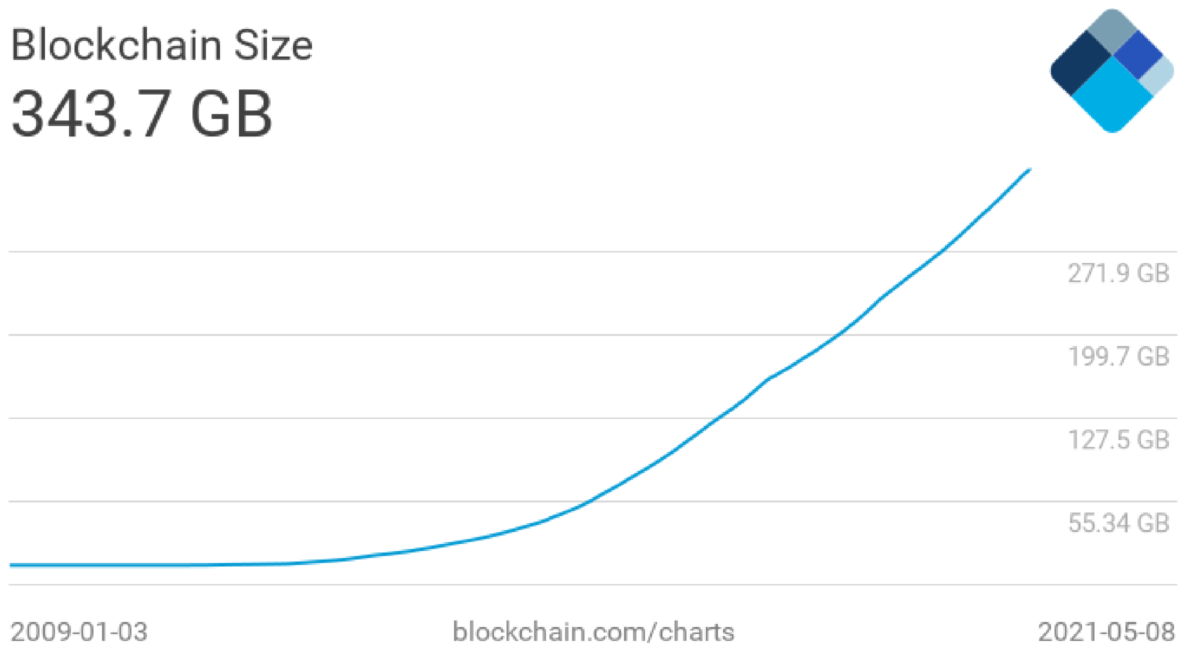

Figure 40: Block chain size [31]

4.2 Data import

To import the parsed Bitcoin blockchain data to the MySQL Database, *reader.py* script was executed. All output files of *parser.py* script were parsed after 9 days 2 hours 32 minutes and 29 seconds, resulting in 652 GB of disk size for the Database, using row compression and field indexing. Figure 41 depicts the parsing time to process each result file sequentially, showing again a steady increase as the blockchain grows.

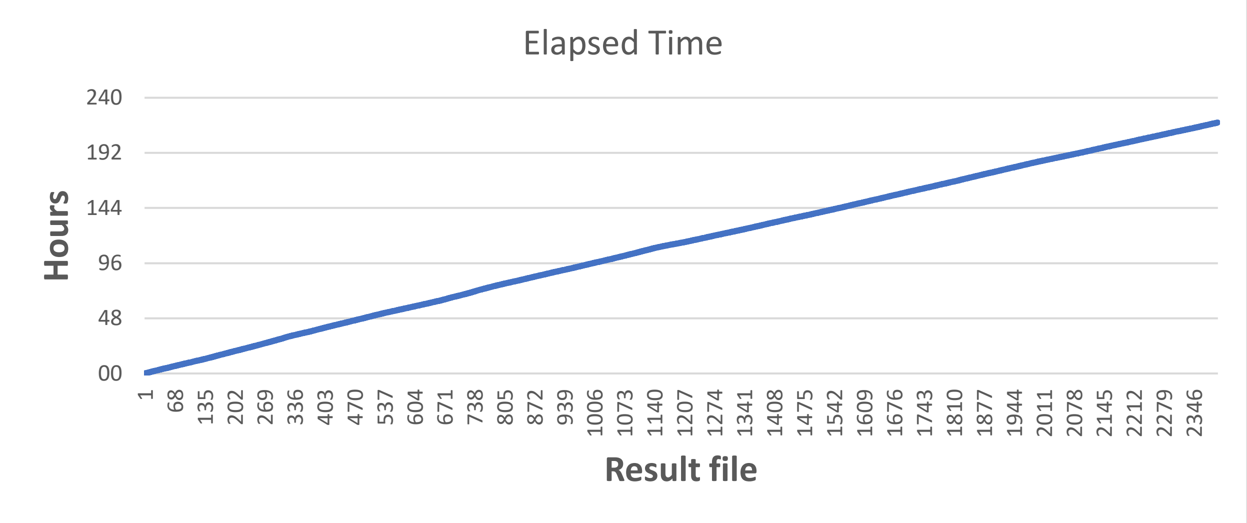

Figure 41: reader.py parsing time

4.3 Machine Learning task

By executing *transactions_retriever.py* script, the generated dataset was utilized by *analyzer.py* script to create the graph on which the Address classification task using unsupervised Machine learning was executed. The graph contained 22236 nodes and 27290 edges in total, distributed among the classes as presented in Table 11.

Table 10: Generated graph class count

Class	Count	Percentage
Transaction	906	4.07%
Unknown	4590	20.64%
Exchanges	806	3.62%
Gambling	1495	6.72%
Historic	5103	22.95%
Malicious	269	1.21%
Mining	8299	37.32%
Services	768	3.45%

For node representation learning, the machine learning model was configured using the default values provided by the StellarGraph demos. Specifically, the GCN model used by DeepGraphInfomax was configured with one hidden layer of 128 units, using the ReLU (Rectified Linear Unit) activation function. DeepGraphInfomax model was configured with TensorFlow's sigmoid_cross_entropy_with_logits as the loss function, along with the Adam learning rate optimizer. Figure 42 shows the loss reduction as epochs increase.

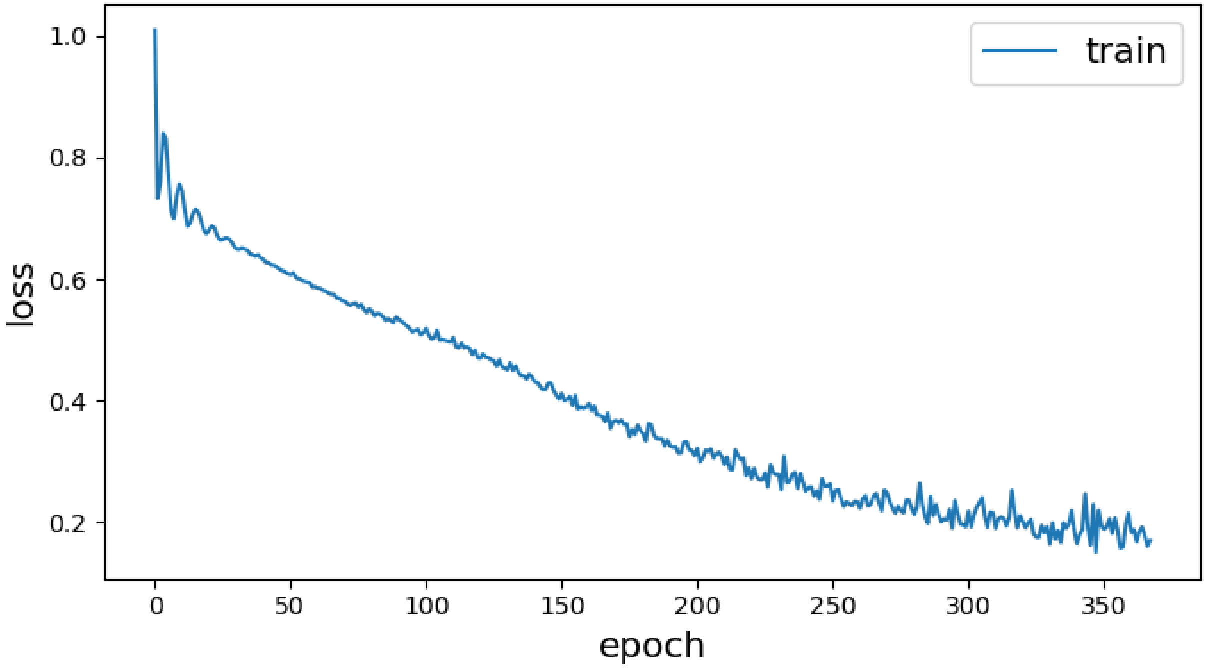

Figure 42: Loss over epochs

After the dataset has been shuffled and split into 10 folds by the StratifiedShuffleSplit function, each fold contained 15565 nodes as the train set and 6671 as the test set. Random state was constant, so that folds would remain the same between each execution. Logistic Regression classifier processed each fold to determine which one had the best accuracy. Average accuracy of folds over epochs is depicted in Figure 43, showing a slight increase as epochs are rising, with 400 epochs as the optimal configuration.

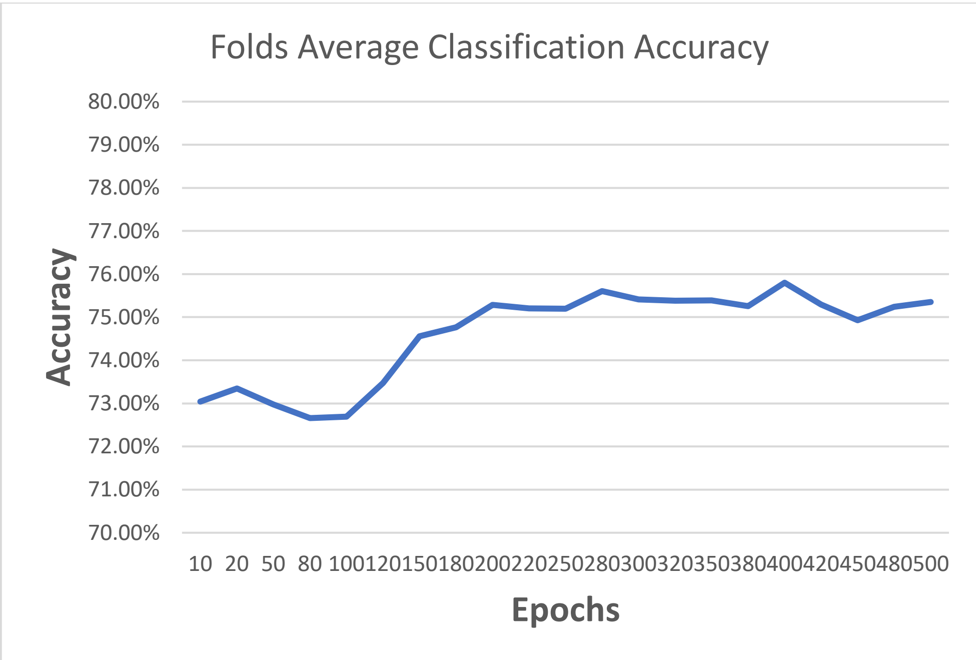

Figure 43: Folds average classification accuracy over epochs

Best overall classification accuracy of 76.39% was achieved using 480 epochs on fold 7. Figure 44 presents the classes composition for both the predicted and actual sets. Figure 45 depicts the predicted class of nodes for each class of the actual set, further detailed in Table 11.

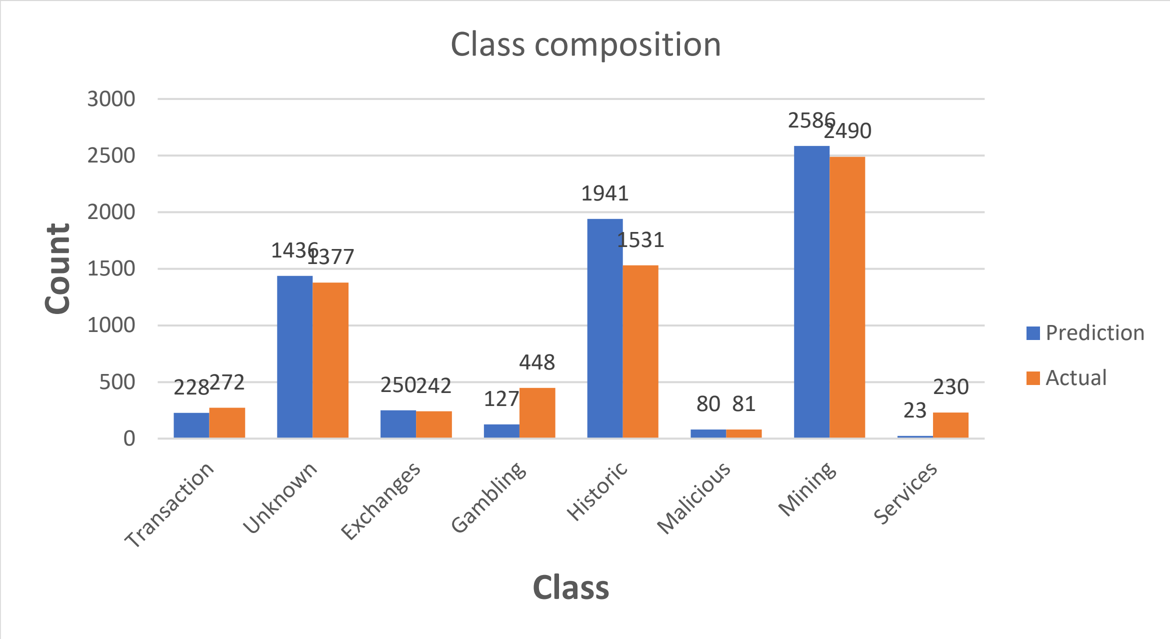

Figure 44: Class composition of predicted and actual sets

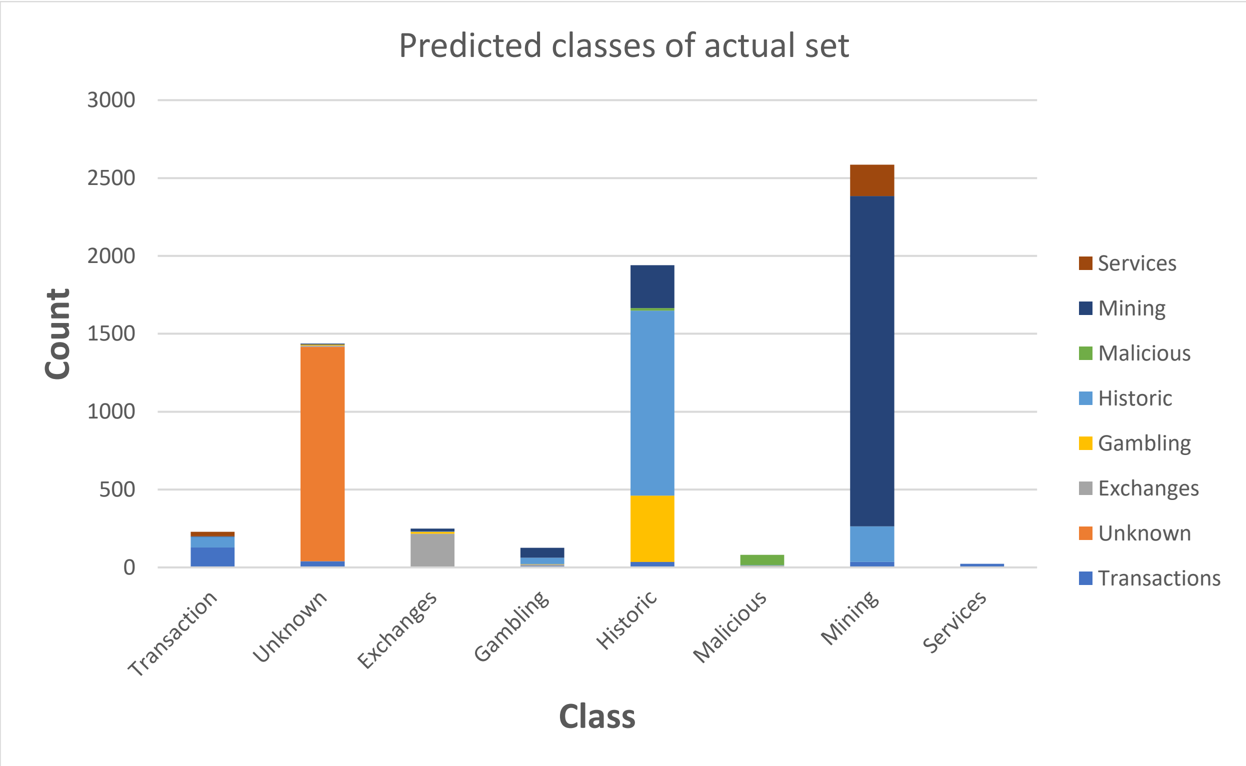

Figure 45: Predicted classes of actual set nodes

Table 11: Predicted classes of actual set nodes details

Class	Transaction	Unknown	Exchanges	Gambling	Historic	Malicious	Mining	Services	Accuracy
Transaction	129	40	0	8	35	2	35	23	47.43%
Unknown	0	1376	0	0	1	0	0	0	99.93%
Exchanges	0	9	216	11	0	6	0	0	89.26%
Gambling	0	5	13	3	424	3	0	0	0.67%
Historic	66	0	2	41	1188	5	229	0	77.60%
Malicious	0	0	0	0	17	64	0	0	79.01%
Mining	5	6	19	64	276	0	2120	0	85.14%
Services	28	0	0	0	0	0	202	0	0.00%

Through further inspection of Table 11, it's obvious that the Logistic Regression classifier deals with problems in identification of the nodes of Gambling and Services classes, mislabeling them mainly as Historic and Mining classes respectively. Furthermore, Transaction nodes may be mislabeled as Address nodes. On the other hand, identifying Malicious addresses stands at a very respectable 79.01%. Further optimizing of the classifier and improving of the class system, using detail subclasses provided by the Dataset, should be explored, to identify potential classification accuracy improvements.

4.4 Challenges

Various challenges were encountered during the implementation of the proposed solution, due to the massive data size. The Bitcoin blockchain was weighted 298 GB at the time of development, resulting in an inefficient and time-consuming parsing and importing of data to the Database. To overcome this issue, the process was split into two distinct modules, to enable concurrent execution of each module on different files, as each process was executed faster when not bounded by the other one. Additionally, data import could be further optimized with the usage of a batching commit code.

After importing all the data into the MySQL Database, disk size was 1129 GB. Due to this size, a slow 2 TB HDD drive was used to host the Database, resulting in slow query execution times. Row compression was enabled, reducing the total size to 652 GB, allowing the usage of a faster NVME SSD 1 TB drive. To further increase Database performance, field indexing was enabled, along with the modification of MySQL configuration parameter *innodb_buffer_pool_size*, which was set to 16 GB, to increase the Database RAM cache size, for faster query executions.

The initial approach of executing the unsupervised Machine Learning task, included the generation of the execution dataset and ML processing at the same script, which was slowing development process, as the dataset was rebuilt in each execution. To overcome this issue, execution dataset build was removed from the script and *transactions_retriever.py* script was created. This resulted in faster development times and repetition of executions with different configurations, on the same dataset, without the need to rebuild the dataset.

As *analyzer.py* script was developed, each consecutive execution of the Logistic Regression classifier resulted in memory usage reaching system limit. This issue occurred due to a known memory leak bug in TensorFlow library, which was bypassed by disabling eager execution. Another memory optimization placed in the code, was the usage of the SQL "RESTART" query, after records retrieval was completed, to reset the Database memory cache.

5. Conclusions and Future Work

In this dissertation, a complete solution on how to store Bitcoin blockchain data and analyze them using Machin Learning algorithms, was presented. Bitcoin data were imported to a MySQL Database using Python scripts, allowing access using complicated queries from other resources, to fully utilize the parsed information. This enabled the creation of a Python script which creates a graph and performs a Node Classification task using unsupervised Machine Learning.

The proposed solution was evaluated in terms of processing time, storage usage and classification accuracy of the corresponding components. Parsing Bitcoin blockchain data to a more usable format took 60 hours, with an output folder of 426 GB in total size. Importing the parsed data to the MySQL Database completed after 9 days 2 hours 32 minutes and 29 seconds, resulting in 652 GB of disk size for the Database, after applying optimizations. Executing the unsupervised Machine Learning task, the best accuracy achieved was 76.39%, showing that it is possible to classify Bitcoin addresses without knowing any features.

This work paves the way for further research discussion. Using similar parsing techniques, a Database containing the complete information of the Bitcoin blockchain can be created. This will allow researchers to analyze its data with different approaches and extract information related to their search field. The promising results of the classification task indicate that the proposed classification method can be further improved, by optimizing and training the classifier using different architectural approaches, or by using a more detailed class system. Ideally, such approaches can facilitate blockchain forensics operations and enhance the capabilities of tracing malicious actors in bitcoin and other cryptocurrencies.

Bibliography

[1] Nakamoto, S. (2008). *Bitcoin: A peer-to-peer electronic cash system.*

[2] Antonopoulos, A. (2014). *Mastering Bitcoin: Unlocking Digital Crypto-Currencies.* O'Reilly Media. ISBN 978-1-4493-7404-4.

[3] "*Statement of Jennifer Shasky Calvery, Director Financial Crimes Enforcement Network United States Department of the Treasury Before the United States Senate Committee on Banking, Housing, and Urban Affairs Subcommittee on National Security and International Trade and Finance Subcommittee on Economic Policy*" (2013) fincen.gov. Financial Crimes Enforcement Network. Archived from the original on 9 October 2016. Retrieved 1 June 2014.

[4] BitcoinCore. About, url: https://bitcoincore.org/en/about/

[5] Bitcoin. Protocol Documentation, url: https://en.bitcoin.it/wiki/Protocol_documentation

[6] "*What is MySQL?*". MySQL 8.0 Reference Manual. Oracle Corporation. Retrieved 3 April 2020

[7] StellarGraph, url: https://stellargraph.readthedocs.io/en/stable/README.html

[8] Hamilton, W.L., Ying, R., and Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. *Neural Information Processing Systems (NIPS).*

[9] Zhang, D., Jie, Y., Zhu, X. and Zhang, C. (2019). Attributed Network Embedding via Subspace Discovery*. Data Mining and Knowledge Discovery*.

[10] Veličković, P. et al. (2018). Graph Attention Networks. *International Conference on Learning Representations (ICLR).*

[11] Kipf, T. N., Max Welling, (2017). Graph Convolutional Networks (GCN): Semi-Supervised Classification with Graph Convolutional Networks. *International Conference on Learning Representations (ICLR)*.

[12] Chiang, W., Liu, X., Si, S., Li, Y., Bengio, S. & Hsiej, C., (2019). Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. *KDD* arXiv:1905.07953.

[13] Wu, F., Zhang, T., A. H. de Souza, Fifty, C., Yu, T. & Weinberger, K. Q. (2019). Simplifying Graph Convolutional Networks. *International Conference on Machine Learning (ICML).*

[14] Klicpera, J., Bojchevski, A., Günnemann, A. & S., (2019). Predict then propagate: Graph neural networks meet personalized PageRank. *ICLR*. arXiv:1810.05997.

[15] Grover, A. & Leskovec, J., (2016). Node2Vec: Scalable Feature Learning for Networks. ACM SIGKDD *International Conference on Knowledge Discovery and Data Mining (KDD).*

[16] Dong, Y., Nitesh V. Chawla, & Swami, A. (2017). Metapath2Vec: Scalable Representation Learning for Heterogeneous Networks. *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD),* 135–144.

[17] Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., & Welling, M. (2018). Modeling relational data with graph convolutional networks. *European Semantic Web Conference.* arXiv:1609.02907

[18] Trouillon, T., Welbl, J., Riedel, S., Gaussier, É. & Bouchard G., (2016). Complex Embeddings for Simple Link Prediction. *ICML.*

[19] Donnat, C., Zitnik, M., Hallac, D., & Leskovec, J. (2018). Learning Structural Node Embeddings via Diffusion Wavelets. *SIGKDD*, arXiv:1710.10321.

[20] Abu-El-Haija, S., Perozzi, B., Al-Rfou, R. & Alemi, A. (2018). Watch Your Step: Learning Node Embeddings via Graph Attention, *NIPS*. arXiv:1710.09599.

[21] Veličković, P., Fedus, W., Hamilton, W. L., Lio, P., Bengio, Y., Hjelm, R. D., (2019). Deep Graph Infomax. *ICLR,* arXiv:1809.10341.

[22] Nguyen, G. H., Lee, J. B., Rossi, R. A., Nesreen K. A., Koh, E., & Kim, S. (2018). Continuous-Time Dynamic Network Embeddings. Proceedings of the 3rd International Workshop on Learning Representations for Big Networks *(WWW BigNet).*

[23] Yang, B., Yih,W., He, X., Gao, J., & Deng, L. (2015). Embedding Entities and Relations for Learning and Inference in Knowledge Bases. *ICLR,* arXiv:1412.6575

[24] Zhang, M., Cui, Z., Neumann, M. & Chen, Y. (2018). An End-to-End Deep Learning Architecture for Graph Classification. *AAAI.*

[25] Zhao, L., Song, Y., Zhang, C., Liu, Y., Wang, P., Lin, T., Deng, M. & Li, H. (2019). T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction. *IEEE Transactions on Intelligent Transportation Systems*.

[26] Leonov, D. (2020). Blockchain parser, url: https://github.com/ragestack/blockchain-parser

[27] btcpy, url: https://github.com/chainside/btcpy

[28] Entity-address dataset for 2010-2018 Bitcoin transactions, url: https://github.com/Maru92/EntityAddressBitcoin

[29] Jourdan, M., Blandin, S., Wynter, L., Deshpande, P. (2018). Characterizing Entities in the Bitcoin Blockchain. *Data Mining Workshop (ICDMW), IEEE International Conference.* arXiv:1810.11956

[30] Jourdan, M., Blandin, S., Wynter, L., Deshpande, P. (2019). A Probabilistic Model of the Bitcoin Blockchain. *Computer Vision and Pattern Recognition Workshop (CVPRW).* arXiv:1812.05451

[31] Blockchain Size, url: https://www.blockchain.com/charts/blocks-size

[32] Thibault de Balthasar, Julio Hernandez-Castro:An Analysis of Bitcoin Laundry Services. NordSec 2017: 297-312

[33] Heterogenous GraphSAGE (HinSAGE), url: https://stellargraph.readthedocs.io/en/stable/hinsage.html

[34] Casino, Fran, Thomas K. Dasaklis, and Constantinos Patsakis. "A systematic literature review of blockchain-based applications: Current status, classification and open issues." Telematics and informatics 36 (2019): 55-81.

[35] Juan A. Garay, Aggelos Kiayias, Nikos Leonardos: The Bitcoin Backbone Protocol: Analysis and Applications. EUROCRYPT (2) 2015: 281-310

Abbreviations

ML	Machine Learning
SHA	Secure Hash Algorithm
P2PKH	Pay to Public Key Hash
P2SH	Pay to Script Hash
UTXO	Unspent Transaction Outputs
PoW	Proof-of-Work
RDBMS	Relational Database Management System
SQL	Structured Query Language
DB	Database
API	Application Programming Interface
SAGE	SAmple and aggreGatE
GAT	Graph Attention Network
GCN	Graph Convolutional Network
SGC	Simplified Graph Convolutional Network
PPNP/APPNP	(Approximate) Personalized Propagation of Neural Predictions
RGCN	Relational Graph Convolutional Network
CTDNE	Continuous-Time Dynamic Network Embeddings
DGCNN	The Deep Graph Convolutional Neural Network
TGCN	Temporal Graph Convolutional Network
CPU	Central Processing Unit
RAM	Random Access Memory
GPU	Graphics Processing Unit
*C/*T	number of Cores/ number of Threads
MHz/GHz	Mega/Giga Hertz
MB/GB/TB	Mega/Giga/Tera Bytes
SSD	Solid State Drive
NVMe	Non-Volatile Memory express
HDD	Hard Disk Drive
OS	Operating System

Glossary

Bitcoin	Cryptocurrency invented in 2008 by Satoshi Nakamoto.
Blockchain	Blockchain is a system of recording information in a way that makes it difficult or impossible to change, hack, or cheat the system.
Hash	Hash function coverts data of arbitrary length to a fixed length.
Mainnet	Bitcoin main blockchain.
Satoshi	The smallest unit of the Bitcoin cryptocurrency
MySQL	Open-source relational database management system.
Database schema	A database schema represents the logical configuration of all or part of a relational database.
Python	Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.
StellarGraph	Python library for machine learning on graph-structured data.
Machine Learning	Machine learning is an application of artificial intelligence (AI) that provides systems with the ability to automatically learn and improve from experience without being explicitly programmed.
Classification task	A task that requires the use of machine learning algorithms that learn how to assign a class label to examples from the problem domain.
Graph	A common data structure that consists of a finite set of nodes (or vertices) and a set of edges connecting them
Script	A collection of commands in a file designed to be executed like a program.
Function	A function is a block of code which only runs when it is called.

List of Figures

List of Tables