FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Generating Malware Family Signatures from Behavioral Graphs using Unsupervised Learning
Student:	Bc. Tomáš Zvara
Supervisor:	Mgr. Martin Jureček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2022/2023

Instructions

The aim of this thesis is to design and implement algorithms that will recognize malicious behavioral patterns in datasets of behavioral graphs and evaluate them as a means for malware detection. The datasets are generated by Avast internal systems, and they contain behavioral graphs from multiple well-known active malware families.

The following steps will be part of our work:
1. Study the selected supervised (e.g., Graph neural networks) and unsupervised (e.g., K-means, DBSCAN) learning methods.
2. Analyze multiple well-known malware families with respect to their malicious behavioral characteristics.
3. Perform experiments with different clustering methods. Evaluate the quality of clusters and generate signatures from them.
4. Evaluate the quality of the output signatures of malware families and discuss the results.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 1 February 2022 in Prague.

Master's thesis

GENERATING MALWARE FAMILY SIGNATURES FROM BEHAVIORAL GRAPHS USING UNSUPERVISED LEARNING

Bc. Tomáš Zvara

Faculty of Information Technology
Department of Information Security
Supervisor: Mgr. Martin Jureček, Ph.D.
Consultants: Mgr. Martin Bálek, Ing. Václav Belák, Ph.D.
May 4, 2022

Contents

List of Figures

List of Tables

I wish to express my sincere thanks to my supervisor, Mgr. Martin Jureček, Ph.D., and to my external consultants, Mgr. Martin Bálek, and Ing. Václav Belák, Ph.D., for their valuable guidance, insightful comments and constructive advice. I would also like to thank my family for their continuous support throughout my studies.

Declaration

In Prague on May 4, 2022 ....

Abstrakt

Behaviorálny štít je komponenta antivíroveho softvéru firmy Avast zodpovedná za monitorovanie systému a identifikovanie podozrivého správania bežiacich procesov. Správanie procesov je zachytené vo forme behaviorálnych grafov. Prebiehajúci interný výskum skúma možnosti aplikácie neurónových modelov, takzvaných grafových neurónových sietí, za účelom umožnenia strojového učenia nad týmito grafmi. Cieľom práce je skúmať tri rozličné komprimované reprezentácie grafov, ktoré boli vyprodukované existujúcimi modelmi neurónových sietí, a overiť, či tieto reprezentácie umožnujú rozlišovať škodlivé správanie jednotlivých malvérových rodín. Analýza štruktúry týchto reprezentácií bola vykonaná použitim známych klastrovacích algoritmov, a to k-means, DBSCAN a aglomeratívne klastrovanie. Výsledky klastrovacieho procesu boli vyhodnotené pomocou interných a externých merítok. Cieľom je overenie hypotézy, že vytvorené klastre by mali reprezentovať správanie jednotlivých malvérových rodín a umožniť jeho zachytenie vo forme detekcie. Avšak, experimenty ukazujú, že aplikovanie spomenutých klastrovacích metód nevedie k uspokojivým výsledkom a metódy produkujú nekvalitné klastre, ktoré neoddeľujú grafy jednotlivých rodín. To je primárne spôsobené dvoma faktormi. Prvý je, že behaviorálne grafy nezachycujú správanie rodín dostatočne na to, aby mohli byť použité na ich rozlíšenie. Druhý faktor je nízka kvalita poskytnutých označení malvérových rodín.

Kľúčové slová škodlivý softvér, rodina škodlivého softvéru, behaviorálna detekcia škodlivého softvéru, grafová neuronová sieť, zhlukovanie, behaviorálna analýza, behaviorálny graf

Abstract

The behavioral shield is a component of Avast AV responsible for monitoring the system and identifying suspicious behavior of running processes. The behavior is captured in the form of behavioral graphs. There is ongoing internal research that studies the options to use novel deep learning models, i.e., graph neural networks, to allow high-scale learning on these graphs. This thesis aims to study three different graph embeddings, which were produced by the existing graph neural network models, and verify whether the embedded representations allow distinguishing the malicious behavior of various malware strains. The structure of embedded spaces is analyzed using well-known clustering methods, namely k-means, DBSCAN, and agglomerative clustering. The results of the clustering process are evaluated by intrinsic and extrinsic measures. The hypothesis is that the formed clusters should represent individual malware families and thus can be used to create a behavioral signature to detect them. However, performed experiments show that the applied clustering methods produce low-quality clusters that do not allow separating the selected malware strains. There are two factors that cause the low performance. The first one is the poor expressibility of the behavioral graphs with respect to the individual malware strains. The second one is the low quality of the provided labels.

Keywords malware, malware strain, malware behavioral detection, graph neural network, clustering, behavioral analysis, behavioral graph

Acronyms

AV	Antivirus (software)
BCE	Binary cross-entropy
CNN	Convolutional neural network
CS	Cosine similarity
DBI	Davies-Bouldin index
DGA	Domain generating algorithm
ED	Euclidean distance
GNN	Graph neural network
GCN	Graph convolutional network
IOC	Indicator of compromise
KL	Kullback–Leibler (divergence)
LOL	Living of the land techniques
MD	Manhattan distance
MLP	Multilayer perceptron
OS	Operating system
PE	Portable executable
PI	Purity index
PUP	Potentially unwanted application
RI	Rand index
RNN	Recurrent neural network
SI	Silhouette index
TTP	Tactics, techniques and procedures
UMAP	Uniform manifold approximation and projection
VAE	Variational autoencoders

Introduction

Modern AV solutions have multiple layers of protection that defend the user endpoints from malicious programs. In Avast AV, these layers are represented by individual protection units, also called shields, where each shield is capable of detecting the threat on a different level. There is file system protection, network communication protection, or behavioral protection, while each detection mechanism has certain advantages. The advantage of behavioral malware protection is that it allows detecting previously unseen malware that naturally cannot be detected by signature-based detections. A good example is a modern *ransomware* malware where the distributed samples usually evolve and morph rapidly, but it requires untrivial effort from the adversary side to change the ransomware behavior.

The behavior is captured in the form of graphs, where the individual running processes and system objects, which processes interact with, are represented as graph nodes, and the performed actions are represented as graph edges. The novel deep learning approaches allow training the deep neural network models that can process such graph structures directly on the input and generate the embedded representation of the nodes, edges or whole graphs. These models are called graph neural networks (GNN). There are emerging practical applications in areas such as antibacterial discovery, physics simulations, fake news detection, traffic prediction and recommendation systems, where these neural networks outperform the traditional approaches used for machine learning on graphs [1].

Due to the promising results of GNNs, there is an ongoing company internal research that attempts to create graph neural network models capable of learning to distinguish malicious behavior from harmless non-malicious behavior. However, the behavioral graphs are very complex data structures where nodes alone contain hundreds of features and graph structure brings additional information complexity into the given task. Thus, it is challenging to interpret whether the neural network learns some sensible features useful for separating malware and clean graphs. One of the ideas, which could help understand the extracted features, is to examine the low-dimensional graph embeddings produced by these models and check whether the compressed graph representations allow differentiation of distinct types or strains of malware. This hypothesis can be verified using clustering methods in the space of embedded representations. If any of the clustering algorithms produce results that separate the graphs into clusters according to the malware strain or type well, the assumption is that the neural network selected "valuable" features. Moreover, the behavioral graphs in such clusters can then be used to produce a behavioral signature for the specified malware family.

The main goal of this thesis is to perform an exploratory data analysis of the provided graph embeddings that are produced by three different GNN models. The embeddings are analyzed using well-known clustering methods, i.e., k-means, DBSCAN, and agglomerative clustering. The clustering results are evaluated by selected intrinsic and extrinsic measures. In case of good cluster separation regarding distinct malware types or strains, the formed clusters should be used to produce behavioral signatures for the malicious behavior they represent.

Chapter 1 sums up the formal graph theory background and the possible approaches to a graph representation. Chapter 2 continues by introducing the deep learning techniques focusing mainly on graph neural networks. Chapter 3 is devoted to cluster analysis. It contains a description of selected cluster algorithms, and it discusses how can the cluster results be evaluated. Chapter 4 introduces the behavioral graphs and describes what kind of information they store.

The practical part begins with Chapter 5, where the one day of data provided by the behavioral shield is analysed and filtered. There is also a discussion regarding the quality of the behavioral graphs with strain labels assigned. Chapter 6 describes the clustering experiments and evaluates the results of clustering, followed by the discussion of clusters quality.

Chapter 1

Graph Theory

This chapter starts with basic formal definitions of graph theory, followed by an introduction to the possibilities for storing information within graph structure. Then, two methods for comparing graphs are presented, one traditional (graph isomorphism) and the second one adapted from the set domain (Jaccard index). The first chapter ends with an overview of different possibilities on how to represent a graph in a digital form, both in compressed and uncompressed form. This chapter lays a foundation for the following chapter describing graph neural networks and provides tools for working with graphs in the practical part of this thesis.

1.1 Formal definitions

Graph can be formally defined as a pair $G = (V, E)$ where:

- $V = \{v_1, v_2, ...\}$ is a nonempty set of nodes (or vertices);
- and $E = \{e_1, e_2, ...\}$ is a set of edges.

Affiliation to a particular graph G can be denoted with subscript, i.e. V_G, E_G sets belong to graph G. Further, $|V_G|$ (resp. $|E_G|$) stands for the number of nodes (resp. number of edges). The edges of a graph can be either *directed* or *undirected*. *Directed* graph edge $e = (v_i, v_j)$ is an ordered pair of two nodes, whereas *undirected* edge $e = \{v_i, v_j\}$ is a two-element subset from V. Some graphs can also have *weighted* edges which means that each edge has assigned a weight $w_i \in \mathbb{R}$.

Node v_j is *adjacent* to the node v_i when there exists such edge $e = \{v_i, v_j\}$ that $e \in E$ and the nodes are connected with an arc. Then, both v_i and v_j are *incident* with the edge e and vice versa. The number of incident edges with node v is called degree $d(v)$. An *isolated* node is one that is not incident with any edge. The degree is in case of isolated node v equal to $d(v) = 0$. The graph definition generally permits *loops* - an edge to be associated with a node pair $\{v_i, v_i\}$.

All the adjacent nodes of v define a node neighborhood. If node v is included in the set, the neighborhood is denoted as *closed*. In case v is not included, the neighborhood is called *open*. Formally, an open neighborhood of node v of graph G would be defined as:

$$\mathcal{N}_v = \{u \in V_G \setminus \{v\} | (u, v) \in E_G\} \tag{1.1}$$

A graph with a finite set of nodes as well as a finite set of edges is denoted as a *finite* graph; otherwise, it is an *infinite* graph. A very significant graph structure feature is cyclicity. A graph is considered to be cyclic whenever there exists a non-empty path from a node back to itself. A *path* is a sequence of edges that joins a sequence of nodes that are all distinct. When there are no cycles in the graph, the graph is called *acyclic* [2].

1.1.1 Storing information in graph

The versatility of the graph structure lies in the expressiveness of its individual elements. In many practical applications, it is useful to enrich the graph elements with additional attributes. A most basic extension of a node is to imbue it with a type, meaning that the set V will be partitioned into disjoint sets $V = V_1 \cup V_2 \cup \dots \cup V_k$ where $V_i \cap V_j = \emptyset, \forall i \neq j$. The same can be done for the edges to allow modelling different types of interactions between nodes – a multi-edge graph. The edge notation is extended to include an edge type or a relation type τ, such as $(u, \tau, v) \in E$. Graphs where both nodes and edges are extended with types are called **heterogeneous graphs** [3].

A more sophisticated extension associates each node $v \in V$ with a feature vector $\boldsymbol{x}_v = (x_1, \dots, x_d)$ where $d \in \mathbb{N}$ is the vector dimension. All the nodes in graph G can be then represented via feature matrix $\boldsymbol{X} \in \mathbb{R}^{|V| \times d}$. The edge-level feature matrix is defined likewise as $\boldsymbol{I} \in \mathbb{R}^{|E| \times d'}$ where each row is a particular edge feature vector $i_{uv} = (i_1, \dots, i_{d'})$. The graph definition is extended into $G = (V, E, \boldsymbol{X}, \boldsymbol{I})$ [2].

1.1.2 Comparing graphs

To express similarity between two graphs, two additional definitions need to be introduced:

- Graph H is said to be a **subgraph** of graph G if $V_H \subseteq V_G$ and $E_H \subseteq E_G$.
- Two graphs G and G' are **isomorphic** when there exists a one-to-one correspondence between their nodes and between their edges such that it preserves the incidence relations. This correspondence can be mathematically expressed as a bijective function $f : V_G \to V_{G'}$ such that $(u, v) \in E_G$ if and only if $(f(u), f(v)) \in E_{G'}$ for every pair of nodes $u, v \in V_G$. The example of two isomorphic graphs is presented in Figure 1.1.

Graph isomorphism can also be seen as an equivalence relation on graphs. In the isomorphism definition, graphs are understood to be directed simple graphs without any labels as the definition focuses purely on the structure. Although, isomorphism may be applied to all other variants of graph from the previous section by adding the requirements to preserve the corresponding additional elements of a structure. However, graph isomorphism is relatively computationally expensive, and it gives only true-false statement, i.e., it does not express how (dis)similar two graphs are [4].

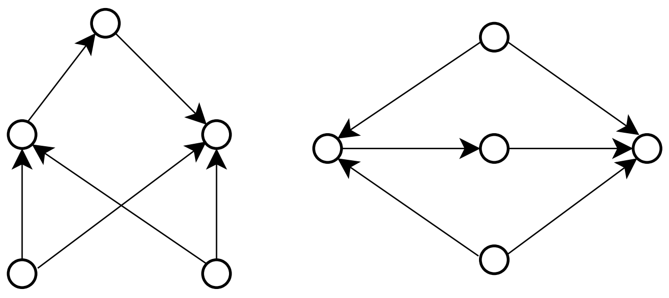


Figure 1.1 Example of two isomorphic directed unlabeled graphs

Despite the fact that there are more sophisticated methods for calculating graph similarity [5], for the purpose of this thesis, the comparison of the two graphs will be reduced to a comparison of the node sets of two graphs. *Jaccard index* is a classic measure of similarity between two sets, and it is defined by:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{1.2}$$

By design, *Jaccard index* values are from the interval $0 \leq J(A,B) \leq 1$. If both sets are empty, *Jaccard index* is set to $J(A,B) = 1$. A complementary measure to the *Jaccard index* is *Jaccard distance* that express dissimilarity of two sets:

$$d_J(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{1.3}$$

Even though this metric completely ignores the graph structure (which is more acceptable for behavioral graphs, see Section 4.2.1), it gives some notion about graphs similarity by comparing their node elements. Also, calculating the *Jaccard index* is much faster than solving the isomorphism problem and, therefore, more suitable for application on big clusters of graphs where one needs to compare all graphs with each other. An example of the *Jaccard Index* application for comparison of two graphs can be seen in Figure 1.2.

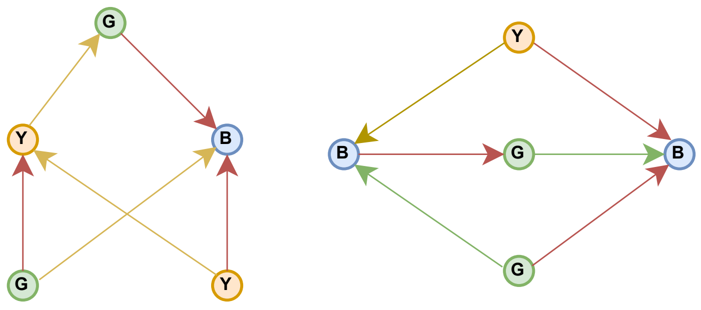


Figure 1.2 There are two heterogeneous graphs on the picture; both nodes and edges have a 1-dimensional attribute – *type*. Graphs are no longer isomorphic if one takes into account also type information. The *Jaccard Index* is $J(\{G,Y,B,G,Y\},\{Y,B,G,B,G\}) = 4/6 = 0.\overline{6}$

1.2 Graph representations

There are a couple of approaches how to turn a graph into a format that is suitable for computational tasks and can be stored in the computer memory. The challenge is in expressing effectively both the feature information and relationships. Matrices come as a very natural representation of graph structure. To represent a graph with a matrix, one needs to artificially order the nodes in the graph so that every node indexes a particular row and column.

Degree matrix $\boldsymbol{D} \in \mathbb{N}^{|V|\times|V|}$ is a diagonal matrix, i.e., non-diagonal entries are all zeros, where the nodes index rows and columns, and each diagonal entry gives the degree of the corresponding node. This matrix alone does not allow correct graph reconstruction since it does not contain information about the incidence. The formal definition is:

$$\boldsymbol{D}ij = \begin{cases} d(v_i) & \text{if i = j} \\ 0 & \text{otherwise} \end{cases}$$

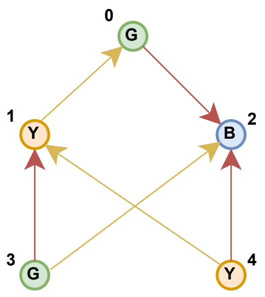
Adjacency matrix $\boldsymbol{A} \in \mathbb{N}^{|V|\times|V|}$ is a widespread matrix representation that captures the adjacency relation. The formal definition is:

$$\boldsymbol{A}ij = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

It is possible to use edge labels or types instead of 1 in the adjacency matrix in the case of weighted or heterogeneous graphs. The other option for multi-edge graphs is to have multiple adjacency matrices for each edge type. By the definition, the adjacency matrix of an undirected graph is always symmetric. This does not hold for the directed graph since the edge is an ordered pair, i.e., $(v_i, v_j) \neq (v_j, v_i), \forall i \neq j$.

If the graphs do not contain loops (edges like (v_i, v_i)), then the adjacency matrix diagonal contains just zero elements. Therefore, the diagonal can be used to store additional information. Usually, it is a degree of a particular node $d(v)$. The combination can be achieved by subtracting the degree matrix from the adjacency matrix. This particular matrix, $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$, is called the *Laplacian matrix* [6]. Besides having useful mathematical properties [3], the Laplacian matrix represents the graph structure without any loss of information.

Both the Laplacian matrix $\boldsymbol{L}$ and the adjacency matrix $\boldsymbol{A}$ can be used together with the node feature matrix $\boldsymbol{X}$ as a viable option for digital graph representation. Figure 1.3 depicts a simple graph and its respective matrix representation.



$$\begin{bmatrix} 1 & 0 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 2 & 0 \\ 0 & -1 & -1 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Figure 1.3 An example of a heterogeneous oriented graph with $\boldsymbol{L}, \boldsymbol{D}, \boldsymbol{A}$ matrices (from left to right). In the case of an oriented graph, the degree matrix only takes into account the incident edges from the node (outdegree).

These graph representations, which represent graphs without loss of information, have several drawbacks. The number of graph nodes in practical applications can be in the order of millions. Moreover, most natural graphs tend to have the number of edges linear with the number of vertices. That leads to big and sparse matrices with most of the elements set to 0 [7].

Consequently, graph algorithms that work with these representations suffer from high computation cost. Furthermore, some machine learning methods, such as clustering, do not work well with high dimensional data.

1.2.1 Graph embedding

To tackle the above mentioned issues, there exist methods that convert the graph data into a low dimensional space whilst maximally preserving properties like structure and information stored in graph elements. This conversion process is called *graph embedding*.

Embedding graphs into low dimensional spaces is a daunting task, mainly due to the diversity and flexibility of practical graphs. For example, the representation of molecules may lead to small and sparse graphs, whereas a social network could form a big, dense graph structure. Hence, it is quite difficult to find one silver-bullet technique.

As a consequence, there exist many approaches to graph embedding, such as *Laplacian matrix factorization*, *graph kernel methods*, *deep learning (with/without random walk)* or *generative models* [8]. Each of these methods brings different insights into the graph structure. This thesis will focus on the deep learning approach, and precisely, deep convolutional neural networks, i.e., without random walk.

The selection of a preferred method depends on the embedding output that is usually given in accordance with the expected result. Depending on the task that needs to be solved, the output granularity differs. There are four variants of graph embedding output [8]:

- **Node embedding** is encoding nodes as low-dimensional vectors that summarize their position and information about local graph neighborhood. The goal is to project nodes into a latent space where geometric relations in this latent space correspond to relationships. This embedding can be used for solving problems, such as *node classification* or *node clustering*.
- **Edge embedding** aims to represent edges as an embedded vector. This variant projects both ending nodes together with additional edge information like type and orientation. It is useful for prediction on probable node relationships - *link prediction*.
- **Whole-graph embedding** embeds the graph G into a n-dimensional space in a way that the graph-level similarity is preserved. The key challenge of whole-graph embedding is to find the compromise between expressiveness of embedding and efficiency of algorithm. This embedding is commonly used for solving *graph classification* or *graph clustering* problems.
- **Hybrid embedding** is a combination of aforementioned methods. It can be used for embedding different graph structures, for instance node and edge sequences, communities in graph, paths. This embedding can be used for *community identification* or *graph-level information extraction*.

Chapter 2

Deep Graph Learning

This chapter provides an overview of deep learning methods designed specifically to process graphs. Section 2.1 presents fundamental concepts of deep learning and introduces the models for processing euclidean data, such as images and sequences. Section 2.2 describes the convolution operation and explains how is the convolution used in convolutional neural networks, a popular method for image processing. Section 2.3 contains a description of convolution applied in the context of graphs. Section 2.4 further expands on this topic and explains how are the graphs neural networks (GNN) build. The chapter ends with an explanation of the supervised and unsupervised approaches to training the GNNs.

2.1 Deep learning fundamentals

Deep learning is a class of machine learning algorithms based on computational models composed of multiple processing layers capable of learning complicated data representations with multiple levels of abstraction. Artificial neural networks are popular realizations of those multilayer hierarchies. In recent years, the growing computational power of graphics processing units has allowed the training of deep neural networks with many hierarchical layers. Also, the availability of large training datasets helped improve the ability of model generalization. As a result, the neural network models have brought significant improvements in a variety of practical tasks, from speech recognition and translation to image analysis and computer vision.

The power of neural networks lies in their ability to leverage the statistical properties of the data and distil the important features from it without any human supervision. There is no need to select the features manually, as other machine learning methods usually require, which is one of the most significant advantages of this approach. Book Deep Learning by Goodfellow et al. [9] provides a good background on this topic. The following list presents a short sum-up of essential definitions from this book:

Neuron (Perceptron) is the basic building block of neural networks. The input is an n-dimensional vector $\boldsymbol{x} = (x_1, \ldots, x_n)$, and the output is scalar value $y \in \mathbb{R}$. A neuron has three different attributes: a *weight vector* $\boldsymbol{w} = (w_1, \ldots, w_n)$, a *bias* $b \in \mathbb{R}$ and *activation function* f. Internally, the output value y is computed as $y = f(b + \sum_{i=1}^{n} x_i w_i)$. Weight $\boldsymbol{w}$ and bias b are denoted as *trainable variables*. Their values are usually randomly assigned at the beginning of the training and then continuously adjusted during the training process to reflect the data and capture the learned knowledge.

Activation function f is a non-linear function inside the neuron that "activates" the neuron output y and keeps it within a reasonable range. Some examples of activation function are ReLU, i.e., $f(x) = max(0, x)$, sigmoid, i.e., $f(x) = 1/(1 + e^{-x})$, or hyperbolic tangent.

Neural network layer is a group of neurons at the same "depth level" in a neural network. With the exception of recurrent networks, there are no connections between the neurons within this group. The neurons of a particular layer process the previous layer's outputs and produce inputs for the next one. The shape of the first layer of the neural network, i.e., the input layer, is determined by the data that network processes. The type of last layer, i.e., an output layer, depends on the task that the neural network needs to solve (e.g., binary classification, regression). All the in between layers the input and output layer are called hidden, and their purpose is to enable a model to learn complex tasks. The size of each hidden layer and depth of the neural network influences the number of trainable parameters and, thus, affect the model learning capacity. Too small and shallow models do not have the capacity to approximate the output properly. On the other hand, too deep networks have significant problems with information propagation through all the layers.

Loss function is used to determine the error between the output of the neural network and the real value from training data during the training process. The function expresses how far the computed output is from the ground truth. The selection of loss function depends on the task that the neural network solves. For example, *mean squared error* – $f(x, y) = (x - y)^2$ is a commonly used loss function for continuous output, i.e. regression, while *cross-entropy* is used for classification tasks.

Forward propagation is a term for computing the output of a neural network. Each hidden layer accepts the input data, processes it and passes the output to the successive layer until the output layer is reached. Forward propagation is used during training to get a network output that is compared with the actual value using the loss function. Further, forward propagation is used for model predictions after the training. This process is called *model inference*.

Backpropagation is the process of updating trainable neural network variables during the model training phase. It can be seen as a reverse process to forward propagation. Backpropagation computes the gradient of the *loss function* with respect to the trainable variables, i.e., weights and biases, of each network layer by the chain rule, starting from the output layer and iterating backwards to the input layer. The outcome of the backpropagation algorithm is that the trainable variables are updated accordingly to minimize the loss function, which is in the direction of gradient descent. This process is repeated for every input-output pair from the training set.

2.1.1 Selected types of neural networks

The most basic type of neural network is **Multilayer perceptron** (MLP), which serves as a base of all other types of models. The model consists of at least three *fully connected layers*, where every neuron in each layer is connected to all neurons in the successive layer. MLPs are poorly scalable and computationally expensive, and thus are not really suitable for processing big and complex inputs. In practice, the fully connected layers are often combined with more sophisticated approaches and they usually learn from the output produced by the other methods. MLP is a typical example of a *feed-forward network*, where the information moves only from the input layer directly through any hidden layers to the output layer without any cycles. The expected input of MLP is a *vector*.

Recurrent Neural Network (RNN) is an example of a network that contains loops. Hence it does not belong to the feed-forward network category but into *recurrent networks*. The RNNs are able to process sequences of vectors on the input. The training process as well as model inference run in iterations where, in each iteration, the next element from the input is processed. The building block of RNN is the *recurrent layer* that propagates the output (in addition to the forward pass) back to the cell input for the next iteration. The backward loop is led through the state memory that allows storing important information about the previous iterations. The input of the recurrent layer consists of the new data to analyze and the state of this memory. The RNNs are useful for processing sequential data, such as speech, voice or text.

Convolutional neural network (CNN) is a type of network that was specifically designed to process image data. The input layer expects a 3D vector with *height* $\times$ *width* $\times$ *channels* dimensions. The network uses convolution layers to allow the processing of huge inputs. Using the convolutional layer brings two advantages. Firstly, it reduces the number of trainable parameters, which leads to the possibility to build much deeper networks since the computation complexity is reduced. Secondly, the convolution allows the network to recognize and learn local spatial patterns regardless of their position. The CNNs belong to the *feed-forward* network model category. A more detailed description of the convolution process is provided in the next section.

2.2 From Euclidean to graph data

The previous section introduced three different neural networks, each specialized for a specific type of input data. This specialization of models leads to better predictive performance, lower training time and better generalization since the model can leverage the characteristics of that particular data type. When labeling images, the model can take advantage of the fact that an object has the same meaning, whether it is in the top-left or bottom-right corner.

Therefore, the CNN models that use convolutions achieve much better results in image processing than regular MLP. In the case of text, the sentence tokens, e.g., words or letters, have their natural order, so the RNN models process the tokens in the iteration. That allows tokens to pass the information on the successive elements. However, the information from the beginning of the sequence fades away after a few iterations. Some words (e.g., not) that significantly affect the rest of the sentence can be washed out after a while. Therefore, transformers that have the ability to give “attention” quality to input sequence consequently outperform the RNN models in the task of text comprehension [10]. Models that reflect regularities, symmetries and constraints of input data have far better results than those that do not take that into account. These inherent characteristics of particular data types are been denoted as inductive biases.

So far, the models have taken into account the inductive bias of Euclidean data only. *Euclidean data* is data that can be sensibly plotted in n-dimensional linear space, which is the case of image or text data. However, some data does not map neatly into $\mathbb{R}^n$ space. Such data can be embedded into the physical shape to fit n-dimensional space but only with some consequences since it is not the data natural representation. Graphs are a prime example of non-euclidean data, i.e., data without underlying Euclidean or grid-like structure. The field of deep learning specializing in the processing of non-euclidean data types is called *Geometric deep learning* [11].

In the case of graph processing, one needs to design a new class of models that would be able to reflect the relational inductive biases of graph components. Such models are called **Graph neural networks** (GNN). GNNs allow analyzing graph structures in their native form rather than reducing the representation to the lower-dimensional space before processing it.

Most of the GNN models can be partitioned into two classes based on the information diffusion mechanisms. It can be either *recurrent* or *convolutional (feed-forward) approach* [2]. In this thesis, the focus will be on the latter, where the information diffusion is accomplished by stacking multiple *graph convolutional layers* on top of each other. Such networks are denoted as **Graph Convolutional Networks**. The inner workings of a graph convolutional layer could be explained through the optics of better-known image convolutions.

2.2.1 Image convolution

Formally, convolution is defined as an operation where a filter function g applied to the input function f produces function $g * f$ as a filtered result. Formally, it is defined as:

$$(g * f) = \int f(t-a)g(a)da \tag{2.1}$$

In the case of image convolution, the input is a 2-dimensional vector. Therefore, the convolutional operation is also generalized to two dimensions:

$$\boldsymbol{K} * \boldsymbol{I}_{i,j} = \sum_{m,n} \boldsymbol{I}_{i-m,j-n}\boldsymbol{K}_{m,n}, \tag{2.2}$$

where $\boldsymbol{I}$ is the input image, and $\boldsymbol{K}$ is called filter or kernel. Usually, the input image has a whole vector of values for a single pixel representing channels (i.e., RGB), but each channel has its own filter $\boldsymbol{K}_c$. The process of image convolution consists of 2 primary operations [12]:

1. **Applying kernel** of the size $w \times h$ on the input image. This operation aggregates the information from the pixel's neighbourhood into each s-th pixel, where s is the kernel stride. For example, if the kernel stride $s = 2$, then the aggregation is applied to every other pixel, and the image size is reduced to half. The number of trainable parameters is equal to $w \times h \times o$, where o denotes the number of output channels. Those channels are called also feature maps. This allows the neural network to produce multiple filters that are applied simultaneously. The non-linear function is usually applied after kernel operation.

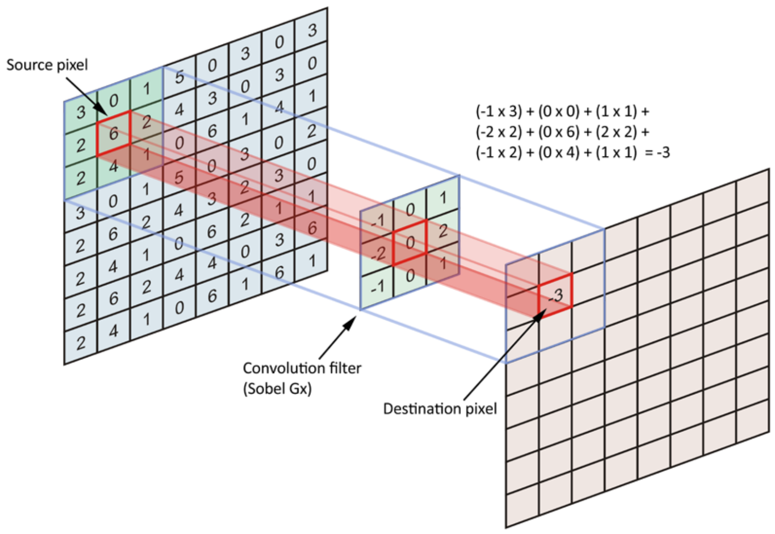

Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel

Figure 2.1 An example of applying kernel on the input image. [13]

2. **Pooling** is a similar operation to convolution, but a fixed operation is performed instead of kernel multiplication. Pooling reduces the dimension of input data by combining the outputs of surrounding pixels. However, instead of aggregation, pooling either singles out only the most important values (max pooling) or averages all the values into one. Pooling layers can be intertwined in between the kernel application layers or can be used at the end of the convolutional process.

Compared to fully connected layers, where the number of trainable parameters is equal to the size of the input, the convolutional layers have only $w \times h \times o$ trainable parameters. The same weights are reapplied over the whole input, which brings two significant advantages. The first one is that a convolutional network respects *translational invariance*. It means that patterns are recognized irrespective of their position. The second advantage is that the number of parameters is significantly reduced, so convolutional networks are usually deeper and allow processing of big images.

2.3 Graph convolution

As discussed in Section 1.2, graphs can be represented by two matrices: matrix $\boldsymbol{X}$ containing node features in the rows and adjacency matrix $\boldsymbol{A}$ describing the node relations. These matrices are used as input of the neural network. Therefore, one could get an idea to define the graph neural network simply as an MLP model and use flattened $\boldsymbol{X}$ and $\boldsymbol{A}$ as [3]:

$$y = MLP(\boldsymbol{X}_1 \oplus \cdots \oplus \boldsymbol{X}_{|V|} \oplus \boldsymbol{A}_1 \oplus \cdots \oplus \boldsymbol{A}_{|V|}) \tag{2.3}$$

However, it should be clear from the previous discussion that this network model has several drawbacks. Firstly, it does not respect inductive biases of the graph structure on the input at all. Secondly, it expects graphs of identical size, and it does not scale up well for really big graphs, i.e., big input leads to a combinatorial explosion of vector operations with trainable variables [3].

When processing images, the convolutional operation was used to reduce the number of trainable parameters and enable sharing of the same parameters over the whole image to respect the translational variation. Similar operation can be also defined for graphs but it has to meet additional requirements.

Graphs are much more irregular and flexible structures compared to images. Each graph node can have a different number of neighbours, while images have a strict grid-like structure with each pixel having exactly 8 neighbours (except the border pixels). Also, images can be inserted into the 2-dimensional grid, and the kernel in convolutional operation can move up, down, right or left, which is not possible in the case of non-euclidean graphs, which do not have any notion of the direction.

Moreover, the graph convolution needs to preserve the isomorphism transformation. If there are two isomorphic graphs G and H, the result of the convolutional operation should be the same in both cases. When looking back on the input representation of the graph, one could notice that both matrices $\boldsymbol{X}$ and $\boldsymbol{A}$ introduce an inherent node ordering by stacking the node information in a specific order. Therefore, any convolutional transformation defined upon the graph input should produce the same result without any regard to the node ordering of input matrices. To rephrase this constraint in the graph language, convolution operation should enforce isomorphism-preserving transformation. The following section reformulates the isomorphism-preserving requirement in a more formal way.

2.3.1 Permutation invariance and equivariance

To describe the restrictions of graph convolutional operation, one has to define an operation that shuffles the node ordering of input matrices into a different one. Such operation is known as **permutation**. An example of 4 node permutation, out of 4! possibilities, could be $\pi = (4, 3, 1, 2)$ that reorders the four nodes in the following manner:

$$\pi([x_1, x_2, x_3, x_4]) = [x_4, x_3, x_1, x_2]$$

Permutation can also be defined as a $\{0, 1\}^{n \times n}$ matrix. Permutation matrices $\boldsymbol{P}$ have only single 1 per every row (or column) and zeros elsewhere. They can be applied to permute rows of matrix $\boldsymbol{X}$ like in the following example:

$$\boldsymbol{P}_{(4,3,1,2)}\boldsymbol{X} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \end{bmatrix} = \begin{bmatrix} x4 \\ x3 \\ x1 \\ x2 \end{bmatrix} \tag{2.4}$$

The intention of the following paragraphs is to design functions over graph representation matrices where the result does not depend on the node ordering. For a start, only the node feature matrix $\boldsymbol{X}$ (graph with no edges and only nodes) is considered. The designed function $g(\boldsymbol{X})$ should always return the same result without regard to the permutation of the node features [1]:

$$g(\boldsymbol{PX}) = g(\boldsymbol{X}) \tag{2.5}$$

Such function $g(\boldsymbol{X})$ is then denoted as a **permutation invariant** function. Permutation invariance is sufficient for obtaining the outputs on the level of the entire set of nodes. However, it is not a sufficient criterion for a node-level task since it does not require the result to keep the ordering of the nodes the same. Thus, it is impossible to identify the specific node outputs after the function application. A typical example of a permutation invariant function is any aggregator, e.g., *sum*, *maximum* or *average*.

For node-level tasks, the designed function $f(\boldsymbol{X})$ should not be allowed to shuffle the node order. The function needs to adhere to a more strict assumption [1]:

$$f(\boldsymbol{PX}) = \boldsymbol{P}f(\boldsymbol{X}) \tag{2.6}$$

If this expression holds true for all permutation matrices $\boldsymbol{P}$, then the function $f(\boldsymbol{X})$ is called **permutation equivariant**. Permutation equivariance allows permuting the nodes before or after the function application. Equivariance mandates that each node's row is unchanged by the function f. Therefore, one can think of equivariant set functions as local transformations of each node $\boldsymbol{x}_i$ (row of $\boldsymbol{X}$) into a vector $\boldsymbol{h}_i$ [1]:

$$f(\boldsymbol{h}_i) = \psi(\boldsymbol{x}_i) \tag{2.7}$$

The function ψ can be any function applied in isolation to each node. Stacking $\boldsymbol{h}_i$ yields $\boldsymbol{H} = f(\boldsymbol{X})$. For the adjacency matrix $\boldsymbol{A}$, one needs to appropriately permute both rows and columns of $\boldsymbol{A}$ because each of them indexes nodes in inherent order. The permutation matrix $\boldsymbol{P}$ has to be applied twice. i.e., $\boldsymbol{PAP}^T$. The adjusted definitions of permutation invariance $g(\boldsymbol{X}, \boldsymbol{A})$ and permutation equivariance $f(\boldsymbol{X}, \boldsymbol{A})$ look like this:

$$g(\boldsymbol{PX}, \boldsymbol{PAP}^T) = g(\boldsymbol{X}, \boldsymbol{A}) \tag{2.8}$$

$$f(\boldsymbol{PX}, \boldsymbol{PAP}^T) = \boldsymbol{P}f(\boldsymbol{X}, \boldsymbol{A}) \tag{2.9}$$

2.3.2 Graph neighborhood as a convolution filter

If a convolution filter function is either *permutation invariant* or *permutation equivariant*, the graph convolution operator respects the isomorphism-preserving transformation. To make sure that the convolutional operator can be used to produce node-level output, one has to choose the latter. *Permutation equivariant* function can be any arbitrary function applied to each node separately. The convolution filter has to aggregate the localized information about node features while respecting the graph structure. This can be achieved by applying *node's neighbourhood* (see Section 1.1), whose definition can be expanded to aggregate the node features. The adjacency information is explicitly expressed by aggregating only adjacent nodes. The expanded definition is:

$$\boldsymbol{X}_{\mathcal{N}_i} = \{\boldsymbol{x}_j : j \in \mathcal{N}_i\}, \tag{2.10}$$

where $\mathcal{N}_i$ denotes the i-th node neighbourhood, and $\boldsymbol{x}_j$ is a neighbour feature vector. Then, any local function g can be applied over this set as $g(\boldsymbol{x}_i, \boldsymbol{X}_{\mathcal{N}_i})$ [1]. By applying this function g locally over all neighbourhoods, one can construct a permutation equivariant function $f(\boldsymbol{X}, \boldsymbol{A})$ as:

$$f(\boldsymbol{X}, \boldsymbol{A}) = \begin{bmatrix} g(\boldsymbol{x}_1, \boldsymbol{X}_{\mathcal{N}_1}) \\ g(\boldsymbol{x}_2, \boldsymbol{X}_{\mathcal{N}_2}) \\ \vdots \\ g(\boldsymbol{x}_n, \boldsymbol{X}_{\mathcal{N}_n}) \end{bmatrix} \tag{2.11}$$

The permutation equivariance is ensured only when the function g does not depend on the order of the vertices in $\boldsymbol{X}_{\mathcal{N}_i}$. As a consequence, a local function $g(\boldsymbol{x}_i, \boldsymbol{X}_{\mathcal{N}_i})$ needs to be *permutation invariant*. To sum it up, the convolutional filter function is constructed as a permutation-equivariant function $f(\boldsymbol{X}, \boldsymbol{A})$ that is composed by locally applying the permutation-invariant function $g(\boldsymbol{x}_i, \boldsymbol{X}_{\mathcal{N}_i})$ [1].

This approach brings similar advantages as the convolution function in CNNs. The local function $g(\boldsymbol{x}_i, \boldsymbol{X}_{\mathcal{N}_i})$ usually contains trainable parameters that are applied all over the graph, so that the local pattern can be learned and recognized no matter their position in the graph. This also significantly reduces the number of parameters required by GCN, as the same parameters are being reused across all nodes. Moreover, it effectively and efficiently combines the information of all nodes and graphs in the dataset to learn a single function. The permutation invariance of the local function $g(\boldsymbol{x}_i, \boldsymbol{X}_{\mathcal{N}_i})$ also elegantly solves the problem with variable neighbourhood shape thanks to the aggregation properties of permutation invariant functions [2].

2.4 Graph convolutional network

The application of a function $f(\boldsymbol{X}, \boldsymbol{A})$ in GCN is a complementary operation to applying kernel in CNN. From the perspective of neural network models, both operations are seen as convolutional layers. The GNN usually contains multiple convolutional layers, similarly to the CNN. The reason for using multiple convolutional layers is to diffuse the information to more remote parts of the graph (or image). This diffusion mechanism is also denoted as a "message passing" algorithm in the domain of GNN.

2.4.1 Message passing

Each convolutional layer in GNN can be seen as one iteration of the message passing algorithm, which outputs a *hidden embedding* $\boldsymbol{h}_u^{(k)}$ for each node $u \in V$. This embedding represents information aggregated from u's graph neighbourhood $\mathcal{N}_u$. The embedding of the node u in the $k+1$ layer can be expressed as [3]:

$$\boldsymbol{a}_u^{k+1} = \mathrm{AGGREGATE}^{(k)}\left(\left\{\boldsymbol{h}_v^{(k)} : v \in \mathcal{N}_u\right\}\right) \tag{2.12}$$

$$\boldsymbol{h}_u^{(k+1)} = \mathrm{UPDATE}^{(k)}\left(\boldsymbol{h}_u^{(k)}, \boldsymbol{a}_u^{k+1}\right) \tag{2.13}$$

Both UPDATE and AGGREGATE are arbitrary differentiable functions, and $\boldsymbol{a}_u^{k+1}$ is the aggregated message from u's adjacent nodes. Superscripts distinguish the embeddings and functions at different iterations of the message passing algorithm. The function UPDATE is semantically equivalent to the local permutation invariant function $g(\boldsymbol{x}_i, \boldsymbol{X}_{\mathcal{N}_i})$.

At the $k+1$ iteration, the AGGREGATE function gathers k-th embeddings of the nodes in the u's neighbourhood and generates a message $\boldsymbol{a}_u^{k+1}$. The UPDATE function combines the message with the previous embedding $\boldsymbol{h}_u^{(k)}$ of the node u and creates a new node embedding $\boldsymbol{h}_u^{(k+1)}$. In the first iteration, where $k = 0$, the embeddings are initialized with input node features, i.e.

$\boldsymbol{h}_u^{(0)} = \boldsymbol{x}_u, \forall u \in V$. After running all K iterations, where K corresponds to the number of convolutional layers in the network, the network generates the final embeddings for each node:

$$\boldsymbol{z}_u = \boldsymbol{h}_u^K, \forall u \in V \tag{2.14}$$

To give a specific example of UPDATE and AGGREGATE operators, a basic GNN model [3], which is a simplification of the original GNN model, can be presented:

$$\boldsymbol{h}_u^{(k+1)} = \sigma\Big(\boldsymbol{W}_{\text{self}}^{(k+1)} \boldsymbol{h}_u^{(k)} + \boldsymbol{W}_{\text{neigh}}^{(k+1)} \sum_{v \in \mathcal{N}_u} \boldsymbol{h}_v^{(k)} + \boldsymbol{b}^{(k+1)}\Big) \tag{2.15}$$

Matrices $\boldsymbol{W}_{\text{self}}^{(k+1)}, \boldsymbol{W}_{\text{neigh}}^{(k+1)} \in \mathbb{R}^{d^{(k+1)} \times d^{(k)}}$, and bias $\boldsymbol{b}^{(k+1)} \in \mathbb{R}^{d^{(k+1)}}$ are trainable variables, and σ denotes an activation function.

In the provided definition of node neighbourhood $\mathcal{N}_u$ (see Section 1.1), the neighbourhood does not take into account the node itself. Because of that, the node features need to be combined with the neighbourhood via the UPDATE function that can have its own weights. This way, the information coming from the node itself can be differentiated from the neighbourhood information. On the other hand, such approach has the inclination to overfitting [3].

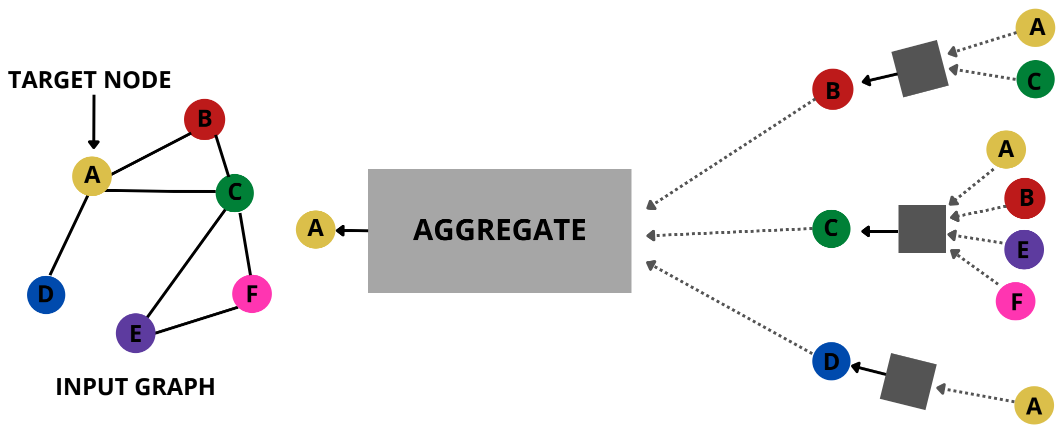


Figure 2.2 The illustration of the message passing algorithm from the node's point of view. The diagram on the right shows the second iteration of the message passing process. The messages coming from A's neighbours (i.e., B, C, D) are based on information aggregated from their respective neighbourhoods. This way, node A aggregates information from the whole graph. The computation graph forms a tree structure by unfolding the neighbourhood around the target node. [3]

The iterative application of message passing allows spreading the information across the graph in the form of aggregated messages. After the first iteration, every node embedding contains information from its local neighbourhood. After the second iteration, every node embedding contains information from its 2-hop neighbourhood (see Figure 2.2). As these iterations progress, each node embedding incorporates more information from further nodes of the graph [2].

Each iteration represents a new convolutional layer. If there are K iterations of the message passing algorithm, the GCN contains K convolutional layers, and every node embedding gathers information from its K-hop neighbourhood. In this case, the multiple-layer architecture not only performs automatic feature extraction, but also serves for context diffusion [2].

On the other side, the message passing paradigm also has some drawbacks [3]. Researchers have empirically found out that message-passing suffers from over-smoothing. Over-smoothing describes a process when a node-specific information becomes "washed out" or "lost" after several iterations of GNN message passing. Distinct node features are wiped at the expense of more common features shared in the node neighbourhood, and representations for all the nodes in the graph can become very similar to one another. Due to this fact, it is really challenging to build deeper GNN models with the ability to aggregate deeper dependencies in the graph since these deep GNN models tend to generate over-smoothed embeddings.

At a more intuitive level, one can see that the *AGGREGATE* and *UPDATE* message-passing structure of GNNs inherently induces a tree-structured computation (see Figure 2.2). This points out another limitation that is their inability to identify cycles consistently and to capture long-range dependencies between nodes [3].

2.4.2 Graph pooling

The output of the message passing algorithm is a set of node embeddings $\{\boldsymbol{z}_u, \forall u \in V\}$, where each embedding consists of locally aggregated information of every node's surroundings. These embeddings are suitable for solving node-level tasks, such as *node classification*. However, certain types of problems require to embedding the whole graph into the one vector $\boldsymbol{z}_G \in \mathbb{R}^n$, where $n \in \mathbb{N}$ is the specified dimension of the graph embedding.

The reduction of node embeddings $\{\boldsymbol{z}_1, \dots, \boldsymbol{z}_{|V|}\}$ into single graph embedding vector $\boldsymbol{z}_G$ is called *graph pooling*. Graph pooling is identical to the pooling mentioned in CNN networks (see Section 2.2.1). The most straightforward approach for generating graph-level embeddings is to take any aggregation, e.g., *sum*, *max* or *mean* of the node embeddings:

$$\boldsymbol{z}_G = \frac{\sum_{v \in V} \boldsymbol{z}_u}{f_n(|V|)}, \tag{2.16}$$

where f_n is some normalizing function (e.g., identity). This simple variant is often sufficient for practical applications [3]. The disadvantage of simple aggregation is that it does not exploit the structural information of the graph. An example of a more sophisticated method that takes that into account is graph coarsening.

Graph coarsening is able to decrease the size of the graph iteratively. Hence, it can be interleaved between the convolutional layers and repeated multiple times. When applying graph coarsening, one has to define a function that maps the node embeddings $(\{\boldsymbol{h}_1^{(l)}, \dots, \boldsymbol{h}_{|V|}^{(l)}\})$ into vector $\boldsymbol{a} \in \mathbb{R}^{+|V| \times c}$ – an assignment over c clusters. This function could be implemented as simple neural network with softmax applied in the last layer to learn the clustering on the nodes. The neural network then produces a soft-membership matrix $\boldsymbol{S}^{(l+1)}$ that returns the probability of a node belonging to cluster class c. The $\boldsymbol{S}^{(l+1)}$ matrix is used to recombine the current graph representation into one of reduced size:

$$\begin{aligned} \boldsymbol{X}^{(l+1)} &= \boldsymbol{S}^{(l+1)^T} \boldsymbol{X}^{(l)} \\ \boldsymbol{A}^{(l+1)} &= \boldsymbol{S}^{(l+1)^T} \boldsymbol{A}^{(l)} \boldsymbol{S}^{(l+1)} \end{aligned} \tag{2.17}$$

The result is a coarser graph representation with a denser adjacency matrix and a set of aggregated node features. This new adjacency matrix represents the strength of association between the clusters in the graph, and the new node feature matrix represents aggregated embeddings for all the nodes assigned to each cluster [3].

2.5 Training process

The GNN training process does not differ from the training process of any other neural network. The problem of training is equivalent to the problem of minimizing the loss function. At the beginning of training, the trainable variables, i.e., weights and biases, are initialized with random values. Afterwards, the variables are adjusted in each iteration to better represent the training data by minimizing the defined loss. The optimization of the loss function is performed using *gradient descent methods* (e.g., SGD, ADAM) that approximate the objective function gradient and move in the descent direction to reach the function minimum. An adjustable *learning rate* parameter determines the step size of each iteration.

The learning process can be divided into two main categories based on the learning task. The first one is *supervised learning*, which learns to produce desired output by approximating the “ground truth” labels that need to be provided with training data. The second category is *unsupervised learning*, which learns by discovering patterns in the unlabeled data using, for example, generative models.

2.5.1 Binary classification

Binary classification is an example of the *supervised learning* approach. The neural network learns to classify the output as one of the two mutually exclusive classes. For example, the GNN accepting behavioral graphs on the input (see Section 4.2) could learn to distinguish malware from clean graphs. The most commonly used loss function is **binary cross-entropy**:

$$BCE(\boldsymbol{x}, \hat{\boldsymbol{x}}) = -\frac{1}{n}\sum_{i=1}^{n}\big(\hat{\boldsymbol{x}}_i \ln(\boldsymbol{x}_i) + (1-\hat{\boldsymbol{x}}_i)\ln(1-\boldsymbol{x}_i)\big), \tag{2.18}$$

where $\boldsymbol{x}_i$ represents the value produced by the neural network and $\hat{\boldsymbol{x}}$ is the “ground truth” label.

2.5.2 Variational autoencoder

Autoencoder is a type of generative model that is used for learning on unlabeled data – *unsupervised learning*. The model learns to efficiently encode data into low-dimensional representations (i.e., embeddings) by attempting to regenerate the input from compressed embedding. The bottleneck embedding should ignore the insignificant “noise” in the data and extract only valuable properties.

Autoencoder architecture consists of two parts: an *encoder* that maps the input into the embedding and a *decoder* that maps the embedding to a reconstructed output. The training process consists of copying input X from the encoder, generating the low-dimensional embedding, and producing the decoder output X' from it. Afterwards, the loss function compares how the regenerated output differs from the provided input.

Practical applications have proven that it is more advantageous to represent the embedding not as a single latent vector but rather as a probability distribution. *Variational autoencoder* (VAE) is an extension of classical autoencoder models that implements this idea. It consists of two parts:

- **Encoder**: This part is responsible for encoding input data $\boldsymbol{X}$. It generates a mean $\boldsymbol{\mu} \in \mathbb{R}^d$ and a variance parameter $\log \boldsymbol{\sigma} \in \mathbb{R}^d$ for a single embedding using two separate neural networks. It defines a distribution $q_\phi(\boldsymbol{Z}|\boldsymbol{X})$ that is used to sample latent embeddings $\boldsymbol{z}$.
- **Decoder**: The decoder takes a latent representation $\boldsymbol{z}$ as an input and uses it to specify a prior distribution $p_\theta(\boldsymbol{X}|\boldsymbol{Z})$.

A loss function of the variational autoencoder has two goals. The first goal is to reduce reconstruction errors between the network's input and output. The second goal is to have $q_\phi(\boldsymbol{Z}|\boldsymbol{X})$ distribution as close as possible to $p_\theta(\boldsymbol{X}|\boldsymbol{Z})$. Both goals can be achieved by minimizing the evidence likelihood lower bound (ELBO) [3]:

$$ELBO = E_{q_\phi(\boldsymbol{Z}|\boldsymbol{X})}(\log p_\theta(\boldsymbol{X}|\boldsymbol{Z})) - KL(q_\phi(\boldsymbol{Z}|\boldsymbol{X})||p_\theta(\boldsymbol{Z})), \tag{2.19}$$

where the first term represents the reconstruction likelihood (i.e., binary cross-entropy) and the other term ensures that the learned distribution q is similar to the true prior distribution p (i.e., KL-divergence loss).

Chapter 3

Clustering Methods

This chapter presents an overview of cluster analysis methods useful for the exploratory analysis of unlabeled data. Clustering works with the data in the form of n-dimensional vectors. The caveats of this representation are discussed in the introduction, followed by a description of the most common distance metrics used to measure the distance between two data points. Section 3.2 sums up different approaches to cluster definition and presents three instances of clustering algorithms: K-means, Agglomerative clustering, and DBSCAN. This chapter ends with a discussion on the cluster quality evaluation and the proposal of suitable evaluation metrics.

Clustering is one of the essential tasks in the data mining. It is a form of data analysis which divides the observed data into groups that share some common characteristics. These groups are referred to as *clusters*. The aim of clustering analysis is to construct clusters so that objects in the same cluster are in a certain way more similar to each other than to objects in other clusters. Clustering is a typical example of *unsupervised learning*, where the provided dataset has no labels. It means no supervisor or teacher tells the model what to learn specifically. The model learns valuable properties of the presented data with minimal human supervision.

The objects that clustering methods usually work with are in the form of n-dimensional vectors $\boldsymbol{x} \in \mathbb{R}^n$. Each vector dimension represents a certain feature of the expressed object $\boldsymbol{x} = (x_1, \ldots, x_n)$. In general, the object features can be divided into three categories:

Numeric feature is naturally expressed as a numerical value that describes a measurable quantity. Its domain range can be either *continuous* (e.g., time, height) with an infinite number of different values or *discrete* (e.g., age, size) with values from a finite set.

Ordinal feature values can be split into mutually exclusive and exhaustive (i.e., finite) categories. They tend to be expressed by non-numeric values, but they still can be logically ordered or ranked. An example of an ordinal feature would be grades (A, B, C) or clothing sizes (S, M, L, XL). Since ordinal features can be easily converted into numerical representation, because of their intrinsic ordering, it is possible to treat them similarly to numeric features.

Nominal feature values are also split into categories but without intrinsic ordering. That means they cannot be easily organized in a numerical sequence. The examples of nominal features are: color, place of birth, university, etc. These features are usually encoded with the *one-hot encoding* method, where each possible value is assigned its own dimension. Each nominal value is thus expanded into a *binary feature* that is also a nominal feature but with only two possible states represented by 0 or 1.

When working with real-world data, the vector usually consists of all three types of features. The usual procedure is to convert all features into numerical types. However, each feature has a different span of possible values, or the features can be in distinct measurement units. For example, a person could be represented by a vector with two features: *height* and *weight*. *Height* can be expressed in meters or inches, and the *weight* can be either in kilograms or pounds, which results in a different range of values for any choice. Expressing a vector feature in smaller units leads to a bigger range, giving the particular feature greater emphasis.

Due to a varying range of vector features, some of them could be undesirably given a higher significance. To help avoid the dependence on the value ranges, the data should be **normalized**. Normalization is a process of transforming the data into a particular range, such as $[0.0, 1.0]$. That helps to prevent features with initially large ranges from outweighing attributes with initially smaller ranges [14].

3.1 Distance metrics

Data is split into clusters based on the (dis)similarities between them. However, it is very challenging to define the notion of similarity rigorously. Since the data are represented in the form of n-dimensional feature vectors, they can be interpreted as points in the n-dimensional space, where the similarity between two objects can be expressed in terms of distance between them. A *distance metric* is an important parameter of the clustering algorithm. It defines how the distance of two elements $\boldsymbol{x}, \boldsymbol{y}$ is calculated, which influences the shape of the created clusters.

The most commonly used distance metric is **Euclidean distance**, which measures a distance of two points using classical Euclidean geometry. It is defined as follows:

$$ED(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{3.1}$$

The ED is not scale-invariant, which means that distances between the points might be skewed depending on the units of features. Therefore, the data needs to be normalized before using this metric. The other disadvantage of ED is that it does not work well with high-dimensional data. The reason is a curious phenomenon denoted as the *curse of dimensionality*. In high dimensions, the points essentially become uniformly distant from each other. This phenomenon can be observed for a variety of distance metrics, but it is more pronounced for ED [15]. Nonetheless, the ED is still a reasonable default to choose for experiments.

Another well-known measure is the **Manhattan distance** (sometimes called *city block distance*) which calculates the distance between two points as if they were on a uniform grid. The diagonal movement is not allowed when calculating Manhattan distance, so only orthogonal turns are possible. Its definition is:

$$MD(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{n} |x_i - y_i| \tag{3.2}$$

When compared to ED, the MD is less prone to the curse of dimensionality. Therefore, MD is more appropriate for high-dimensional data than the ED. Another difference is that MD is more likely to assign higher values than ED since it does not find the shortest possible path. MD is also a reasonable choice in case the dataset consists only of discrete and binary features.

The last metric that will be mentioned is called **cosine similarity**. The cosine similarity does not express the distance but rather reflects the angle between data points. It is defined as a cosine of the angle between two vectors. Two vectors pointing in the same direction have a cosine similarity equal to 1, whereas two diametrically opposed vectors have a similarity of -1. A cosine value of 0 means that the two vectors are orthogonal to each other. The formal definition is:

$$CS(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{||\boldsymbol{x}|| \; ||\boldsymbol{y}||}, \tag{3.3}$$

where $||\boldsymbol{x}||$, respectively $||\boldsymbol{y}||$, is the Euclidean norm of a vector. It is defined as $\sqrt{\sum_{i=1}^{n} x_i^2}$, which is conceptually the length of the vector. Formula $\boldsymbol{x} \cdot \boldsymbol{y}$ represents a *scalar product* that sums the products of corresponding vector components together.

CS is less sensitive to the curse of dimensionality than ED. Another important aspect of CS is that it does not take into account the magnitude of vectors. It means that the data normalization beforehand is not necessary. A good example of CS application is the text analysis, where each document is represented as a vector of word counts. Such vector is very sparse, and the feature values depend heavily on the length of the text. Therefore, CS is a suitable measure since it disregards the magnitude of word counts. All three metrics are illustrated in Figure 3.1.

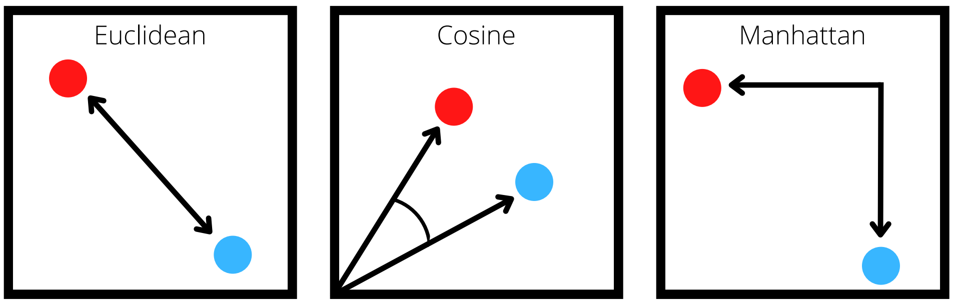


Figure 3.1 The illustration of Euclidean distance, cosine similarity, and Manhattan distance.

3.2 Clustering methods overview

There are a number of different ways how to define a *cluster* as well as how to evaluate the quality of found clustering (see Section 3.3). As the Estivill-Castro pronounced [16]: "*Clustering is in the eye of the beholder*". Researchers have come up with a plethora of various cluster models, and for each of these models, a number of algorithms can be designed. One can apply two different methods on the same dataset and get very distinct results, whilst both of them could be interpreted as correct (Figure 3.2).

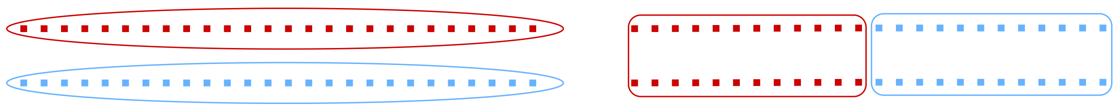


Figure 3.2 On the left side of the picture, there is a clustering method that prefers to keep the close points in the same cluster. On the right side, the clustering method focuses on reducing the distance between any two points in one cluster to a minimum. Both approaches produce reasonable clusters and it is not clear which method is superior.

The following list contains some requirements that should be considered when selecting a clustering algorithm [14]:

- **Discovering arbitrary shaped clusters:** Clusters can be of any shape. Some clustering algorithms discover only clusters that are convex, have similar size, or are flat geometry objects. To detect specifically shaped clusters like non-flat manifold, one needs to consider other methods that work with non-flat geometry.

- **Performance:** The most basic methods work well on small data containing fewer than several hundred objects but are not reasonably effective with large datasets consisting of millions of objects. A scalable algorithm is capable of keeping the time and memory complexity within a reasonable threshold and scales well for huge datasets and a big number of clusters.
- **Intuitive parameters:** Many clustering methods require a number of different input parameters that significantly influence the quality of produced clusters. For example, a user needs to specify the number of clusters that should be created or the stopping criterion. These parameters are often hard to determine and require a certain domain knowledge that is data-dependent. That can make the quality of clustering difficult to control. If one varies the clustering algorithm parameters, the clustering should change in a somewhat stable, predictable fashion.
- **Ability to remove noise:** In real-world use cases, the data contains noise. This noise can be caused by an error in measurements, missing data, unknown features or the presence of outliers in data. Clustering algorithms can be sensitive to such noise and may produce poor-quality clusters. Therefore, it might be beneficial for clustering methods to be able to filter out such data and not assign them to any clusters. Such clustering methods can then be used in applications where those outliers are more interesting than the actual data, i.e., credit cart fraud systems or detection of malicious activities.
- **Incremental and stable clustering:** Sometimes it is required to process data in sequential order as they arrive. Some clustering algorithms cannot deal with new incoming data and update the existing structures. Instead, it is necessary to recalculate the whole clustering from scratch. Also, clustering methods can be sensitive to the ordering of input data or random initialization. Usually, it is more convenient to get predictable clusters that do not depend on the data order or random initialization.
- **Capability of clustering high-dimensional data:** Finding clusters of data objects in a high-dimensional space is challenging, especially considering that such data can be very sparse. Every clustering algorithm eventually reaches its dimensional limit due to the curse of dimensionality. Nevertheless, some clustering methods are less prone to it than others.

Clustering algorithms can be categorized based on their approach toward cluster definition. These categories may overlap so that an algorithm may have features from several categories. Jiawei Han et al., in their book *Data Mining: Concepts and Techniques* [14], provide a relatively organized overview and possible categorization of clustering methods into the following categories:

Partitioning methods divide the data into k groups where $k \in \mathbb{N}$ and k is lower or equal to the number of objects in the dataset. Each group must contain at least one object, and each object must be assigned to exactly one group (there exists an extension called *fuzzy partitioning* that relaxes this criterion allowing one object to belong to multiple clusters). Most of these algorithms require the number of clusters k to be specified in advance, which is considered to be one of the biggest drawbacks. Reaching global optimality requires an exhaustive enumeration of all possible partitions, and it is often very computationally expensive. Because of that, most of the applications work with greedy heuristics that improve the quality of clustering in iterations and approach (only) local optimum. Partitioning methods are usually distance-based and discover geometrically flat clusters exclusively. These methods can be scaled for small up to medium size data sets. Algorithms such as *k-means* and *k-medoids* belong to this category.

Density-based methods define clusters as areas with a high density of data points. The density-based methods have the ability to identify outliers, i.e., data that do not have any close neighbors and can be considered noise. The general idea of density-based methods is to continue growing a given cluster as long as the density (number of objects or data points) in its neighborhood exceeds some threshold. For each point within a cluster, the neighborhood of a given radius has to contain at least a certain number of points. Therefore, the implementations usually require parameters modifying algorithm sensitivity to density and also a number of minimal samples to consider the dense area to be the cluster. These parameters have to be adjusted according to particular data. Density-based methods can discover clusters of arbitrary shapes (e.g., non-flat manifolds) and are scalable enough to perform on very large data sets. Typical practical implementations of this method are *DBSCAN* and *OPTICS* clustering.

Hierarchical methods (also called *connectivity-based methods*) create a hierarchical decomposition of the given set of data objects. There are two possible approaches, i.e., *agglomerative* and *divisive*. *Agglomerative clustering* starts with each object having its own cluster. Then, the algorithm gradually merges the closest clusters together, one after another. *Divisive clustering* works exactly the opposite way, starting with one big cluster of all data and splitting it into smaller and smaller parts. In both cases, a termination condition is usually defined, which stops the process at a certain point. The termination condition can be, for example, a number of expected clusters or a maximum distance of two clusters. *Agglomerative clustering* or *BIRCH* are typical representatives of these methods.

Grid-based methods divide the object space into a finite number of cells in the form of a grid structure. The process of clustering is performed on the grid in the quantized space. The grid-based methods are fast and have low computational complexity that is typically independent of the number of data objects and dependent only on the number of cells in the provided grid. These methods can be integrated with other clustering methods, such as density-based and hierarchical methods. An example of practical implementation is *WaveCluster* algorithm.

Three selected clustering algorithms will be discussed in greater depth in the following sections. Those algorithms are also applied and evaluated in the practical part of the thesis.

3.2.1 K-means

K-means algorithm is a typical representative of the partitioning method. Instead of "looking for the clusters", k-means partitions the whole space into k sub-spaces (Figure 3.3). Parameter k defines the number of expected clusters and it have to be set by the user beforehand.

One can think of k-means as an optimization problem. Given a set of observations $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n)$ where n is the size of the dataset and each $\boldsymbol{x}_i$ where $1 \leq i \leq n$ is a d-dimensional vector, the k-means aims to partition the observations into k disjoint clusters $C = \{C_1, \dots, C_k\}$ in a way that would minimize the criterion known as clusters *inertia* (or *within-cluster sum-of-squares* error). Formally, the objective function expressing inertia is defined as:

$$\arg\min_C \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in C_i} ||\boldsymbol{x} - \boldsymbol{\mu}_i||^2 \tag{3.4}$$

Each cluster is described by $\boldsymbol{\mu}_i$ – the mean of all the samples in the cluster. The mean is commonly referred to as the *centroid*, a cluster centre. The centroid does not have to be one of the data points, although it lives in the same n-dimensional space as the data. The adapted version of k-means, where the centroid is "rounded" to the closest data point, is called *k-medoids*.

Expression $||.||$ denotes the Euclidean distance function. Clusters discovered by *k-means* are convex and isotropic. K-means is not suitable for high-dimensional data.

Finding a global optimum of the objective function from Equation 3.4 is an NP-hard problem [17]. Therefore, k-means implementation applies a greedy heuristic and approaches only the local optimum. For that reason, the algorithm is run multiple times with different random chosen initial centroids. The steps of k-means are:

1. Randomly choose k initial centroids from all samples, i.e. data points.
2. For each sample calculate, the Euclidean distance between the sample and all centroids. Then, assign the sample to the nearest centroid. After this step, all the samples are partitioned to k clusters.
3. Recalculate the cluster centroids by taking the mean value of all the samples assigned to each cluster.
4. Find out the Euclidean distance between the old cluster centroids and a new one from step 3. If the shift of new centroids is below some threshold, stop the process and return the current labels. Otherwise, go back to step 2.

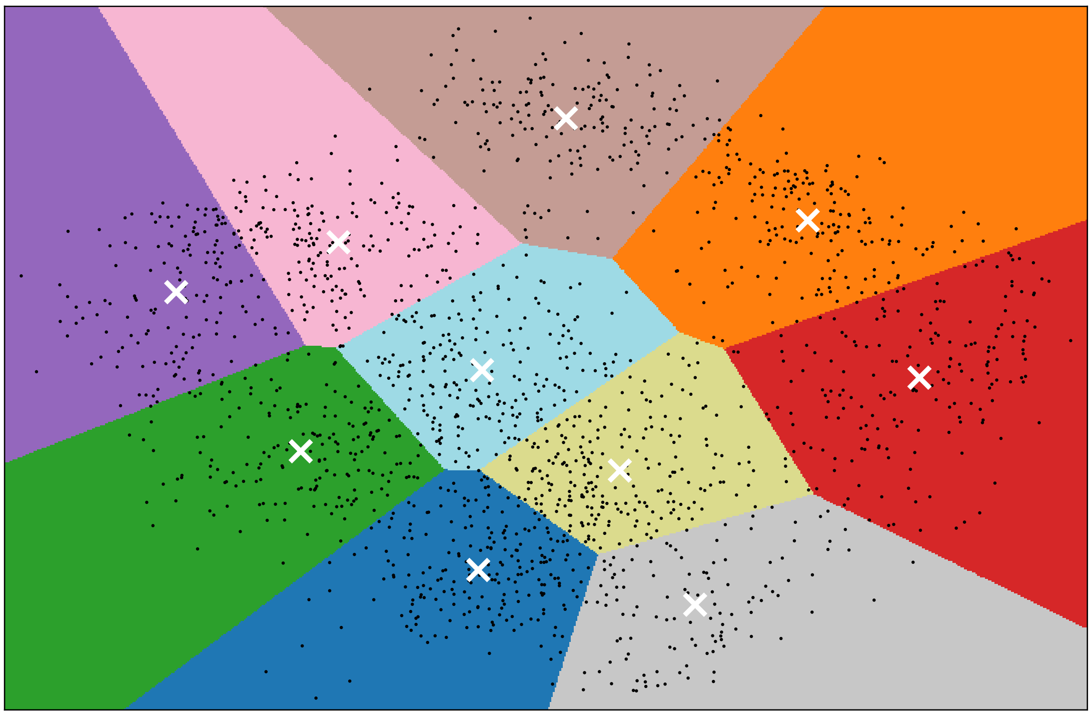


Figure 3.3 This figure illustrates how k-means algorithm partitions the whole 2D space into $k = 10$ chunks. The white marks are cluster centroids, and black dots are 2D data vectors. The background color represents an assignment to a particular cluster. [18]

Given enough time, k-means eventually converges to a local optimum. However, the quality of the resulting clusters depends on the random initialization of the centroids in step 1. The extension of the algorithm called k-means++ addresses this issue by initializing the centroids to mutually distant data points.

3.2.2 Agglomerative clustering

Agglomerative clustering is an implementation of the hierarchical clustering model that creates a “bottom-up” cluster hierarchy. It starts with each object forming its own cluster. Then, it iteratively merges the two closest clusters until all the objects are in a single cluster or predefined termination conditions are satisfied.

Let N denote the number of objects in the dataset. Then, the agglomerative clustering takes $N-1$ iterations at maximum. After the last iteration, the merged cluster contains all the objects from the dataset and becomes the hierarchy's root. The process of agglomerative clustering could be represented via a tree structure with a huge cluster of all objects as the root and clusters containing only one data point as its leaves. A graph that captures the evolution of clustering from its node to the root is denoted as *dendrogram* (Figure 3.4).

In every iteration, the algorithm has to decide which two clusters are the closest and should be merged. So far, the notion of distance was defined only between the two vectors as a distance metric $d(\boldsymbol{x}, \boldsymbol{y})$ (see Section 3.1), but there are multiple ways how to expand this definition to compare the distance between two clusters. The *linkage criterion* determines the distance between sets of observations, i.e., clusters, as a function of the pairwise distances $d(\boldsymbol{x}, \boldsymbol{y})$ between individual points. Some commonly used linkage criteria between clusters A and B are:

- **Single linkage** minimizes the distance between the closest objects in a pair of clusters:

$$D(A,B) = \min_{\boldsymbol{x} \in A, \boldsymbol{y} \in B} d(\boldsymbol{x}, \boldsymbol{y}) \tag{3.5}$$

- **Complete (Maximum) linkage** minimizes the distance between the furthest objects in a pair of clusters:

$$D(A,B) = \max_{\boldsymbol{x} \in A, \boldsymbol{y} \in B} d(\boldsymbol{x}, \boldsymbol{y}) \tag{3.6}$$

- **Average linkage** minimizes the average of distances between all objects in a pair of clusters:

$$D(A,B) = \frac{1}{|A||B|} \sum_{\boldsymbol{x} \in A, \boldsymbol{y} \in B} d(\boldsymbol{x}, \boldsymbol{y}) \tag{3.7}$$

- **Ward's linkage** minimizes the sum of squared differences within a pair of clusters. Similarly to k-means objective function, $\boldsymbol{\mu}_I$ is a centroid of cluster I, and $||.||$ denotes Euclidean distance. Ward's linkage does not allow the use of other than the Euclidean distance metric. The formula is:

$$D(A,B) = \sum_{\boldsymbol{x} \in A \cup B} ||\boldsymbol{x} - \boldsymbol{\mu}_{A \cup B}||^2 - \sum_{\boldsymbol{x} \in A} ||\boldsymbol{x} - \boldsymbol{\mu}_A||^2 - \sum_{\boldsymbol{x} \in B} ||\boldsymbol{x} - \boldsymbol{\mu}_B||^2 \tag{3.8}$$

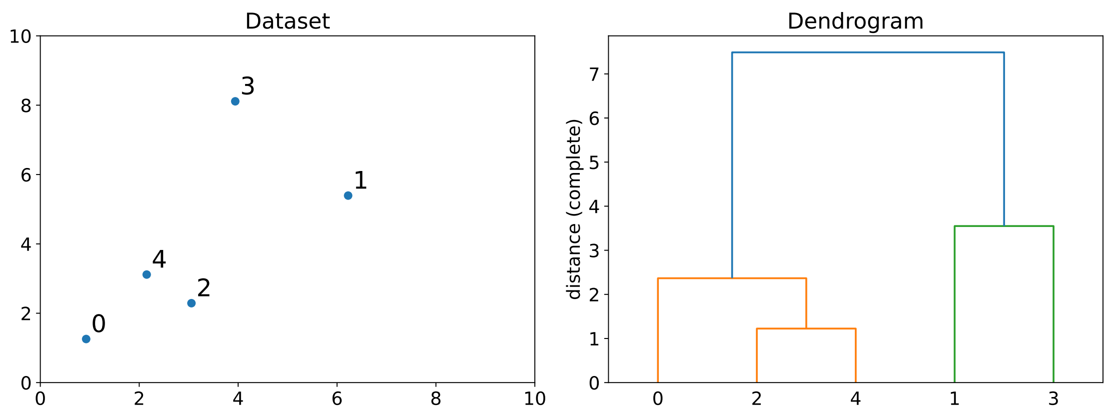


Figure 3.4 An example of the agglomerative clustering process with complete linkage criterion. The left graph shows a 2-dimensional dataset consisting of 5 objects. The right graph depicts the generated dendrogram. The dendrogram y-axis shows at what distance each cluster merged. If one defined a termination condition as a linkage distance equal to 3, the result of clustering would be 3 clusters: {0,2,4}, {1} and {3}.

Agglomerative clustering has a tendency to create a few clusters with uneven sizes that can grow very quickly. This is denoted as “rich get richer” behavior [19]. In this regard, a single linkage is the most pathological strategy, whilst Ward’s linkage clusters the most evenly. On the other hand, the distance metric used in Ward cannot be varied (Euclidean). Thus, an average linkage is a good alternative for non-Euclidean metrics. Despite these facts, the single linkage is often used for larger datasets since it can be computed efficiently and performs well on non-globular data [19].

3.2.3 DBSCAN

The *DBSCAN* (Density-Based Spatial Clustering of Applications with Noise) algorithm defines clusters as dense regions in the data space that are separated by low-density areas. Intuitively, dense regions contain many objects gathered closely together, whilst the object in low-density regions are scattered with greater distances between them. DBSCAN searches for dense areas from the perspective of data points. If the object’s close neighborhood contains at least a certain number of objects, then a cluster can be formed around such object, and it can be iteratively expanded over its neighborhood.

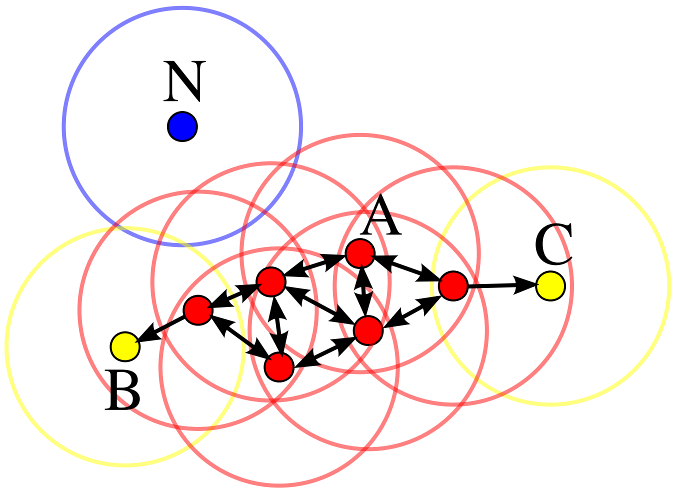


Figure 3.5 The illustration of DBSCAN categorization process with $minPts = 4$. The circles depict the ε-neighborhood. The arrow between the points indicates the direct reachability relation. Red points are categorized as the core objects, yellow as the border objects, and blue as the noise. [20]

Three user-specified parameters define an object neighborhood. The first one is the size of the neighborhood, denoted as $\varepsilon > 0$. The second is the density threshold $minPts$ specifying the minimal number of neighbors necessary to consider the area dense. The third one is a distance metric $d(\boldsymbol{x}, \boldsymbol{y})$ that determines the shape of the object’s neighborhood, e.g., the neighborhood of Manhattan distance has a rectangular shape. After these parameters are specified, the DBSCAN can define the following categories for each object from the dataset [21]:

- An object $\boldsymbol{x}$ is a **core object** if there are at least $minPts$ objects in its ε-neighborhood.
- An object $\boldsymbol{y}$ is **directly reachable** from $\boldsymbol{x}$ if $\boldsymbol{y}$ belongs to the $\boldsymbol{x}$ ε-neighborhood.
- An object $\boldsymbol{x}_n$ is **reachable** from $\boldsymbol{x}_1$ if there exists a path $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ where each $\boldsymbol{x}_i$ is directly reachable from $\boldsymbol{x}_{i-1}$ where $1 < i \leq n$.
- Objects not reachable from any other object are **outliers**.

Then, any core object $\boldsymbol{x}$ together with all its reachable objects form a cluster. Each cluster have to contain at least one core object and at least $minPts$ objects. Objects that are reachable from some core object but do not have $minPts$ neighbors in ε-neighborhood are called *border objects*. Border objects are located on the fringes of a cluster. Figure 3.5 depicts the possible categories and their location within the cluster.

The following list presents a possible high-level implementation of DBSCAN. The data points are categorized in this order [20]:

1. For each object, count the number of neighbors in the ε-neighborhood and identify core objects.
2. Join neighboring core objects into one cluster.
3. Identify the objects that were not categorized as core objects in step 1. Iterate over them and check whether they are directly reachable from some core object. If yes, add them to the corresponding cluster. Otherwise, categorize them as noise.

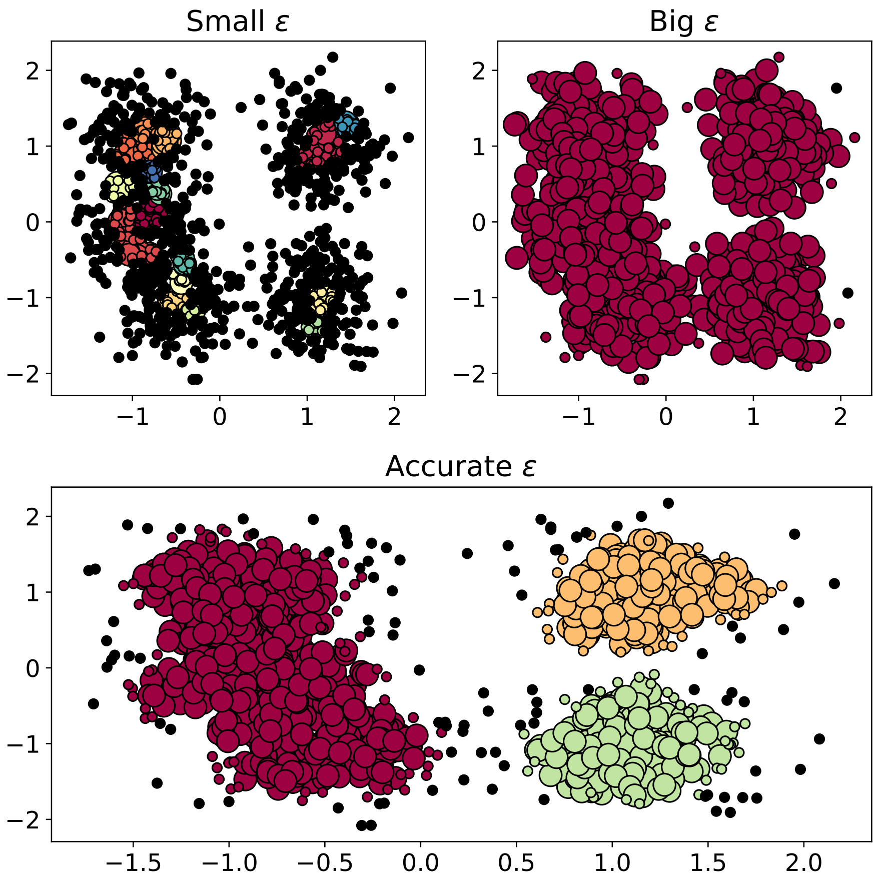


Figure 3.6 An illustration how different ε values affect the final clustering. The color indicates a cluster membership. Larger circles denote core objects, smaller circles denote border objects, and the outliers are indicated by black dots. [19]

The density approach allows DBSCAN to find arbitrary shaped clusters, not only those of convex and isotropic shape [19]. This behavior can be seen on the bottom graph in Figure 3.6, where the prolonged red cluster on the left side is accurately identified as one. Nonetheless, DBSCAN can have trouble identifying clusters if their density varies widely [21]. Ideally, the density parameters ε and $minPts$ would be defined for each cluster specifically, but it is problematic to distil the proper values for every cluster automatically. In practical implementations, the parameters are set according to a simple effective heuristic that determines the least dense cluster from the dataset and sets the parameters correspondingly.

The parameter $minPts$ controls the tolerance of the algorithm towards noise. Increasing the parameter leads to more objects ending outside the clusters, which is ideal for noisy datasets. The ε, together with the distance function $d(\boldsymbol{x}, \boldsymbol{y})$, controls the notion of density for a given dataset and has to be chosen according to the input data. When ε is too small, most data will not be clustered at all and end up classified as noise. If it is too large, it causes close clusters to be merged into one cluster, and eventually, the entire dataset is identified as a single cluster (see Figure 3.6) [20].

The DBSCAN algorithm is deterministic, but the result may change when the input data are provided in a different order. Firstly, cluster labels can change depending on the order in which the clusters are discovered. Secondly and more importantly, border objects in the DBSCAN model can be reachable from more than one cluster. This happens when a border sample has a distance lower than ε to the two core samples in different clusters. In that case, most DBSCAN implementations assign border points to the first cluster they are reachable from [20].

3.3 Clustering quality evaluation

The quality evaluation of clustering results is a daunting task. Firstly, there are no apparent "ground-truth" labels as it is in the case of supervised learning, where the evaluation is an integral part of the model development. Secondly, due to the variety of equally valid cluster definitions, it is not possible to define one specific way how to measure the quality of created clusters.

Clustering can be seen as an exploratory data analysis, so the evaluation might even seem to be an unnecessary step in this process. However, the clustering algorithms are not able to distinguish between bad and good clusterings. If one took, for example, uniformly distributed data, most of the clustering methods would still return some results. Therefore, having some evaluation methods is vital, but the results should always be taken with a grain of salt.

Besides a very valid and valuable *manual* evaluation by a human expert, which is especially helpful for identifying bad clusterings, there exist two other approaches: *intrinsic* and *extrinsic*. Both of them are discussed in the following sections, together with examples of particular evaluation metrics. The list of metrics and their description were taken from the *scikit-learn* clustering documentation page [19].

3.3.1 Intrinsic methods

Intrinsic approach judges the quality of a clustering structure without respect to external information, i.e., in an unsupervised way. It considers two concepts when defining a well-formed set of clusters. The first one is *cohesion*, which determines how close are the observations within a cluster. The second one is *separation*, which determines how distinct or well-separated the clusters are from each other. The drawback of this approach is that each intrinsic measure is also based on a particular notion of cluster. The evaluation is, thus, biased towards algorithms that use the same cluster model. Good evaluation results mean that model specific structure exists in the data, which does not necessarily have to result in effective information retrieval.

Evaluation of two different clustering models requires two different intrinsic measures. Also, any intrinsic measure can be used as an objective function of a clustering algorithm and vice versa. Comparing two measures produced by two algorithms only shows which algorithm results in a better approximation of the optimization problem for that particular model. Because of that, intrinsic metrics are usually used to estimate reasonable model parameters to get the best results for a particular model and data.

3.3.1.1 Silhouette Index

The Silhouette Index (SI) is defined for each object and is composed of two values: the mean distance between an object and all other objects in the same cluster (a) and the mean distance between an object and all the other points in the next nearest cluster (b). SI is calculated as:

$$SI = \frac{b - a}{\max(a, b)} \tag{3.9}$$

To calculate the index for the whole cluster, the mean of all SI values for each object in that particular cluster is taken. The silhouette range is $[-1, 1]$, where the higher score indicates well-separated clusters. SI can be used with any distance metric. In general, SI tends to be higher for convex clusters than other types, such as density-based clusters.

3.3.1.2 Davies-Bouldin Index

The Davies-Bouldin Index (DBI) is defined only on the cluster level. It represents the similarity of two clusters C_i, C_j as a measure R_{ij} that trades off: the average distance between each point in the cluster and its centroid (s_i, respectively s_j); and the distance between the clusters' centroids (d_{ij}). R_{ij} can be constructed as:

$$R_{ij} = \frac{s_i + s_j}{d_{ij}} \tag{3.10}$$

The Davies-Bouldin index is defined as:

$$DBI = \frac{1}{k} \sum_{i=1}^{k} \max_{i \neq j} R_{ij} \tag{3.11}$$

DBI lowest possible score is 0. Values closer to zero indicate better partitioning. DBI is limited only to the Euclidean distance metric and is generally lower for convex cluster types.

3.3.2 Extrinsic methods

Extrinsic approach requires the existence of some external information that was not used during the clustering process. This external information is usually in the form of externally derived class labels. Those labels could be, for example, produced by a human expert. Extrinsic methods measure the degree of correspondence between the cluster and class labels, as is usual for supervised learning. These methods can, for example, evaluate the extent to which a cluster contains objects of a single class or measure the extent to which two objects in the same class are in the same cluster and vice versa.

The extrinsic measures compare the clustering results with the “ground truth” and evaluate whether the manual classification process can be automatically produced by cluster analysis. However, from a knowledge discovery point of view, the reproduction of known knowledge may not necessarily be the intended result of the clustering process, as the clustering might reveal other data structures that do not correspond with the information provided via labels.

3.3.2.1 Purity Index

The purity Index (PI) measures the extent to which the clusters contain a single class. As a first step, the metric selects the most prevalent label within each cluster and counts the number of occurrences in that particular cluster. Then, the occurrences are summed together and divided by the total number of objects in the dataset. Formally, given a set of clusters C, a set of labels M, and a dataset of size N, purity is defined as:

$$PI = \frac{1}{N} \sum_{c \in C} \max_{m \in M} |m \cap c| \tag{3.12}$$

The range of PI is $[0, 1]$. PI does not penalize for too many clusters, so the maximum score of 1 is always possible by putting each object into its own cluster. If the provided data labels are not balanced, meaning some labels are more common than others, PI might return "skewed" results towards higher values, even though the clustering is not separating the clusters properly.

3.3.2.2 Rand Index

Random Index (RI) is a function that computes the similarity of two set assignments: ground truth class assignments L and clustering algorithm assignments C, ignoring permutations. It can be calculated using the formula:

$$RI = \frac{TP + TN}{TP + FP + FN + TN}, \tag{3.13}$$

where TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives. RI range is $[0, 1]$ with lower values indicating different labeling, i.e. worse clustering with regard to "ground truth". The adjusted Rand index [22] corrects for chance and will give such a baseline.

Chapter 4

Behavioral Graphs

This chapter is dedicated to behavioral graphs, which capture the behavior of running programs and applications in the Windows operating system within a small time window. The chapter starts with an explanation of how the graphs are created and why they are useful for malware detection. Then, it continues with the description of the behavioral graph components, what information they can store, and what kind of malware-related labels could be assigned to them. The last section describes three neural network models that process behavioral graphs and produce their low-dimensional representation in the form of 32-dimensional vectors.

4.1 Behavioral shield

The behavioral shield is a technology that was introduced in 2017 as one of the new protection mechanisms in the Avast antivirus program [23]. Its purpose is to monitor the behavior of all running processes on the computer, look for unusual activity, and, in case of very suspicious behavior, stop the threat and notify the user in real-time. Examples of suspicious behavior that behavioral shield aims to detect are: installation of a password capture program that captures the keystrokes, a PDF document trying to download something from the web, or a calculator app trying to delete all the personal documents.

In its default state, the behavioral shield sits in the background and silently monitors selected parts of the operating system. In case of some unusual activity, the behavioral shield is triggered and becomes active. The shield captures executed actions and relations between the individual processes running in the operating system in the form of a graph. This graph can be seen as a snapshot of the state of the operating system within a certain time window. Depending on the severity of the captured actions, the shield calculates a score that denotes the probability of maliciousness of the actions and relations for a given graph. If the score is sufficiently high, the system neutralizes the threat immediately.

Otherwise, the snapshot of graph is submitted to the Avast cloud and is subjected to further analysis. This ensures a continual stream of recent data from user endpoints and allows quick identification and detection of new unknown threats. The submitted snapshots are then processed on the Avast back-ends. Namely, uninteresting activity is filtered out (a snapshot of the whole system would be too big and noisy), the graphs are split into smaller parts, i.e., communities, enriched with other information from internal back-end systems and stored in the graph database.

4.1.1 Motivation

The first antivirus software (AV) products were basically huge virus definition databases that were constantly compared against all the files, URLs, IP addresses and other artefacts that users encountered while using the computer. These databases stored various *Indicators of compromise* (IOCs) for all discovered malware in the form of file hashes, blacklists of IPs and URLs, signature byte sequences identifying malicious code and potentially other forms of static detection.

Although, it is no surprise that the adversaries came up with countermeasures that circumvented these protections and started to use more sophisticated methods. The attackers started to deploy polymorphic malware prone to classical signature-based detection. Instead of statically embedded domain names into the binaries, they used DGAs (Domain Generating Algorithms) [24] to get around the blocklists. They challenged the AV scanning capabilities with file-less malware [25], which uses various techniques to stay for the whole time of execution in the computer memory, making it harder for AV to scan it. File-less malware also leverages legitimate programs and pre-installed system tools ("Living off the Land" technique - LOLBins, LOLScripts) for malicious activities to make the attacks harder to observe.

This forced the AV systems to evolve and respond to the adversaries with better ways of protection. The AVs started to use a combination of the old signature-based approach and new methods aimed at detecting the behavior. The combination is necessary since both approaches have their pluses and minuses. Static detections work very well for precise identification and naming of the threat and are less prone to false positives when compared to behavioral detections. On the other hand, static signatures can only detect previously seen tools and indicators. The hybrid approach allows to leverage the advantages of both methods and makes it harder for malware creators to bypass the protection.

Some of the adversary techniques mentioned above (polymorphism, DGA) can be addressed using "classical" behavioral analysis, i.e., running the sample in an emulator, sandbox, or virtual machine. This type of behavioral analysis is usually used when the AV runs into an artefact that it has not seen before. The execution of the unknown artefact is postponed until the behavioral analysis finishes outside of the user environment. The sample could be either run in a specialized sandboxed environment within the AV engine or sent to the AV back-end systems for evaluation. In both cases, the behavioral analysis is strictly time-limited since the user has to wait for the results, and thus, the malicious behavior might be overlooked.

The other issue with this type of behavioral analysis is that the related artefact does not run in the intended environment, which might cause the execution of malicious actions to be unsuccessful. The adversary might use some fingerprinting technique that recognizes the virtual environment, i.e., anti-VM protections. Further, there might be some dependency on the real user environment, e.g., specific tools have to be installed/running, environment variables need to be set, or the attacker makes a targeted payload for the specific machine. Running the samples in other environments is especially problematic in the case of file-less malware, which is highly interconnected with running OS and split into the chain of actions that have to be executed in a particular order. Therefore, there is a very low chance that AV would capture the whole chain of actions and execute it properly somewhere else. Any of those issues might cause AV not to be able to discover malware in the pre-execution phase.

Considering these facts, most modern AVs introduce an EDR-like functionality (Endpoint detection and response) that detects the threats on the fly directly on the running system. This adds an additional layer of protection for threats unnoticed by regular AV protections and brings multiple benefits. Firstly, it allows killing already running malware that was not discovered in the pre-execution phase. Secondly, it improves the AV's ability to stop a more sophisticated attack by allowing to look at the attack as a chain of malicious actions, not only focusing on the analysis of individual artefacts, which, if discovered in isolation, might appear to be harmless. This is especially true for file-less malware that uses standard system tools as an attack chain. It also gives the AV the ability to detect tactics, techniques, and procedures (TTP) instead of blocking only IOCes that are much easier for the attacker to change (see David Bianco *Pyramid of Pain* [26]). Thirdly, it allows a better reaction to previously unknown threats and adheres more to the modern “assume breach” mentality shift.

4.2 Components of behavioral graphs

A behavioral graph is a snapshot of running processes produced by the behavioral shield. When a suspicious action triggers graph generation, the behavioral shield gathers information about running processes and their interaction with each other as well as with OS during a particular time interval in the form of a graph [27]. The notion of time is not captured explicitly in the graph structure, meaning nodes and relations do not evolve in time as it is in the case of dynamic graphs. A behavioral graph is just a simple graph that depicts all the actions from a particular time window, so it is not possible to tell exactly in which order the actions were executed. Although, there are some implicit cues that suggest the order of actions, e.g. the `node_id` feature allows the topological ordering of the nodes by the time of creation or `SPAWN` relation defining a parent-child relationship between processes indicates which process was created sooner.

At the very beginning, the behavioral graph contains information about all the processes running in the system. Given the complexity of modern operating systems, the graph has hundreds of nodes and relations where only a tiny part of the graph possesses the interesting information regarding the malicious activities happening in the OS. This initial snapshot, also called *fullgraph*, can be naturally split into smaller connected communities - *subgraphs*. The isolated nodes are filtered out, and the connected communities are later even more simplified based on some heuristic that tries to keep only the relevant part of graphs. These simplified connected subgraphs are then used as the input to the learning system.

In general, the behavioral graphs are directed heterogeneous graphs with multi-feature nodes. Each node is characterized by hundreds of features [27], while the edges hold only 1-dimensional information – label. The behavioral graphs may contain cycles as well as loops. An example of the simplified subgraph can be seen in Figure 4.1. It depicts only the most relevant information, i.e., the graph structure, edge labels and node type with its most significant feature (process name, IP address, URL, command-line, . . .). Other node features are proprietary information of the Avast company, and they cannot be presented in this thesis.

The behavioral graph nodes can be divided into different types based on what they represent. Each type of node has its own node features and allows forming different kinds of relations. The following list details the most important node types:

Process represents a running process or application. It is the most complex node and contains many features, e.g., *process id*, *process name*, *process creation time*, or *command-line with arguments* used to start the process. Process nodes are the centres of action and connect together other types of nodes which are mostly just results of their actions. Examples of the edges that can be connected with process node are: `SPAWN` - connects parent and child process, `CODE_INJECT` - indicates that process injected code into another process, `TERMINATE` - the process terminated another process.

Executable represents an executable file from which is the running process instantiated. The file does not necessarily have to be only Windows Portable Executable (PE) but can also be any script (.bat, .vbs, .vba, . . .) or link file (.lnk). The most important attribute is *file hash*, which serves as a file unique identifier for back-end systems. Executable node is usually connected with the process node via `INSTANCE_OF` relation and sometimes mirrors the relations of the corresponding process.

Registry Operation Info node expresses the modification of a specific key in the Windows registry. Relevant attributes are the name of the modified registry key and the new value.

Registry Operation Info node is connected with `PROPERTY` edge to the process node that changed the particular registry key.

Named Object node represents a creation of Windows NT named object. Named objects are used for sharing object handles between processes in Windows. Examples of named objects are mutexes, semaphores, pipes, events, created windows, or file mapping objects. Named Object node's attributes are *name* and *type*. Similarly to Registry Operation Info, Named Objects are connected via edge `PROPERTY` with process node.

Connection node represents a network connection with some service on the internet and contains either *IP address* or *URL* address as its attribute. An example of edge type is `HTTP_GET`, which indicates HTTP type of connection.

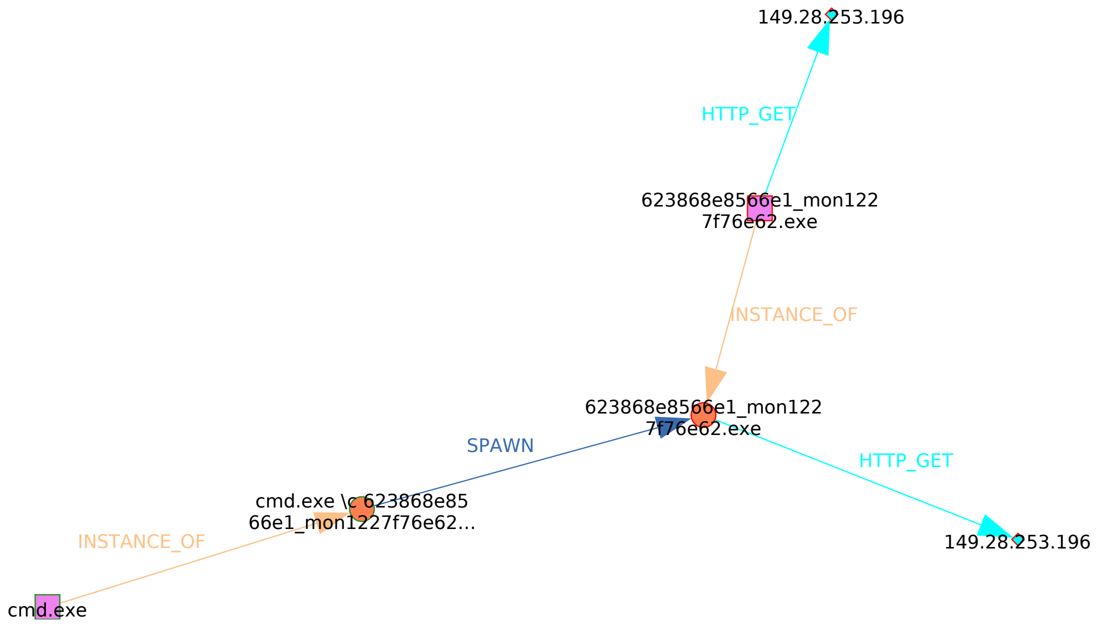


Figure 4.1 This is an example of a behavioral graph from the graph database. Purple square nodes represent *executable* nodes, the orange circles are *process* nodes, and the blue rhombuses are *connection* nodes. The graph describes the following events: Process `623868e8566e1_mon1227f76e62.exe` was started via Windows command-line (`cmd \c [process_name]`), and it established an HTTP connection with the malicious IP address.

4.2.1 Comparing behavioral graphs

For the purpose of comparing two behavioral graphs, the *Jaccard distance* metric is applied to their node sets. The only node feature, which is considered when comparing nodes, is the node type. For example, the behavioral graph from Figure 4.1 is transformed to the set $G_1 = \{$`Process, Process, Executable, Executable, Connection, Connection`$\}$. If one compared this graph with the a similar graph except for the "`cmd` branch" with the node set $G_2 = \{$`Process, Executable, Connection, Connection`$\}$, the resulting Jaccard distance would be $d_J(G_1, G_2) = 2/6 = 0.\overline{3}$.

4.3 Learning on behavioral graphs

Every day, the behavioral shield produces around 1 million behavioral subgraphs that are stored in the graph database. This huge number of graphs creates an opportunity to leverage that amount of data for a machine learning system. The tasks that this system could solve are numerous, from binary classification of graphs into clean/malware class to detection of outliers for discovering new types of behavior, and many others.

When choosing the appropriate machine learning method, there is one significant factor that influences what kind of methods can be used for learning, which is the existence of reliable labels. Given the number of graphs, it is not possible for a human analyst to assign the labels manually. One must bear in mind that it is not a viable option to have one old labeled dataset since the threat landscape is very quickly evolving, and the system has to be scalable enough to allow regular retraining with the new dataset.

4.3.1 Labels

The process of obtaining labels for behavioral graphs starts by gathering all the artefacts present in the graph, i.e., file hashes, named objects, IPs, and URLs. These artefacts are then sent to a back-end service called *Tagger*. If this service is able to "tag" the provided artefact with the label, this label is then expanded on the whole graph. There is also a possibility that Tagger will not be able to assign any label, and the artefact remains unlabeled.

The Tagger response contains several pieces of information. The first one is **confidence**, which signalizes how much Tagger trusts the assigned label. Tagger internally queries various other systems where each of them has different reliability. If the confidence is below a reasonable threshold, which could be somewhere between 60-90%, the label is not taken into account. The Tagger also returns **severity** label that indicates whether the artefact is considered to be *clean* or *malware*. Besides those two, severity can also be set to PUP (Potentially Unwanted Application) but for the sake of simplicity, this value is not taken into account and is equal to no label. Lastly, there are **malware type** and **malware strain/malware family** labels that classify more precisely the artefact with malware severity according to a classical malware taxonomy. Malware can be assigned one of many malware types based on its functionality and objectives. Examples of more known malware types are *ransomware*, *spyware*, *backdoor*, *RAT*, *adware*, or *bot*. Malware strain is a more granular label than malware type, and it denotes a group of malware samples that have a common code base. Malware strain label usually groups malware samples spreading in a particular malicious campaign or samples produced by one actor. However, malware strain labeling is often inconsistent, as the malicious code is shared between multiple entities, often with parts of code reused or modified. Therefore, it might not be clear what can still be considered one family and what is already a different one. Moreover, malware strain naming lacks any standardization across the security industry, leading to the existence of multiple different names for one malware family and vice versa [28]. Despite that, there exists a quite comprehensive collection created by Daniel Plohmann and Steffen Enders - *Malpedia* [29]. Malware type and malware strain labels are left empty for clean artefacts, i.e., clean samples are not divided into any classes.

The severity labels from artefacts are expanded to behavioral graphs using the following rules. A graph is considered clean if and only if all the artefacts were labeled as *clean*. A graph is considered malware if and only if there is at least one artefact that is labeled as *malware*. Otherwise, a graph is labeled as *grey*, which is equivalent to no label. The empirical observations show that half of the graphs are usually labeled as clean for one day of data, while only less than 20% of graphs end up labeled as malware graphs. The reason causing this ratio is that the behavioral shield does the snapshot of the whole OS when some suspicious action happens, which is later split into several subgraphs. In an ideal case, only one of the subgraphs captures the malicious behavior, while the other captures regular OS processes that should end up classified as clean. Malware type and malware strain labels are expanded to *malware* graphs, but only in the case when there is no conflict between the artefacts labels. When two artefacts in one subgraph contain two distinct strain or malware type labels, the values are left empty. The number of graphs, which have the strain label set, is only in order of thousands (6-10K) for one day of data, contributing to only 0.01% of the dataset. The cause for such a low number of labeled graphs is that the behavioral shield producing these graphs can be seen as the AV's last layer of malware protection. When the AV back-end systems can identify the malware strain, there is a high chance that there already exists a static detection for that particular strain, which causes the threat neutralization before its execution.

A practical example of graph labeling can be seen in Figure 4.1. Two files and one IP address were extracted from the graph and labeled by Tagger:

- `cmd.exe` executable is tagged as clean because it is a well-known system executable.
- `623868e8566e1_mon1227f76e62.exe` executable is tagged as malware. Tagger identified it as a variant of *Redline stealer* strain belonging to *password stealer* malware type [30].
- IP address `x.x.x.x` is not present in the tagger database, and thus, no label is set.

Finally, the whole graph is labeled as: severity - *malware*, malware type - *password stealer*, and strain - *Redline stealer*. This example points out one of the issues with the behavioral graphs that have a strain label set. *Redline stealer* strain has various functionalities [30], but the presented graph shows only a single connection. This is because the behavioral shield probably decided that there is enough evidence that the running process is malicious (e.g., the process created malicious mutex) and should be immediately killed. As a result, the strain graph does not contain any interesting strain related activity.

4.4 Preceding work

This thesis builds on the preceding work of Adam Varga presented in his master thesis, *Identification and characterization of malicious behavior in behavioral graphs* [31], where the author proposed a GCN system capable of processing the behavioral graphs on the input and producing classification into selected malware strains. The author chose a supervised approach where the GCN was built to perform binary classification that distinguished whether the behavioral graph belongs to a particular strain or not. His work was further internally expanded into a classifier that was trained to distinguish between graphs with malware vs. clean severity (see Figure 4.2).

The up-to-date supervised model learns to recognize malware graphs from clean graphs based only on the severity label. The malware type and malware strain labels are not taken into account during the training process. Internally, the GCN produces a **supervised embedding** of the graph as an intermediate step. If the training is successful, meaning the metrics such as accuracy are high enough, one could assume that the GCN managed to filter out graph features relevant for distinguishing malicious and clean behavior. Since the malware type and strain labels were not considered during the training phase, they can be used as a sanity check to verify that the embedded graph space behaves "reasonably". The assumption is that the GCN whole-graph embedding should allow the distinction of different malware strains or types. That could be experimentally verified by clustering the graph embedded representations and analyzing the resulting clusters from the perspective of malware strain or type separation.

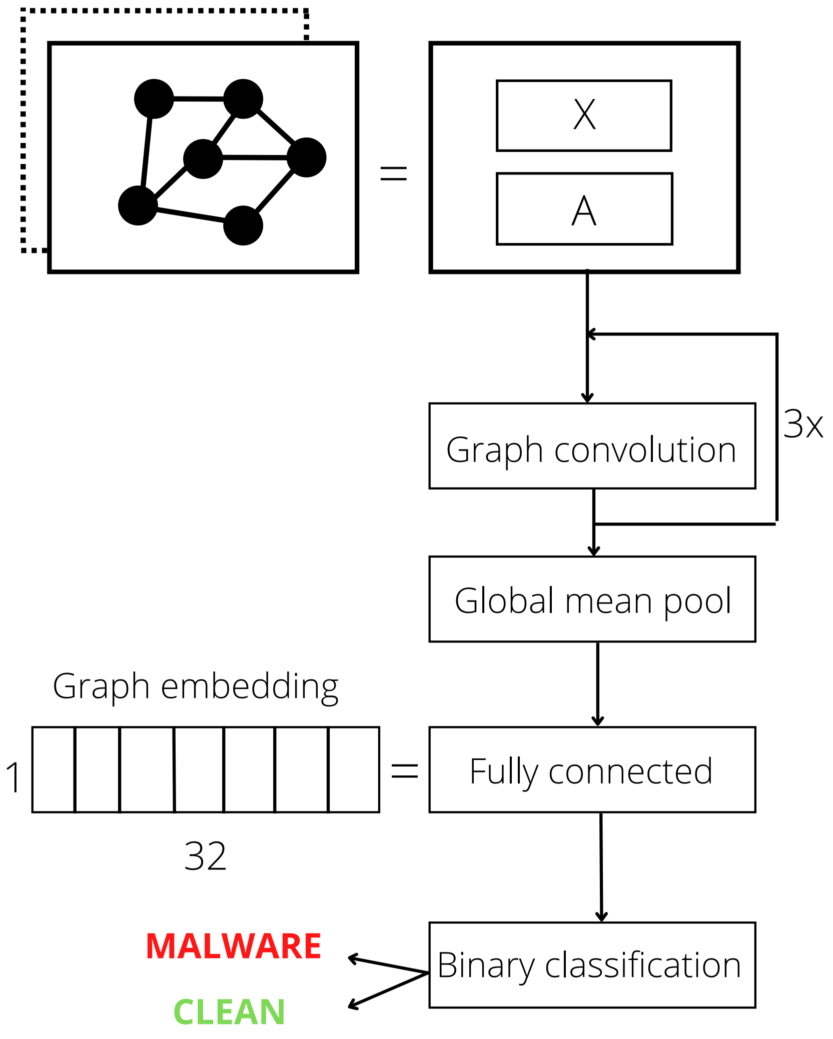


Figure 4.2 The current state of the supervised binary classifier. Behavioral graphs, represented via $\boldsymbol{X}, \boldsymbol{A}$ matrices, are processed by a convolutional layer and pooled to 32-dimensional representation, i.e., whole-graph embedding. The GCN is trained on the labeled part of the dataset to perform a binary classification into malware/clean classes.

In addition to the supervised model, two more approaches were taken when training a GCN to learn a graph embedding. The second approach was also supervised and based only on the severity label. In this case, the graph embedding was learned in a way that isomorphic (similar) graphs were projected into nearby regions while distinct graphs were projected further away from each other. This approach is still trying to separate the malware and clean graphs. To distinguish two supervised approaches, this embedding will be denoted as **matching embedding**.

The third type of embedding was trained purely in an unsupervised manner using a generative VAE model (see Section 2.5.2). Therefore, it will be referred to as **unsupervised embedding**. During the training process, the encoder part is trained to generate such low-dimensional representations of graph $\boldsymbol{z}_G$ that the decoder is able to reconstruct the input graph back. This bottleneck representation $\boldsymbol{z}_G$ can be used as a whole-graph embedding. The advantage of this approach is that the unlabeled graphs can be used as inputs during the training process. Figure 4.3 shows the diagram of the VAE model.

The specific parameters of the neural networks as well as description of their training process are confidential information of Avast company. All produced whole-graph embeddings have 32 dimensions. The number of dimensions was chosen based on intuition as a compromise between the size and expressibility, but no rigorous experiments were performed that would prove correctness of this choice.

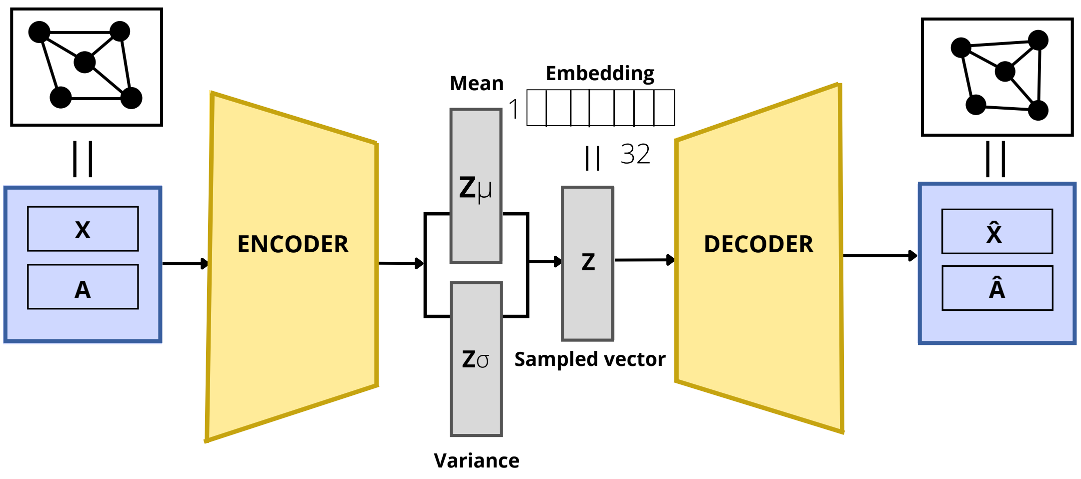


Figure 4.3 Diagram of the generative VAE model. The bottleneck representation can be used as a whole-graph embedding.

Chapter 5

Dataset analysis

The provided dataset consists of behavioral graphs captured by Avast behavioral shield during 48 hours. The first section contains an overview of the whole dataset and explains which part of the dataset is selected for further analysis. Afterwards, the top 5 malware strains, whose graphs were the most prevalent ones in the dataset, are described. The chapter ends with a discussion of behavioral graph labeling issues.

5.1 Data cleanup

The dataset contains 1 280 000 behavioral graphs that were captured during two days (from 1st to 3rd April 2022) by Avast behavioral shield. The graphs were collected from approx. 270 000 unique machines. There are 18% of malware graphs, 36% of grey (unlabeled) graphs, and 56% are clean graphs. Total number of graphs with malware type set is 9 540. Out of those, 8 836 graphs have also malware strain label set.

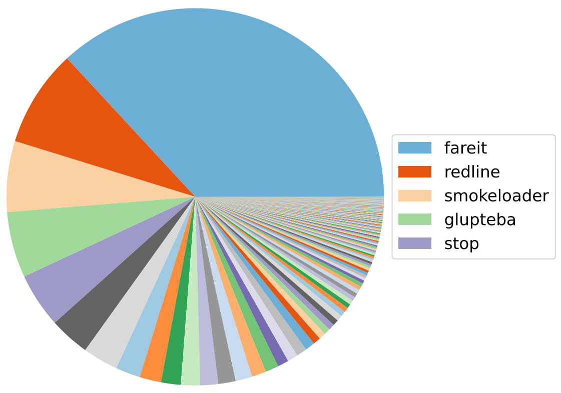


Figure 5.1 Pie chart showing the prevalence ratio of malware strain in the provided dataset. The legend mentions only the five most prevalent strains.

There are 171 unique strains present in the dataset overall. Most of them have a very low prevalence, i.e., below ten graphs. Figure 5.1 shows the prevalence ratio of all strains. It is obvious that the labels are unevenly distributed. This is not unexpected, given that the dataset consists of live data from users' endpoints from all around the world. The high prevalence of a particular malware strain might be caused by an ongoing live campaign, where the strain samples are distributed, leading to a higher number of infected users and thus a higher number of behavioral graphs. The other reason for unbalance might be the bias of what the behavioral shield sees. Some strains might be better covered with static detections, so they are not that often executed. For the purpose of clustering analysis, the low prevalent strains will not be taken into account, and only the top five prevalent strains will be selected. The label disbalance will be solved by sampling the remaining strains onto the prevalence of the last one in this selection. The final number of graphs is around 270 graphs per strain, which gives 1 350 graphs in total.

The significant reduction of the dataset down to 1 350 specifically selected graphs is made for the following reasons. Firstly, the selected graphs seem to be suitable for starting exploratory data analysis from the domain knowledge perspective. This type of analysis also requires manual evaluation, which is doable with that number of graphs. Secondly, it is very challenging to pick the correct parameters for the clustering algorithms due to the nature of the data, i.e., data are noisy and have unbalanced labels. The data cleanup can help deduce the proper parameters. For example, in the case of k-means, the number of clusters can be estimated as a value around $n \cdot k$, where n is the number of selected strains and $k \in \mathbb{N}$ is a small constant. The final sampling of selected strains was performed to allow using extrinsic metrics for clustering evaluation. In the case of unbalanced labels, the Purity and Rand Index metrics might be skewed due to high prevalent labels.

On the other hand, given that all three graph embeddings are 32-dimensional, this low number of samples might worsen the high-dimensionality issues that clustering algorithms have. Although, some preliminary clustering attempts with more samples did not show much difference between the structure of created clusters. Furthermore, the conclusions that are drawn on this specific labeled part of the dataset might be somehow biased by the selection of graphs. Therefore, the conclusions that will be presented should be verified later in the other parts of the dataset.

5.2 Description of prevalent malware strains

The following subsections contain a short description of the top 5 prevalent malware strains. Examples of behavioral graphs for each strain can be found in the `strain_analysis.ipynb` jupyter notebook together with a more specific description of malicious behavior. Given the size of behavioral graphs, they cannot be presented here because they would not be readable.

5.2.1 Fareit

Fareit, also known as *Pony Stealer*, is categorized as a *password stealer* malware type. Fareit spreads through drive-by downloads, phishing emails, fake updates or free software downloads. After infecting the computer, Fareit collects information about the system and attempts to steal credentials from more than 100 programs, including browsers, mail clients and various crypto-wallets. The information is exfiltrated onto the command and control server controlled by the adversary. Besides stealing capabilities, Fareit can also serve as the botnet controller that spreads additional malware [32].

Fareit samples usually drop their copies with names similar to Windows essential processes, e.g., `explorer.exe`, `svchost.exe`, into Windows resource directory. Afterwards, the malware schedules system tasks that start these binaries regularly. It also adds these binaries into the windows registry key `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce` that ensures the samples are run after every system boot.

5.2.2 Redline

RedLine Stealer is categorized as a *password stealer* that is capable of harvesting system inventory information, stealing saved credentials, autocomplete data and credit card information from browsers and stealing credentials for crypto-wallets. Besides that, this malware has the ability to upload and download files, execute commands, and periodically send back information about the infected computer. Redline Stealer is mainly distributed through phishing emails or malicious software disguised as installation files such as Telegram, Discord, and cracked software. Data exfiltration is performed via a TCP connection [33].

5.2.3 Smokeloader

The SmokeLoader strain can be categorized as a *dropper* malware type. Dropper malware is used to install other malware components once it compromises the system. Smokeloader frequently tries to hide its communication by generating requests to legitimate sites, and the actual download returns a 404 error but still contains data in the response body. The downloads are dropped into the `%AppData%` folder. The dropped binaries are mapped into a suspended instance of legitimate Windows process `explorer.exe`. Smokeloader instances create dynamic mutexes, which usually consist of some combination of computer name and Windows drive serial number, to ensure there is only one instance of the dropper running on the machine [34].

5.2.4 Glupteba

Glupteba malware, first seen in the year 2014, is a multi-functional malware written in Go language that is categorized as a *dropper* malware type but posses also other functionalities, like rootkit capabilities. Glupteba is able to spread using EternalBlue exploits. Once it has breached a system, the malware looks for SMB vulnerabilities and tries to exploit them to move laterally within the LAN. There are various versions of Glupteba, and each of them can behave in a slightly different way, so the following description is about particular samples present in the submitted behavioral graphs.

Glupteba copies itself to `C:\Windows\rss\`, as `csrss.exe` and `windefender.exe` binaries, and it creates persistence by changing the Windows registry autorun values. Glupteba modifies the registry key `HKLM\SOFTWARE\Microsoft\Windows Defender\Exclusions\Paths` to add exclusions to the paths of the dropped binaries and evade detection on Windows Defender. The registry `HKCU\Software\Classes\mscfile\shell\open\command` with default key value is created by the malware in order to abuse `CompMgmtLauncher.exe` and bypass Windows User Account Control. Consequently, an unchallenged execution or download of the further payload is enabled [35].

5.2.5 Stop

Stop malware, also known as DJVU, is a *ransomware* malware type. This ransomware uses a public key downloaded from the command and control server to encrypt data on the victim's machine using the *Salsa20* encryption algorithm. Upon the execution, the malware copies itself into the `%AppData%\Local\[SYSTEM_GUID]` directory and tries to execute itself with the escalated privileges. The dropped binary is run with `--Admin IsNotAutoStart IsNotTask` parameters, but it can also be executed in other modes like `--Task` and `--Service`.

Regarding the communication, Stop ransomware attempts to make a few network connections for purposes such as geo-checking, key retrieval, and machine infection with additional malware. Very often, *Vidar Stealer* malware strain, which is malware with password and credentials stealing capabilities, is downloaded after the initial infection. After completing the encryption process, the executed binary calls the adversary server with the unique ID based on the victims' MAC address. Stop ransomware also generates a scheduled task called the `Time Trigger Task` that regularly encrypts newly added files. As a final step, the ransomware drops a ransom note `_readme.txt` with instructions on how to pay the ransom into every encrypted directory [36].

5.3 Issues with strain labeling

Before the application of any clustering algorithms on the embedded graph representations, manual analysis of a few behavioral graphs was performed for each strain. The behavioral graphs selected for analysis were not picked randomly, but they were sorted per strain by the mean value of Jaccard distance between the particular graph and all the other graphs belonging to the same strain. Then, the most distant indexes from this sorted array were selected. The reasoning behind this selection was to pick the graphs with the lowest resemblance to each other and show how diverse are the graphs that belong to one strain. The diversity of strain graph structures can be seen in Figure 5.3

One of the first observations that can be made is that the graphs with the same label can have a totally different structure. That is not a big issue since malware strains can have various functionalities, and thus, the graph can capture distinct behaviors. This endorses the intuition that the number of clusters will not be equal to the number of selected strains, but it will probably be multiplied by some small number.

Another quick observation can be that the behavioral graphs belonging to a specific strain have some notable common features. For example, Glupteba behavioral graphs seem to be very big and fuzzy, or the SmokeLoader graphs contain one noticeable circle in the graph. Unfortunately, these differences cannot be directly attributed to a specific strain behavior and are more a by-product of present system noise. So even if those differences allow separating the strains into different clusters and give statistically pleasant results, this would not be interpreted as a correct solution from a domain knowledge standpoint.

After closer examination, it is evident that the graphs do not capture significant strain behavior that would allow distinguishing the individual strains. The reasons for that are AV protection mechanisms that tend to neutralize known threads. Figure 4.1 shows one such example, where spawned malicious binary has only one connection node and nothing else. Figure 5.2 shows two histograms with the number of nodes distribution. The left histogram shows the distribution for all malware graphs, while the right histogram shows node counts only for malware graphs with the strain label. Most of the labeled graphs have only around seven nodes.

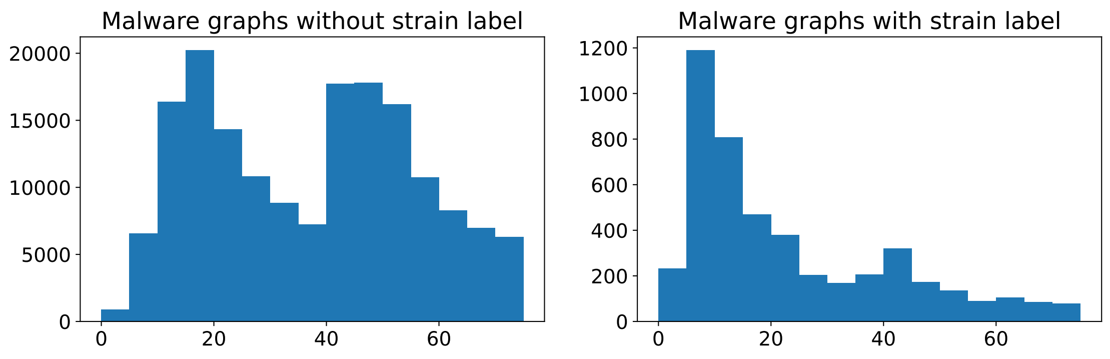


Figure 5.2 Histograms that depict the distribution of number of nodes for unlabeled vs. labeled malware graphs.

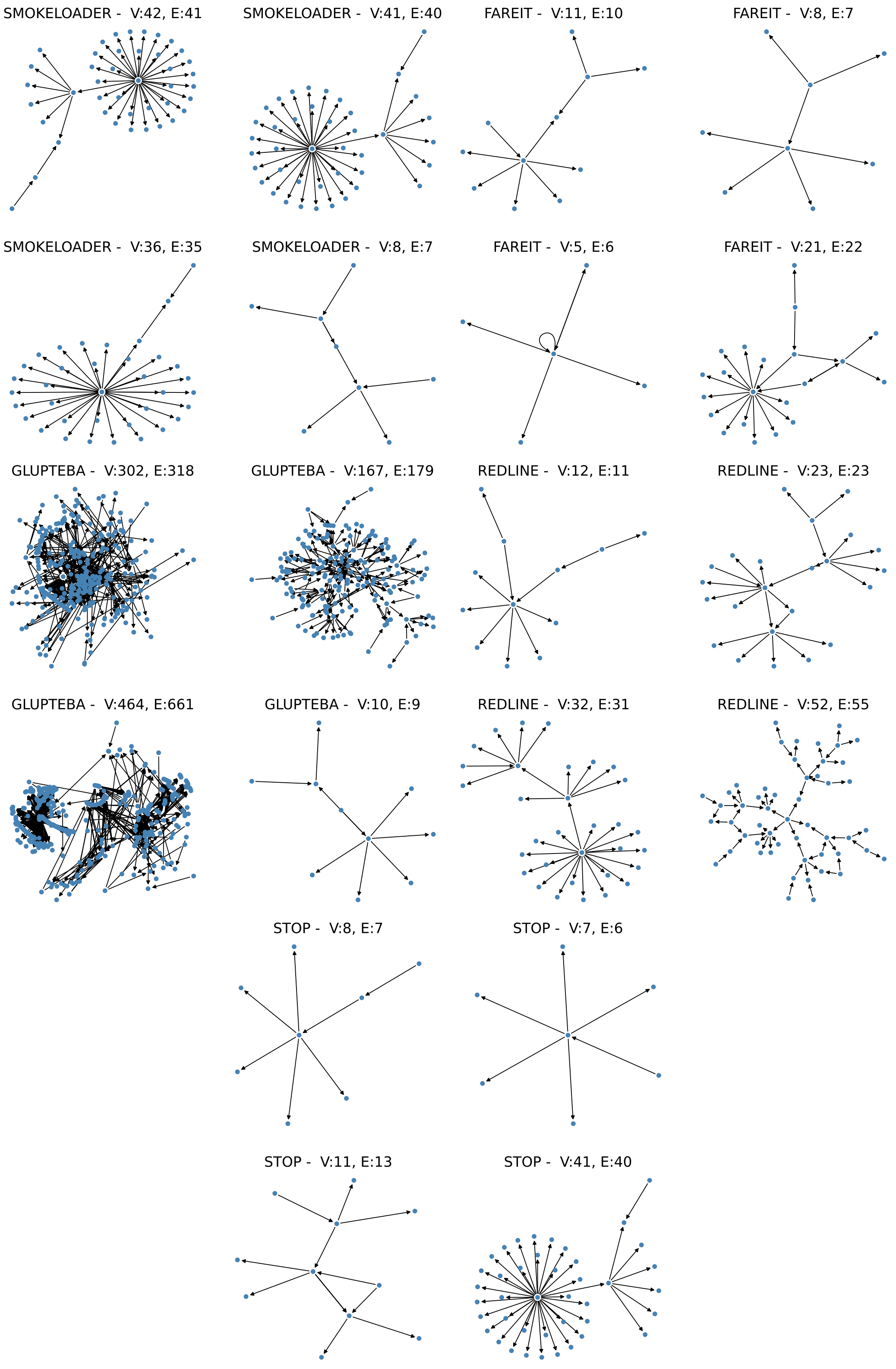


Figure 5.3 The overview of behavioral graph skeletons per strain. The graphs were selected in a way that the most distinct graphs from each strain should be displayed.

Besides that, the behavioral graphs capture events within a short time window after some suspicious action triggers the behavioral shield. However, malware is often executed in multiple stages, and there can be significant time breaks (e.g., deliberate sleeps, network communication) between them. The behavioral graphs usually contain only actions that happened around the time of the behavioral shield trigger, so the whole chain of actions is never present in the graphs. This partial information might be sufficient for the identification of malicious activity, but the identification of strain usually requires seeing "the whole picture". Also, modern malware is usually very modular, and one attack chain is composed of multiple different strains, where each strain is responsible for a specific activity, e.g., *Stop* ransomware encrypts the disk but also downloads *Vidar Stealer* that steals the user credentials. This might cause additional confusion with labeling since the actions of multiple strains present in the graph can overlap.

Furthermore, the behavioral graphs contain noise from the operating system, which sometimes contributes a significant amount of nodes to the graph. In Figure 5.3, three out of four SmokeLoader graphs have this notable circle, whose centre is actually Windows system process `svchost.exe`, and the adjacent nodes are the system mutexes or named objects that were modified as a consequence of regular system activities. The relevant malicious part of those graphs is the `svchost.exe` spawning malicious binary (i.e., the furthest adjacent node), while the rest of the graph is just system noise.

Another quite specific but prevalent issue in the dataset is mislabeling caused by other security products present on the running system, which is the case of Glupteba graphs seen in Figure 5.3. The analysis showed that those huge Glupteba graphs are capturing the behavior of another security software that creates the same mutexes as the Glupteba malware. This protection technique is known as malware vaccination. The Glupteba uses the specific mutex name to check whether its instance is already running on the system and if yes, the malware is not started. Therefore, security solutions create the same mutex to prevent the malware from running, which causes the behavioral shield submits.

Based on the conducted analysis, the author believes that the behavioral malware graphs in their current form do not possess the information needed for distinguishing individual malware strains. On the other hand, what is usually present in the behavioral graphs and can be interesting from an analytical perspective, is the information about the infection vector, i.e. how was the malicious binary executed. For example, if the malicious process was spawned as a consequence of a scheduled task, the usual spawner of that process is the `svchost.exe`, which can be identified in the multiple skeleton graphs in Figure 5.3 as a centre of the notable circle of nodes. Such patterns can be seen across different strains, e.g., `SMOKELOADER - V:41, E:40` or `STOP - V:41, E:40`.

Chapter 6

Evaluation of clustering methods

This chapter outlines the process of selecting adequate clustering algorithm and whole-graph embedding for the purpose of generating malware signatures. At the beginning, the preliminary analysis of provided embeddings is done to ensure that the data are suitable for clustering. Then, three clustering methods will be consecutively applied. Each clustering algorithm requires different parameters that need to be properly selected. The clustering results of each method are evaluated via presented matrices and the best method is selected. Last section presents the process of selecting a graph from cluster that can be used as a base for malware signature.

The experiments presented in this chapter aim to apply the selected clustering algorithms, *k-means*, *DBSCAN* and *agglomerative clustering*, on three existing 32-dimensional whole-graph embeddings, i.e., *supervised*, *unsupervised* and *matching*, and evaluate the quality of constructed clusters. For the experimentation, dataset consisting of 1 350 behavioral graphs from 5 different strains is used. For the evaluation, the following metrics were selected:

- **Intrinsic metrics** - Silhouette Index (SI) and Davies-Bouldin Index (DBI).
- **Extrinsic metrics** - Purity Index (PI) and Rand Index (RI). The malware strain labels are assumed to be the "ground truth".
- **Biggest cluster size ratio** (BCR), **average cluster size** (AC), **median cluster size** (MC). These metrics were added due to the observation that the application of clustering methods leads to clusters of uneven sizes.
- **Jaccard Distance** d_J. This metric is calculated per graph as a mean of Jaccard distances of each graph compared to the rest of graphs in the same cluster. Then, *cluster* d_J is computed as a mean of all graph d_J in that particular cluster. Similarly, the mean value of all *cluster* d_J is taken to get the d_J value for the whole clustering.

When applying a clustering algorithm, the first step is to check the input data features and verify whether they need to be **normalized**. In this case, all three embeddings were created by a neural network, so the assumption is that the data features will be more or less in the same range. To check the range and the variance of features, feature values histograms are plotted in the `embeddings_analysis.ipynb`, together with the statistics like maximum, mean and standard deviation.

The range of the features does not significantly differ, so there is no need to apply any normalization. However, the statistics show that some of the features have a low span of values and contain mostly 0. The unsupervised embedding has half of the features low-variant, meaning that more than 60% of values are 0, while the matching and supervised embedding have two low-variant features. There are few theories why unsupervised embedding has so many “degenerated features”. The first theory is that the unsupervised neural network is overfitting on particular input characteristics of the data, such as the size of the graph. Another theory is that the degenerated features might represent some binary output features, so their variance is not that big since they only represent 0 or 1, which might also be caused by overfitting. It would be interesting to perform experiments with fewer dimensions in the unsupervised embedding to see if this problem vanishes or not.

For the sake of simplicity, the distance metric will be set to the Euclidean metric as a default. During the first iterations of experiments, there were some attempts to modify the metric. Both cosine and Manhattan distances were probed, but the results did not differ significantly, i.e., up to a few points clustered differently, the structure of clusters remained the same.

Each clustering algorithm has different input parameters that need to be selected based on the particular input data. In the following sections, the process of picking the suitable parameters for each clustering algorithm is presented.

6.1 K-means

K-means algorithm requires one parameter, which is the number of expected clusters K. The optimal value of K is usually determined by elbow method. The procedure for finding the value K is repeated separately for each embedding. All three embeddings are analysed in the jupyter notebook `kmeans_all_embeddings.ipynb`. Since the process is very similar, this section contains the detailed description of the process only for one selected embedding – *matching embedding*.

Given the fact that there are 5 malware strains, the expected number of clusters will start at this number. The first search for elbow can be more coarse, so the upper range is set to 100, and the step is set to 5. Plots of the monitored metrics can be seen in Figure 6.1.

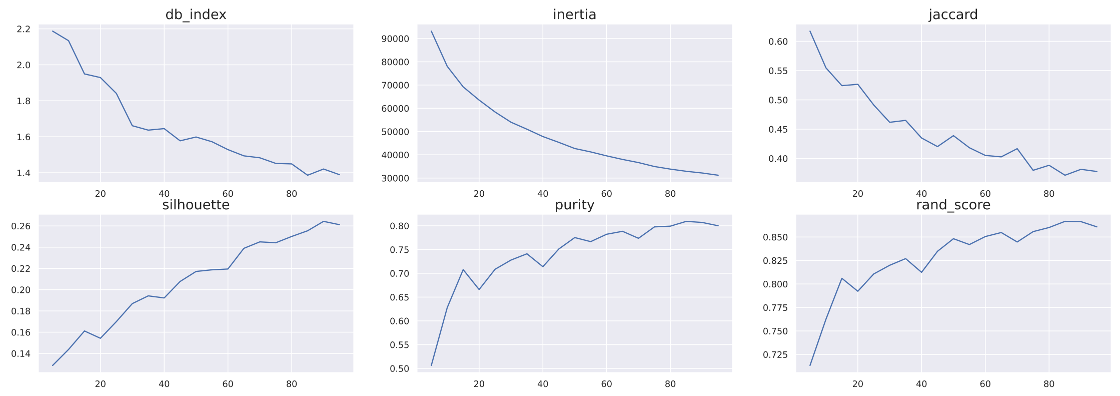


Figure 6.1 The plots of selected metrics for k-means clustering with matching embedding. The x-axis shows the K parameter values, while y-axis depicts the particular metric value.

Besides the aforementioned intrinsic and extrinsic metrics, the plots also depict k-means *inertia* value, the objective function optimized during the clustering process. There is no significant elbow in the plots of intrinsic metrics, inertia, or Jaccard distance. Although, the extrinsic metrics show the elbow somewhere below 20. So, in the next iteration, similar graphs are drawn but with a finer range starting at 10 and ending at 25 with step 1. The new graphs show that the most notable elbow is around 15. Therefore, $K = 15$ is set as the best parameter for the matching embedding. The same value is set for *supervised embedding*. With the *unsupervised embedding*, the optimal number of clusters is identified as $K = 20$.

6.2 DBSCAN

DBSCAN algorithm requires two input parameters that are set according to the density of the "least dense" cluster. Those parameters are $minPts$ and ε-distance. The usual process of choosing the parameters [20] is to set $minPts$ first. Sander et al. [37] suggest that $minPts$ should be set to $2 * dim$, where dim is the dimensionality of data, which is, in this case, $minPts = 64$. However, due to a small number of data points, the author chooses the maximum $minPts = 60$ but lower values are tested as well, i.e., $minPts = 10$ and $minPts = 3$.

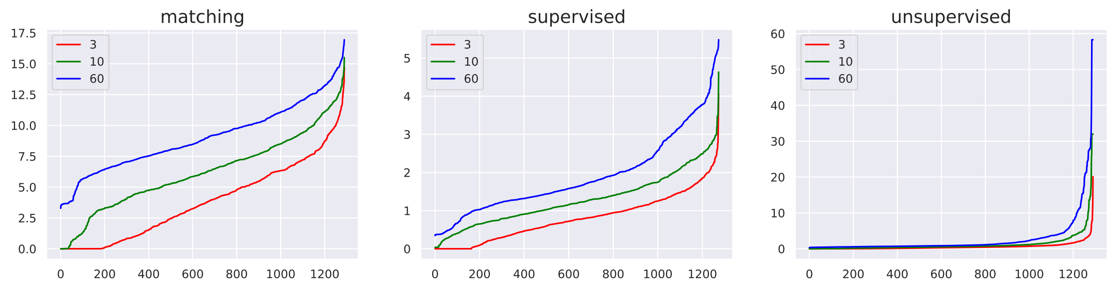


Figure 6.2 *Sorted k-distance plots* of all embeddings with various $minPts$ values. The x-axis denotes sorted data points, while y-axis shows ε distance to the k-th neighbor. If the parameter $minPts$ is set to k and ε is set to some arbitrary value on y-axis, all the data points that are below the ε value on the graph will be categorized by DBSCAN as the core objects.

To select the appropriate ε-distance range for specific $minPts$, Ester et al. [21] suggest defining a function k-*dist* that returns the distance to the k-th nearest neighbor for a given data point. This function can be then plotted as a *sorted k-distance* graph, where the data points are sorted in the ascending order according to their distance from k-th neighbor. Figure 6.2 shows a *sorted k-distance plot* for each embedding and three options of $minPts$ parameter.

The plot of *unsupervised embedding* shows that most of the data points have their k-th neighbor in a very small distance close to zero for any choice of $minPts$ parameter, which indicates that the unsupervised space lack any density structure. The *matching* and *supervised embedding* plots show the appropriate ranges of ε-distance. The author chooses $minPts = 10$ for the following experiments. The $minPts$ value also determines the minimal number of samples in the formed clusters, and the experiments showed that the $minPts = 60$ creates maximally up to 3 clusters, while $minPts = 3$ leads to forming clusters with only three samples, which produces a lot of noise.

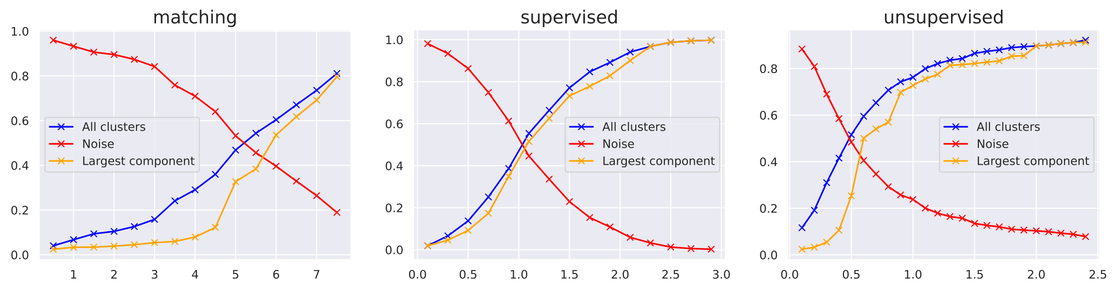


Figure 6.3 Statistics of DBSCAN clustering results for all three embeddings with $minPts = 10$. The x-axis indicates the value of ε-distance parameter, and y-axis displays the ratio of the dataset. The graphs depict how does the ratio of the noise compared to clustered graphs and the biggest cluster evolve with rising ε.

Plots in Figure 6.2 only show how many data points end up classified as the core objects. However, they do not give any information about the clustering structure, meaning how and whether the core objects are separated into distinct clusters or not. Figure 6.3 presents plots that show the resulting clustering structure with respect to changing ε-distance parameter. The plots indicate that most points end up as noise, or they are clustered into one big cluster. These results suggest that the DBSCAN clustering is not a suitable option for clustering any of the embeddings. Jupyter notebook `dbscan_all_embeddings.ipynb` contains all the performed regarding the DBSCAN clustering.

6.3 Agglomerative clustering

Agglomerative clustering allows defining either the number of clusters or the linkage distance threshold as input parameters. The first experimental attempts used the linkage distance as a terminating criterion, but for the purpose of comparison with k-means, the terminating criterion is set to the number of created clusters. The process is very similar to the k-means algorithm, where the appropriate number of clusters is selected based on the elbow technique. This section presents this approach only on the *matching embedding*. All agglomerative clustering experiments are shown in the jupyter notebook `agglomerative_all_embeddings.ipynb`.

A *complete linkage* is selected as a linkage distance criterion. No experiments were performed with regard to different linkage criterion. The author chose this criterion based on the intuition that it better separates denser data, which seems to be the case regarding the used behavioral graph embeddings. However, additional experiments would need to be performed to verify the correctness of the hypothesis.

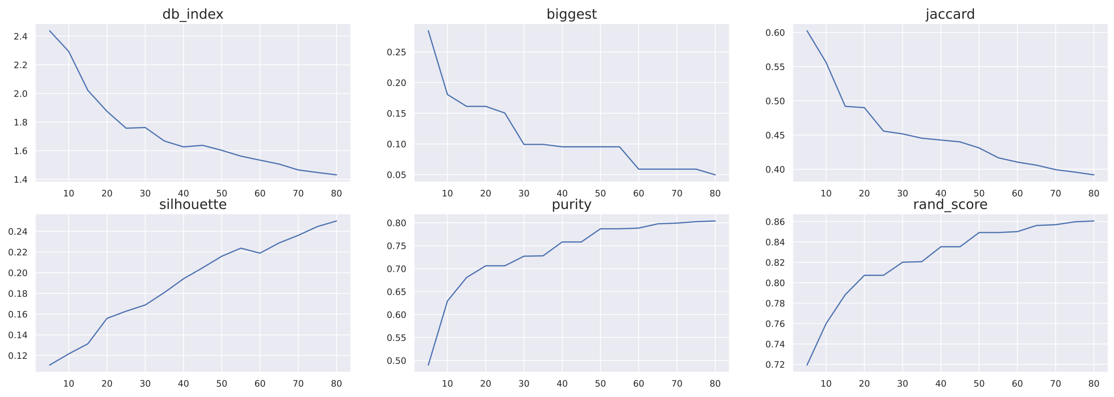


Figure 6.4 The plots of selected metrics for agglomerative clustering with matching embedding. The x-axis shows the number of clusters, while y-axis depicts the particular metric value. Instead of inertia, the biggest cluster ratio is depicted.

Similarly to k-means results, the elbow is only present on the extrinsic metrics graph (see Figure 6.4). The breaking points seem to be around 20, so for the next iteration, the range is set from 10 to 30 clusters with step 1. The zoomed graphs show that the number of graphs set to 20 is a reasonable termination criterion.

6.4 Comparison of algorithms and embeddings

The previous sections show the procedure of choosing the parameters for each clustering algorithm. These experiments were performed for all three clustering algorithms and embeddings to pick the best performing variant.

For the final comparison, the DBSCAN algorithm is not taken into account due to the low quality of the produced clusters. Therefore, only agglomerative clustering and k-means are compared together. Both of the algorithms accept the number of clusters as an input parameter, and in all six cases, the reasonable values seem to be around 15 and 20 clusters. Table 6.1 and Table 6.2 show the values of clustering metrics for $K = 15$ and $K = 20$.

	agglomerative clustering			k-means		
K = 15	matching	supervised	unsupervised	matching	supervised	unsupervised
SI	0.131293	0.182285	0.411038	0.161151	0.210476	**0.556896**
DBI	2.021368	1.529351	**0.704561**	1.949308	1.535314	0.723828
PI	0.680620	0.594976	0.412858	**0.707752**	0.614600	0.412084
RI	0.788309	0.733267	0.599300	**0.806067**	0.760897	0.577060
d_J	0.491947	0.492154	**0.429396**	0.524297	0.561550	0.441869
BCR	0.161240	0.185243	0.591789	0.136434	0.182104	0.620449
AC	86.000000	84.933333	86.066667	86.000000	84.933333	86.066667
MC	75.0	42.0	19.0	73.0	57.0	11.0

Table 6.1 Clustering results for the number of clusters $K = 15$.

	agglomerative clustering			k-means		
K = 20	matching	supervised	unsupervised	matching	supervised	unsupervised
SI	0.155865	0.186401	**0.417140**	0.154271	0.213520	0.257042
DBI	1.875045	1.568177	**0.761903**	1.929138	1.459807	0.884757
PI	**0.706202**	0.610675	0.418280	0.665891	0.621664	0.538342
RI	**0.807301**	0.749972	0.602048	0.792267	0.761344	0.715108
d_J	0.490013	0.483407	**0.414982**	0.526629	0.498898	0.458994
BCR	0.161240	0.185243	0.591789	0.100000	0.170330	0.333850
AC	64.500000	63.700000	64.550000	64.500000	63.700000	64.550000
MC	46.0	41.5	13.5	60.5	42.5	18.0

Table 6.2 Clustering results for the number of clusters $K = 20$.

The *matching embedding* achieves the lowest score in intrinsic metrics compared to the other two embeddings, indicating a worse cluster structure. On the other hand, the extrinsic metrics score was the highest, so the matching embedding is more successful at separating different strains. BCR, AC and MC metrics show that this embedding creates more evenly sized clusters. The *unsupervised embedding* has more than two times higher intrinsic scores. The noticeable difference is caused by the fact that most of the points end up in one cluster since BCR value is above 50 %. As a consequence, the extrinsic metrics are lower. Also, the differences in cluster sizes are significant, which is indicated by the small median cluster size. On the other hand, unsupervised embedding has the lowest Jaccard distance, which indicates higher similarity of graphs in one cluster. The *supervised embedding* results are very similar to the results of matching embedding. Although, the cluster sizes are more uneven.

The measured results show that there is not much difference in the quality of created clusters between the *agglomerative clustering* and *k-means*. The selection of the embedding has a much more significant impact. Although agglomerative clustering provides more helpful information about the clustering process, i.e., the process can be plotted in a dendrogram and allows the linkage distance to be used as a termination criterion.

To summarize the results, the clustering experiments and their evaluation lead to a conclusion that the majority of the graphs cannot be properly separated into distinct clusters. Several arguments support this statement. Firstly, DBSCAN algorithm tends to cluster all data into one giant component. This tendency to keep one big cluster is also seen with other clustering algorithms, and it is most noticeable with unsupervised embedding. Secondly, the low score of intrinsic metrics in the case of agglomerative clustering and k-means indicates a bad separation of the data from the perspective of those metrics. Thirdly, the size of created clusters is very uneven, and most of the formed clusters are small.

On the other hand, the manual analysis of the formed clusters confirmed that they consist of similarly looking graphs. It is proof that graphs with similar input features are put closer together in the embedded space, so the embeddings have some notion of similarity. The insufficient separation of graphs could be caused by the low quality of the dataset or the neural networks overfitting during the training process to particular input features.

6.5 Selecting graph for signature

For the generation of malware signatures, the embedding and clustering algorithm with the highest extrinsic metrics is selected. Matching embedding scored the highest in two cases: agglomerative clustering with $K = 20$ and k-means with $K = 15$. The experiment with a higher number of clusters is chosen, i.e., matching embedding with agglomerative clustering and $K = 20$. Figure 6.5 shows all the data points, which is 1 350 behavioral graphs, in a 2D colored graph where colors are assigned according to the labels of selected clustering. The 32-dimensional data points are reduced into 2-dimensional space using UMAP reduction method [38].

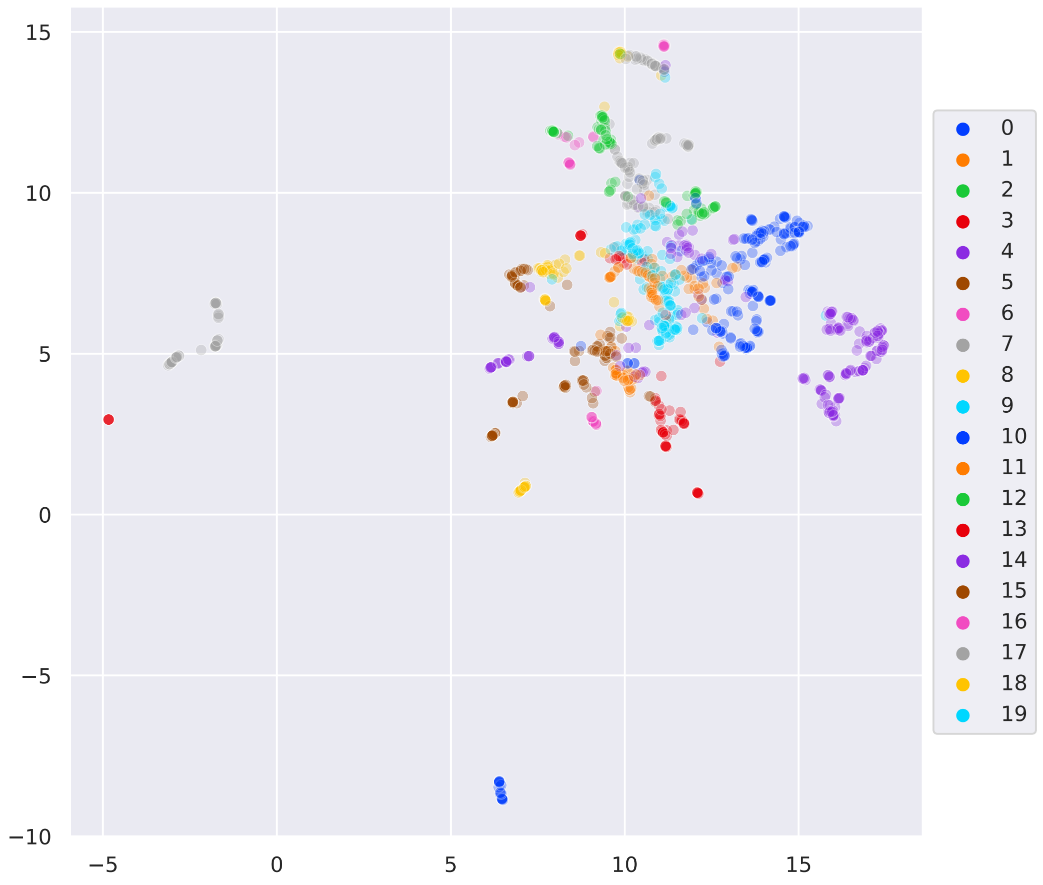


Figure 6.5 Clusters of the matching embedding with agglomerative clustering for $K = 20$. Colors are assigned according to the clustering labels.

Additional statistics are calculated for each cluster to get some notion about what the cluster represents. Those statistics are listed below:

- ID - cluster label that was assigned as a result of the clustering process,
- N - number of graphs in a cluster,
- SI - Silhouette Index per cluster,
- MS - majority strain label,
- MSR - majority strain label ratio or how many graphs in a cluster have majority label,
- ANC - average node count,
- d_J - Jaccard distance between the graphs in a cluster.

Table 6.3 shows the cluster statistics for the selected clustering method, but only clusters with $d_J < 0.5$ are depicted and are further considered for signature generation. That leads to 10 clusters in total. Also, the clusters in the table are sorted according to d_J column. The d_J threshold was selected based on manual analysis since graphs in clusters with $d_J \geq 0.5$ were too diverse and contained a lot of noise. Interestingly, the d_J values do not correlate with SI, suggesting that silhouette index is not a reliable indicator of graph similarity within the cluster. Another interesting observation is the significant differences between cluster sizes. The biggest cluster has 208 samples, while the smallest have only 15.

ID	N	SI	MS	MSR	ANC	d_J
10	**22**	**0.439616**	**stop**	**1.000000**	**9.181818**	**0.151160**
17	48	0.523142	stop	0.687500	7.937500	0.268396
12	39	0.136799	smokeloader	0.769231	42.538462	0.315350
13	**60**	**0.316575**	**stop**	**0.733333**	**9.283333**	**0.322935**
11	35	0.087149	glupteba	0.942857	21.514286	0.345859
16	15	0.342623	stop	0.733333	8.800000	0.389757
0	208	0.224803	smokeloader	0.649038	48.225962	0.396466
6	**26**	**0.092985**	**fareit**	**1.000000**	**17.153846**	**0.407847**
3	18	0.074842	redline	0.722222	26.333333	0.432771
18	16	0.431126	fareit	1.000000	24.500000	0.486314

Table 6.3 Cluster statistics for clusters with $d_J < 0.5$ sorted by the d_J. Bold rows mark the clusters, whose signature graphs are presented at the end of this section.

The most representative graph is selected for each cluster by choosing the behavioral graph with the lowest Jaccard distance. This graph is the most similar to all graphs in a cluster and is denoted as **signature graph**. The restriction of $d_J < 0.5$ ensures that the signature graph should be a good representative of all behavioral graphs assigned to same cluster. Signature graph can be seen as a basic building block of the malware signature for a given cluster.

Signature graph cannot be used as a malware signature straight away because the behavioral graph still contains many features that are irrelevant to the malicious behavior. Therefore, there has to be additional post-processing that will filter out only relevant features from the graph. The implementation of this filtering mechanism is considered to be out of the scope of this thesis, given the complexity of the task. Though, the idea of what this process would look like is presented here. At first, one could try to delete an arbitrary node from the signature graph and then embed and cluster the reduced graph again. If such graph does not end up in the same cluster, the node is considered relevant and will be part of the malware signature. Otherwise, the node is left out. This process is repeated for each signature graph node. Afterwards, an analogous reduction is applied for the graph edges and also for the features of every node. This way, only the features that cause the graph attribution to a particular cluster are left. The result of this post-processing then can be used as a behavioral malware signature.

The manual analysis of the cluster signature graphs was performed to verify whether they contain some relevant information for a particular strain or malicious behavior. Analysis of clusters from Table 6.3 can be found in the jupyter notebook `signature_analysis.ipynb`. In the text of the thesis, only the three selected signature graphs are shown in detail.

Figure 6.6 depicts a signature graph of a cluster with ID = 10, where all behavioral graphs are labeled as *Stop* malware strain. The signature graph contains a process with four-letter random name that is instantiated from the executable located in `%AppData%\Local\Temp` folder. The process is registered as a Windows service. The mentioned characteristics signify malicious behavior but are not enough to identify the strain. It is possible that signature consisting of these characteristics could cause false positives, so it should be tested before release. All the other graphs in the cluster have the same interesting characteristics as the signature graph.

Besides the relevant characteristics, the signature graph also contains noise. The named object nodes, e.g. mutex `windows_shell_global_counters`, are a by-product of system actions and are not relevant to malicious behavior. They are also not present in all the graphs in the cluster with ID = 10.

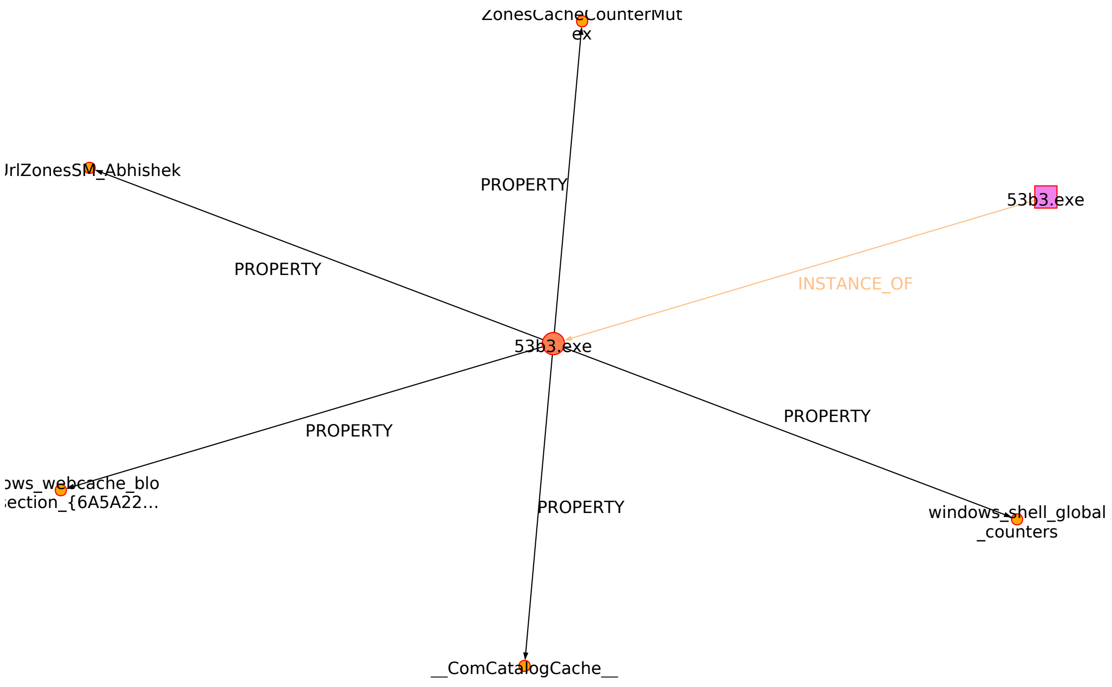


Figure 6.6 Signature graph of a cluster with ID = 10.

Figure 6.7 shows a signature graph of a cluster with ID = 13. The signature graph has one process that is launched as a scheduled task called `Time Trigger Task` and was executed with the command-line argument `--Task`. Both characteristics are indicators of Stop Ransomware behavior (see Section 5.2.5) and could be used for reliable detection. Besides *Stop* strain labels, the cluster also contains a significant amount of *SmokeLoader* strain labels. That is due to the fact that *SmokeLoader* dropper sometimes drops the *Stop* ransomware in the next stage. Further, both Figure 6.7 and Figure 6.6 show the signature graphs for *Stop* malware strain, but Figure 6.7 has much more context than Figure 6.6. That is a good example of why are the strains represented by multiple graphs. It also shows that the behavioral graphs always capture very partial information regarding the strain behavior.

Figure 6.8 shows a signature graph of a cluster with ID = 6, which contains behavioral graph with *Fareit* (see Section 5.2.1) malware strain labels only. The signature graph contains two processes. The first one, `RunOnce`, is a system process responsible for running tasks, which were registered at Windows autorun registry, when the OS boots up. This process spawns `svchost.exe` with unconventional executable path `C:\WINDOWS\RESOURCES\`, which indicates that it is probably malicious binary masquerading as a regular Windows process.

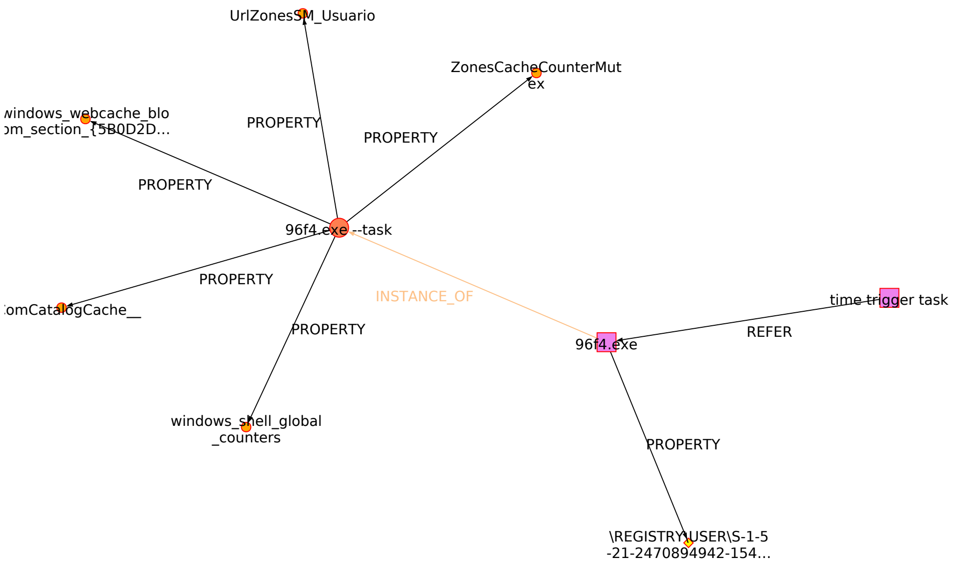


Figure 6.7 Signature graph of a cluster with ID = 13.

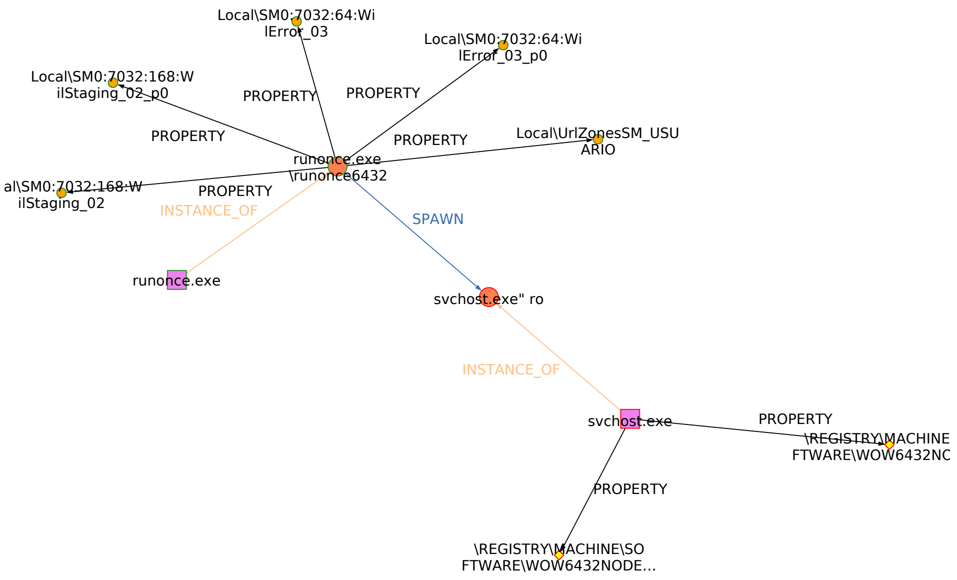


Figure 6.8 Signature graph of a cluster with ID = 6.

Conclusion

The goal of this thesis was to perform an exploratory analysis of the behavioral graph embeddings that were produced as a part of internal research of the Avast company. The embeddings were created using novel techniques for training deep neural networks explicitly designed to process data in the form of graphs. Three distinct approaches were taken towards the model training: supervised learning on clean vs. malware labels, unsupervised learning using variational autoencoder model, and supervised learning matching the isomorphism metric. The analysis was performed using three clustering algorithms: k-means, DBSCAN, and agglomerative clustering. The main aim of the analysis was to verify whether the trained whole-graph embeddings allow distinguishing behavioral graphs of different malware strains. The positive result would suggest that the features extracted from behavioral graphs are relevant for differentiating the malicious and clean behavior. Furthermore, in case of good strain separation in resulting clustering, the clusters would be used to create behavioral signatures for a given malware strain.

As the first step in the analysis process, the behavioral graphs of selected malware strains were examined, and the results were discussed in Chapter 5. The examination pointed out that behavioral graphs do not capture the behavior of selected malware strains sufficiently, primarily due to the fact that the known threats are neutralized, or the graphs capture only partial strain behavior. The behavioral graphs also contain a lot of system noise, which in certain cases contributes to a significant part of the graph, and thus overwhelms the relevant information. The issue with inconsistent strain labeling and mixing of strain labels was discussed as well.

The clustering evaluation in Chapter 6 showed that DBSCAN is not a suitable method for clustering the embedded behavioral graphs since all the graphs ended up either as noise or clustered as one big component. Agglomerative clustering and k-means method performed similarly well. The selection of particular embedding had a more significant effect on the evaluation metrics than choosing k-means or agglomerative clustering. The best performing embedding concerning extrinsic metrics was matching embedding. In combination with agglomerative clustering and parameter $K = 20$, denoting the number of formed clusters, the evaluation metrics Purity Index reached 70%, while Rand Index reached 80%.

The last step in the analysis process was dedicated to discussion regarding the possibility of creating behavioral signatures from the created clusters. Section 6.5 described how a behavioral signature can be generated based on the clustering results. The successful signature generation was out of the scope of this work, though the foundations of the signature generation process had been laid.

In future work, more focus should be put on improving the quality of the input data. Some methods could be developed that would reduce the amount of noise present in the behavioral graphs. Also, the analysis from the perspective of malware domain knowledge suggests that the behavioral graphs are not suitable for distinguishing different malware strains. Instead, the behavioral graphs contain information that allows distinguishing different infection vectors, so future research should take that into account. Furthermore, the suggestions presented in Section 6.5 regarding the graph signature generation could be empirically tested.

Bibliography

1. VELIČKOVIĆ, Petar. *Theoretical Foundations of Graph Neural Networks* [CST Wednesday Seminar]. 2021 [visited on 2022-03-30]. Available from: `https://petar-v.com/talks/GNN-Wednesday.pdf`.
2. A gentle introduction to deep learning for graphs. *Neural Networks*. 2020, vol. 129, pp. 203–221. ISSN 0893-6080. Available from DOI: `https://doi.org/10.1016/j.neunet.2020.06.006`.
3. HAMILTON, William L. *Graph Representation Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool, 2020.
4. DEO, Narsingh. *Graph Theory with Applications to Engineering & Computer Science*. Dover edition. Prentice-Hall, 1974. ISBN 9780486807935.
5. COUPETTE, Corinna; VREEKEN, Jilles. Graph Similarity Description: How Are These Graphs Similar? In: 2021, pp. 185–195. Available from DOI: `10.1145/3447548.3467257`.
6. TONG, Flawnson. *Everything you need to know about Graph Theory for Deep Learning* [online]. 2019 [visited on 2022-03-20]. Available from: `https://towardsdatascience.com/graph-theory-and-deep-learning-know-hows-6556b0e9891b`.
7. DAIGAVANE, Ameya; RAVINDRAN, Balaraman; AGGARWAL, Gaurav. *Understanding Convolutions on Graphs* [online]. 2019 [visited on 2022-03-22]. Available from DOI: `10.23915/distill.00032`.
8. CAI, HongYun; ZHENG, Vincent W.; CHANG, Kevin Chen-Chuan. A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications. *IEEE Transactions on Knowledge and Data Engineering*. 2018, vol. 30, no. 9, pp. 1616–1637. Available from DOI: `10.1109/TKDE.2018.2807452`.
9. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.
10. KARITA, Shigeki et al. A comparative study on transformer vs RNN in speech applications. *IEEE Automatic Speech Recognition and Understanding Workshop 2019*. [N.d.]. Available from arXiv: `1909.06317`.
11. BRONSTEIN, Michael M. et al. Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*. 2017, vol. 34, no. 4, pp. 18–42. Available from DOI: `10.1109/MSP.2017.2693418`.
12. TONG, Flawnson. Graph Convolutional Networks for Geometric Deep Learning. *Towards Data Science* [online]. 2019 [visited on 2022-03-29]. Available from: `https://towardsdatascience.com/graph-convolutional-networks-for-geometric-deep-learning-1faf17dee008%20%5Cnewline`.

13. [Online] [visited on 2022-04-20]. Available from: https://i.stack.imgur.com/YDusp.png.
14. HAN, Jiawei; KAMBER, Micheline; PEI, Jian. *Data Mining: Concepts and Techniques*. Elsevier, 2012. Available from DOI: https://doi.org/10.1016/C2009-0-61819-5.
15. AGGARWAL, Charu C.; HINNEBURG, Alexander; KEIM, Daniel A. On the Surprising Behavior of Distance Metrics in High Dimensional Space. *Towards Data Science*. 2001, vol. 1973. Available from DOI: https://doi.org/10.1007/3-540-44503-X_27.
16. ESTIVILL-CASTRO, Vladimir. Why so many clustering algorithms: a position paper. *ACM SIGKDD Explorations Newsletter*. 2002, vol. 4. Available from DOI: https://doi.org/10.1145/568574.568575.
17. ALOISE, Daniel et al. NP-hardness of Euclidean sum-of-squares clustering. *Mach Learn*. 2009, vol. 75. Available from DOI: https://doi.org/10.1007/s10994-009-5103-0.
18. *A demo of K-Means clustering on the handwritten digits data* [online]. 2022 [visited on 2022-04-08]. Available from: https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_digits.html.
19. *Clustering* [online]. 2022 [visited on 2022-04-09]. Available from: https://scikit-learn.org/stable/modules/clustering.html.
20. SCHUBERT, Erich et al. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. 2017. Available also from: https://doi.org/10.1145/3068335.
21. ESTER, Martin et al. A density-based algorithm for discovering clusters in large spatial databases with noise. 1996. Available also from: https://www.osti.gov/biblio/421283.
22. VINH, Nguyen Xuan; EPPS, Julien; BAILEY, James. Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance. *J. Mach. Learn. Res.* 2010, vol. 11, pp. 2837–2854. ISSN 1532-4435.
23. VLCEK, Ondrej. *Behavior Shield: our newest behavioral analysis technology* [online]. 2017 [visited on 2022-04-12]. Available from: https://blog.avast.com/behavior-shield-our-newest-behavioral-analysis-technology.
24. CHIN, Tommy et al. A Machine Learning Framework for Studying Domain Generation Algorithm (DGA)-Based Malware. In: *Security and Privacy in Communication Networks*. Cham: Springer International Publishing, 2018, pp. 433–448. ISBN 978-3-030-01701-9. Available from DOI: https://doi.org/10.1007/978-3-030-01701-9_24.
25. K., Sudhakar; KUMAR, Sushil. An emerging threat Fileless malware: a survey and research challenges. 2019, vol. 3, p. 1. Available from DOI: 10.1186/s42400-019-0043-x.
26. BIANCO, David. *The Pyramid of Pain* [online]. 2014 [visited on 2022-04-13]. Available from: https://rvasec.com/slides/2014/Bianco_Pyramid%5C%20of%5C%20Pain.pdf.
27. LAB, Avast AI Research. *Catching malware red-handed: Behavioral threat fingerprinting* [online]. 2021 [visited on 2022-04-14]. Available from: https://blog.avast.com/behavioral-threat-fingerprinting-avast.
28. HAHN, Karsten. *Malware family naming hell is our own fault* [online]. 2021 [visited on 2022-04-15]. Available from: https://www.gdatasoftware.com/blog/malware-family-naming-hell.
29. PLOHMANN, Daniel; ENDERS, Steffen. *List of malware families* [online] [visited on 2022-04-15]. Available from: https://malpedia.caad.fkie.fraunhofer.de/families.
30. PLOHMANN, Daniel; ENDERS, Steffen. *Redline Stealer* [online] [visited on 2022-04-15]. Available from: https://malpedia.caad.fkie.fraunhofer.de/details/win.redline_stealer.

31. VARGA, Adam. Identification and characterization of malicious behavior in behavioral graphs. 2021, p. 88. Available also from: `https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=226568`.

32. LAKE, Josh. *Pony: A Breakdown of the Most Popular Malware in Credential Theft* [online]. 2018 [visited on 2022-04-30]. Available from: `https://www.acunetix.com/blog/articles/pony-malware-credential-theft/`.

33. ALI, Muhammad Hasan. *Full RedLine malware analysis* [online]. 2022 [visited on 2022-04-30]. Available from: `https://muha2xmad.github.io/malware-analysis/fullredline/`.

34. X0R19X91. *Analysis of SmokeLoader* [online]. 2021 [visited on 2022-05-01]. Available from: `http://web.archive.org/web/20210704044005/https://x0r19x91.in/malware-analysis/smokeloader/`.

35. ADMIN. *Glupteba back on track spreading via EternalBlue exploits* [online]. 2021 [visited on 2022-05-01]. Available from: `https://labs.k7computing.com/index.php/glupteba-back-on-track-spreading-via-eternalblue-exploits/`.

36. THAKUR, Vishal. *The 'STOP' Ransomware Variant* [online]. 2021 [visited on 2022-05-01]. Available from: `https://angle.ankura.com/post/102het9/the-stop-ransomware-variant/`.

37. SANDER, Jörg et al. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. *Data Mining and Knowledge Discovery*. 2004, vol. 2, pp. 169–194.

38. MCINNES, Leland; HEALY, John; MELVILLE, James. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. arXiv, 2018. Available from DOI: `10.48550/ARXIV.1802.03426`.

Contents of enclosed CD

```
readme.txt ................................. readme file
code ....................................... practical part of thesis
  codebase ................................. common code shared between experiments
  aglomerative_all_embeddings.ipynb ........ agglomerative clustering experiments
  dbscan_all_embeddings.ipynb .............. DBSCAN experiments
  embeddings_analysis.ipynb ................ analysis of the provided embeddings
  figures.ipynb ............................ drawing images for the textual part
  kmeans_all_embeddings.ipynb .............. k-means experiments
  signature_analysis.ipynb ................. signature graph manual analysis
  strain_analysis.ipynb .................... analysis of selected strains from dataset
tex ........................................ thesis text source in LaTeX
  images ................................... images used in the thesis
  text ..................................... content of the thesis split into chapters
diploma_thesis.pdf ......................... text of the thesis in PDF format
```