FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Behavioural graph-based classification of infected network hosts
Student:	Bc. Daniel Nemčík
Supervisor:	Ing. Martin Kopp
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Information Security
Validity:	Until the end of winter semester 2020/21

Instructions

Learn how the function call graphs are constructed in the dynamic analysis of malicious binaries. Modify this approach to construct behavioural graphs for network hosts from network communication captured in proxy logs/netflows. Propose a classification algorithm that will identify infected network hosts based on graph similarity. Compare the classification efficacy of the graph approach with conventional classifiers (random forest, SVM).

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 21, 2019

CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F8

Faculty of Information Technology
Department of Computer Security

Master's Thesis

Behavioural Graph-based Classification of Infected Network Hosts

Bc. Daniel Nemčík

June 2019
Supervisor: Ing. Martin Kopp

Acknowledgements

I want to thank the supervisor, Ing. Martin Kopp, for guidance in the making of this thesis, my family for their continuous support, my sister Slavomíra for help with the text review and finally Cisco Systems for providing me with the assignment of the thesis and infrastructure necessary to carry out the experiments.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act.

Prague, 26. 6. 2019

.....

Abstrakt / Abstract

Táto práca zavádza klasifikátor grafov založený na princípe podobnosti grafov založený na základe princípov využívaných v statickej analýze. Navrhnutý klasifikátor slúži na interpretovateľnú klasifikáciu vzoriek pozostávajúcich z príznakov definovaných na základe správania užívateľa. Navrhnutý klasifikátor je testovaný na reálnych dátach z oblasti sieťovej bezpečnosti poskytnutých firmou Cisco. Interpretovateľnosť klasifikátora je ukázaná na grafe zostrojenom pre triedu advéru. Výsledky klasifikácie sú porovnané s výsledkami dosiahnutými algoritmom náhodný les.

Kľúčové slová: klasifikácia, teória grafov, podobnosť grafov, príznamy správania, interpretovateľnosť modelu

Inspired by concepts used in static analysis, this thesis introduces a graph classifier based on graph similarity. The classifier aims to classify samples consisting of high-level behavioural features in an interpretable way, and is tested on real-world network security dataset provided by Cisco. The resulting model is demonstrated on a graph built to represent adware infection, showing promising results in terms of readability and interpretability. Classification performance of the classifier is compared to performance of a random forest model.

Keywords: classification, graph theory, graph similarity, behavioral features, model interpretability

Contents /

Tables / Figures

Chapter 1
Introduction

Computer networks have become a necessary means of communication by enabling global data transfer and international communication. However, these networks can also be misused by malicious actors, allowing them to reach a high number of targets. Malicious software, also called malware, can quickly spread using computer networks, creating a communication channel between the infected machine and malicious actor, which can result in stealing sensitive information, coordination of multiple malware instances, or malicious advertising distribution.

Since businesses rely on computer networks to operate, securing the networks is one of their main concerns. Network security teams are therefore created to protect critical networks, developing countermeasures against ever-evolving malware. One of such countermeasures is Intrusion Detection System (IDS). IDS is a hardware or a software solution tasked to monitor network traffic and to warn network administrator in case an intrusion is detected. Based on the detection method, IDS can be categorised either as signature based, or anomaly based. Signature-based IDS are excellent in detecting well-known attacks, but struggle to identify either novel or modified, so-called zero-day attacks. Anomaly-based IDS are developed to counter the unseen and zero-day attacks by using statistical models and machine learning to detect network communication deviating from regular traffic.

The application of machine learning helps to process large amounts of data, helping human analysts to do their work much more efficiently. Unfortunately, the scope of suitable machine learning algorithms is often limited by the fact, that an explanation of algorithm's decision is in many cases almost impossible. An explanation of the algorithm's choice, e.g., why an algorithm decided to label a specific sample as positive, is necessary for reliable identification of the cause. In network security, knowing there is an infected host in the network is beneficial, but the contextual information is vital when deciding how to act against it, and whether the detection is correct or a false positive. For example, algorithms like neural networks are usually one of the most accurate classifiers, but their nature makes the explanation of their choice almost impossible. Therefore, classification based on decision trees or Bayesian classifiers suits the needs of network security much better due to their interpretability.

Another inherently interpretable mean of modelling relations of network communication is a graph structure. Graphs are already successfully used in static analysis of executable binaries to classify malicious software. Code graphs, call graphs and flow graphs built during static analysis are a decomposition of the application's structure. Such graphs are commonly used to determine the degree of similarity between the decomposed binaries. In a similar vein, graph structure

can be utilized to model malware communication by processing information about the presence of anomalous events in captured communication. Such graph models can be used to detect infection presence in an unknown communication using graph similarity in an interpretable way.

This thesis proposes an interpretable graph-based classification model suitable for malware communication detection similar to anomaly-based IDS approach. The model can be used to classify anomalous communication detected in the network flows of real network traffic and to detect infected entities on the network in an interpretable and visualizable way.

1.1 Problem Outline

This thesis aims to create a classification system capable of identifying infected hosts on the network, along with an interpretable graph model of the anomalous communication of the host. An outline of the approach is given in the following steps:

1. *Data acquisition.* Real network traffic data is collected in the form of network flows.
2. *Anomaly Detection.* Flows are analysed for anomalies and tagged by the Cognitive engine.
3. *Feature Extraction.* Flows known to be malicious are aggregated based on the class tag from Cognitive. Flows are converted to sets of anomalous events and aggregated for a given period.
4. *Learning and Classification.* Relations between anomalous communication events and the target label are weighted, graphs are built separately for tagged (malicious) and untagged aggregated event sets. Graphs are then classified using Graph Edit Distance similarity measure.
5. *Explanation.* Graphs similar to the graphs of malicious communication can be visually analysed due to the inherently interpretable nature of the graph structure.

Chapter 2
Machine Learning: Classification

This chapter serves as an overview of basic concepts used in machine learning, namely in classification. The first three sections establish machine learning terminology and notation used in the thesis and describe the algorithms applied in this thesis. The fourth section explains and gives solution to problems brought by processing high-dimensional data. Finally, the last section describes explainability as an essential concept in the context of malware detection.

2.1 Learning From Data

Machine learning is a field of research dealing with the automated analysis of data. This field focuses on creating algorithms to analyse the data and also on building models from the inferred information, using principles with a strong foundation in mathematics and statistics. It is becoming more important with increasing amounts of data, finding its use across many domains.

Each data sample processed by a machine learning algorithm is represented by a vector x. An output y, also called a *label*, is assigned to each sample. If $y \in Y$, where Y is a finite set of labels, the problem of creating a model that accurately assigns an output y to a sample x is called a *classification problem*. If the output is continuous, this is called a *regression problem*. Unless stated otherwise, this thesis focuses only on classification problems.

2.2 Supervised Learning

In supervised learning, the main task is to create a model from labelled data samples. This model serves to classify unseen data samples based on the knowledge extracted from the labelled data. Source [1] defines supervised learning in the following way:

Definition 2.1. (Supervised learning.)[1] Given a set of data $D = \{(x^n, y^n), n = 1, \ldots, N\}$, *supervised learning* aims to learn a mapping from an input x to output y such that, when given a novel input x^* the predicted output y^* is accurate. The pair (x^*, y^*) is not in D, but we assume it to be generated by the same unknown process that generated D. To specify what accuracy means one defines a loss function $L(y^{pred}, y^{true})$ or, conversely, a utility function $U = -L$.

The utility, or the loss functions quantify how costly making a classification error is. The classification algorithm provides a predictive probability distribution $p(c|x^*)$ for an unseen sample x^*. The distribution is, however, only approximate,

since the true class is not known to the classifier. If $U(c^{true}, c^{pred})$ represents the utility of making a decision c^{pred}, the expected utility for the decision function is

$$U(c(x^*)) = \sum_{c^{true}} U(c^{true}, c(x^*))p(c^{true}|x^*) \tag{2.1}$$

and the optimal decision function $c(x^*)$ maximizes the expected utility,

$$c(x^*) = \text{argmax}(U(c^x)) \tag{2.2}$$

as defined by [1].

A simple count of incorrect predictions is a *zero-one* loss function

$$L(y^{pred}, y^{true}) = \begin{cases} 0 & \text{if } y^{pred} = y^{true} \\ 1 & \text{if } y^{pred} \neq y^{true} \end{cases} \tag{2.3}$$

which can be interpreted as a count of incorrect predictions, or when converted to a utility function, as a count of correct predictions [1]. In case of binary classification, expected utility is given by

$$U(c(x^*)) = \begin{cases} p(c^{true} = 0|x^*) & \text{for } c(x^*) = 0 \\ p(c^{true} = 1|x^*) & \text{for } c(x^*) = 1 \end{cases} \tag{2.4}$$

This means the decision function should correspond to selecting the class with the highest class probability

$$c(x^*) = \begin{cases} 0 & \text{if } p(c = 0|x^*) > 0.5 \\ 1 & \text{if } p(c = 1|x^*) > 0.5 \end{cases} \tag{2.5}$$

Source [1] also defines a general utility function, using which one can enforce a particular class being preferable in terms of utility, which means misclassification of the class is penalised more if the wrong guess is not acceptable. This characteristic can also be ensured in a more straightforward way, using confusion matrix metrics.

2.2.1 Confusion Matrix

Classification performance metrics taken from the confusion matrix are commonly used in binary classification in case there are specific requirements for classification. For instance, the classification of oncology patients in medicine requires the classifier to make no false predictions, but at the same time to correctly classify all patients who need treatment.

The confusion matrix metrics are more commonly used in the evaluation of the models, not so often in training, when compared to the utility/loss approach. When the model learning is formulated as an optimisation problem, having a metric to optimise is necessary. The confusion matrix defines several metrics suitable to the needs of different optimisation problems.

The confusion matrix consists of four types of predictions a classifier can make:

- *True Positive* (TP)
- *True Negative* (TN)
- *False Positive* (FP)
- *False Negative* (FN)

Positive and *Negative* are the two classes present in binary classification. If the classifier predicts a positive class for a sample that is positive, the classifier is said to have made a *True Positive* prediction. On the other hand, if the classifier marks a sample as positive, but the correct class is negative, the classifier made a *False Positive* prediction. Counting all such cases gives the number of TP and FP cases, respectively, or TPs and FPs. TNs and FNs are defined analogously. The relation between classifiers predictions and true labels can be visualised on *Confusion Matrix* on the picture 2.1, where TPs, FPs, TNs and FNs represent the number of the respective cases.

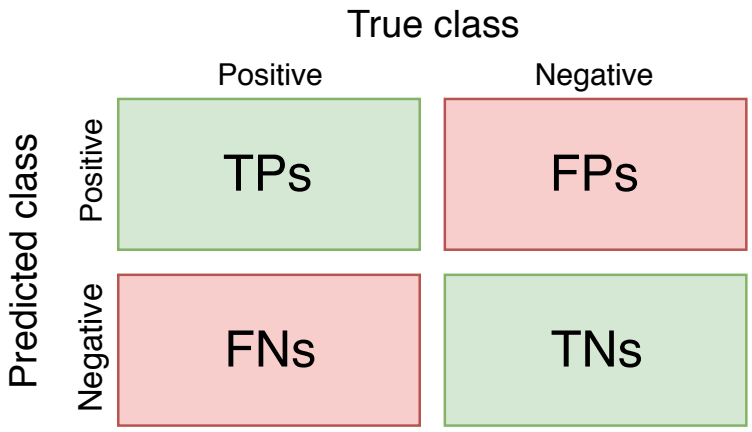


Figure 2.1. Confusion matrix.

Classifier's ability not to make false classifications in context of positive class can be shown by calculating classifier's *Precision*, also called Specificity depending on the literature.

$$Precision = \frac{TPs}{TPs + FPs}$$

Another important measure used to evaluate classifier's ability to correctly classify all positive samples is *Recall*.

$$Recall = \frac{TPs}{TPs + FNs}$$

Recall is also often called Sensitivity in sources dealing with machine learning in medicine.

2.2.2 Precision-Recall Curve

The Precision-Recall curve (PR curve) serves as a visual evaluation tool to show the precision-recall tradeoff of the current setting of the classifier by changing the classification probability threshold of the positive class and evaluating precision and recall of the model for each threshold setting.

The PR curve is constructed in the following way. The decision function of the classifier is defined similarly as the function (2.5), a difference being that only the positive class is considered, meaning how likely is x^* to belong to the positive class:

$$c(x^*) = \begin{cases} 1 & \text{if } p(c = 1|x^*) > p' \\ 0 & \text{otherwise} \end{cases}$$

Here, threshold p' is set to 1 in the beginning, and the performance of the classifier is evaluated, noting the precision and recall of this setting. The threshold is gradually decreased, for example by 0.05 in each step, noting the precision and recall, until the threshold reaches 0.

Plotting the noted values with precision on axis y and recall on axis x results in a PR curve, with an example on figure 2.2.

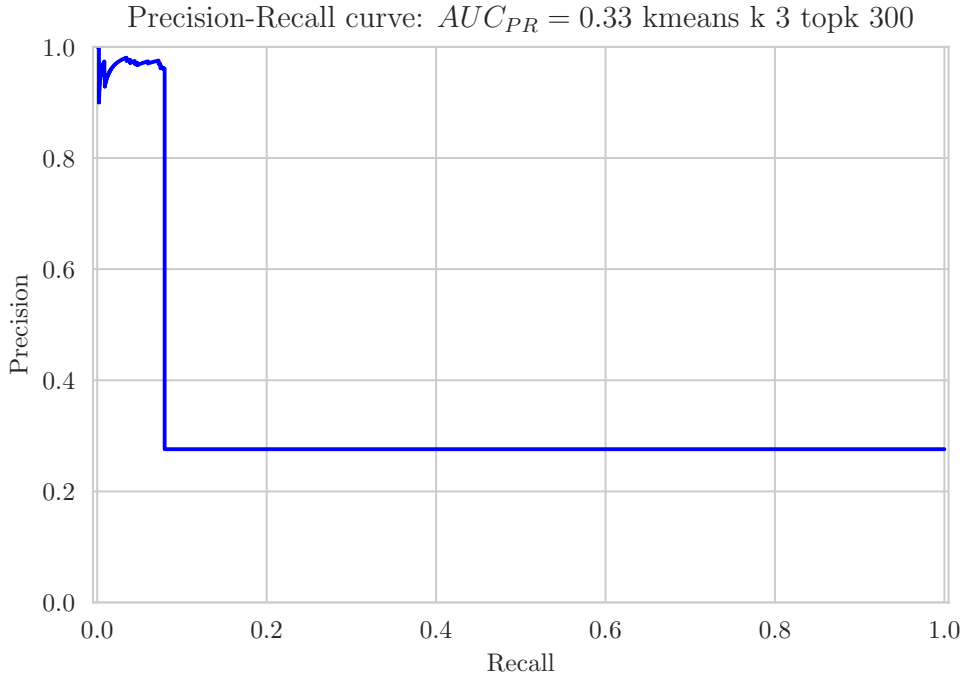


Figure 2.2. PR Curve with AUC_{PR} of 0.33.

The PR curve is an alternative to the commonly used Receiver Operating Characteristic curve (ROC curve), which plots the trade-off between recall and false positive rate. This thesis utilises the PR curve because of its flexibility of evaluation on datasets with imbalanced negative to positive class ratio, where the ROC curve might not accurately represent the model's performance, as noted by the authors of source [2].

The area under the PR curve (AUC_{PR}), also called Average Precision, can also be used as a classification evaluation metric. AUC_{PR} shows how well is the classifier able to separate the two classes in an average case. The perfect classifier would have the AUC_{PR} of 1.0, which means all positive samples are marked as positive with probability of 1.

2.2.3 Class Separation Histogram

Another useful visual tool for evaluating the classifier's performance is plotting a histogram per class of the data used in the evaluation. One histogram is created for each class in the testing data. The histogram contains positive class probabilities assigned to the samples of the given class. In the ideal case, the probabilities for positive samples are close to 1, while samples from other classes are assigned probability close to zero. Overlap of these histograms shows how well is the classifier able to distinguish the positive class from other classes.

Figure 2.3 provides an example of the Class Separation Histogram.

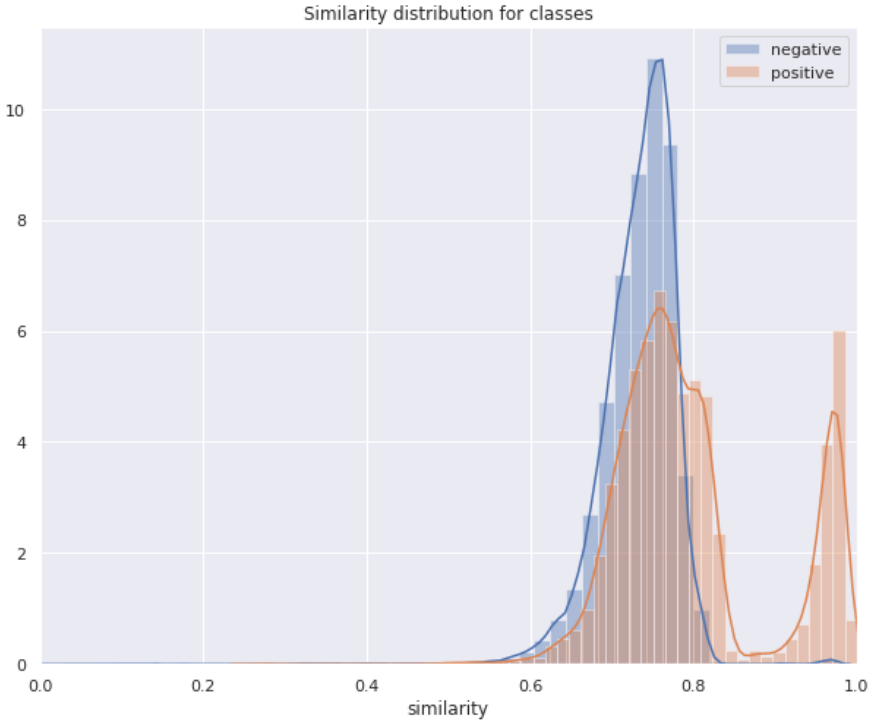


Figure 2.3. Class Separation Histogram.

2.2.4 Decision Tree and Random Forest

Decision tree is a simple but effective model. The simplicity allows for easy interpretation of the model's decision, straightforward visualisation and excellent performance on large datasets. Decision trees can be used for binary and also for multi-class classification tasks. The trees work with discrete features, but also with continuous features by performing implicit discretisation similar to binning. Furthermore, the decision tree is a non-parametric model, which means it can model arbitrarily complex relations of input and output. The decision tree model is also robust to outliers and label errors. [3]

The source [4] describes the decision trees as a set of nodes and edges organized into a hierarchical structure. The tree consists of internal nodes, also called split nodes, and terminal nodes called leaf nodes[1]. Data is processed by decision tree in two phases. The first phase called training phase consists of processing the dataset

[1] The definitions of graph theory terminology is provided in section 3.1

by passing the dataset through the root node of the tree, splitting the dataset based on a task-independent objective metric, such as commonly used Information Gain. Feature importance of each feature can be quantified by calculating how often a given feature is selected for a split, and how good the split was in terms of Information Gain.

Let us say the dataset X_j considered for a split in the node j. The information gain of the split is calculated by

$$I_j = h(X_j) - \sum_{i \in \{L,R\}} \frac{|X_j^i|}{X_j h(X_j^i)} \tag{2.6}$$

where $h(X_j)$ is the Shannon Entropy. The split of the labelled dataset X_j is made by dividing the dataset into two parts based on the values of a single feature. Ultimately, the split with the highest Information Gain is selected, the dataset is divided into two parts, and each part is passed to the descendants of the node. The splitting is done repeatedly, growing the tree, until a stopping criterion is reached, which can be set as a maximum depth of the tree or a minimum information gain. The resulting split of the whole dataset is contained within the tree leaves by preserving the labels of the samples that made it to the leaf node. The ratio of the class labels is used to create a probability distribution $p_l(c|X_j)$.

The testing phase consists of passing vectors of testing samples through the tree. Node passes the sample x^* further below based on the split learned in the training phase until the sample reaches a leaf node. The class probability $p_l(c|x^*)$ learned beforehand is then used in the decision function to classify the sample x^*.

However, the decision trees tend to overfit the data, mainly when the tree grows to consist of many layers. One of the solutions is trimming the tree, which discards some of the information to make the tree able to generalise better. Another solution is creating an ensemble of trees, which is done by the Random Forest algorithm.

Random Forest algorithm, initially proposed by Breiman in [5], is currently one of the most popular machine learning algorithms. It serves as a great baseline classifier because it requires almost no adjustment to the default parameters to achieve good classification results. The resulting decision boundary of the ensemble can be highly non-linear, which makes the Random Forest able to reach decent results when the classes are not linearly separable.

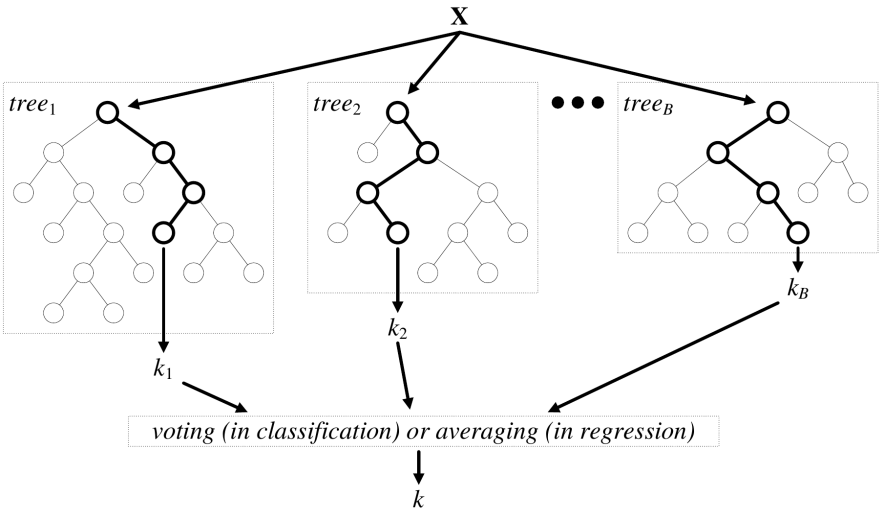


Figure 2.4. Random Forest structure [6].

The Random Forest T consists of several decision trees t. Each tree is built on a randomly selected subset of features, creating multiple different splits of the dataset. Each tree also learns only on a subset of the training data, created by random sampling with or without replacement.

In the training phase, the leaf probability distributions are combined by averaging the probability of all trees in the ensemble:

$$p(c|x^*) = \frac{1}{T}\sum_{t=1}^{T} p_t(c|x^*) \tag{2.7}$$

Therefore, the resulting class assigned to the sample x^* is the class with the highest average probability. Alternatively, a majority vote determines the class – the ensemble chooses the class assigned by the majority of the trees.

2.3 Unsupervised Learning

When the provided dataset X lacks the output information y, the algorithms used in the supervised learning cannot be used. The lack of labels is widespread, mainly when working with a novel data source. However, such data can still be analysed by approaches that find structure, look for anomalous data, and find data clusters.

Definition 2.2. (Unsupervised Learning.)[1] *Unsupervised learning* aims to find a plausible compact description of the dataset $D = \{x^n, n = 1, ..., N\}$. In contrast to supervised learning, which intends to model a conditional probability distribution $p_X(x|y)$, the output variable y is unknown. The unsupervised learning intends to model a prior probability distribution $p_X(x)$.

2.3.1 Clustering

Clustering is an approach of finding groups of data instances that are highly similar, lie in densely populated regions, fit into a specific statistical distribution, or are close to pre-defined centroids.

Algorithm *k-means* is made to group data instances x in the dataset D into k clusters. Source [7] describes the algorithm in the following way. At the start of the k-means algorithm, the means $\{m^{(k)}\}$ are initialised in some way, for example to random values. The k-means is then an iterative two-step algorithm, for which in the first step, also called assignment step, each data point x is assigned to the nearest mean. In the update step, the means are moved to be located in the centre of the data points to which they are assigned. These steps are repeated until the algorithm converges. The K-means algorithm always converges to a fixed point, when the means do not change their position after the update step.

Initial placement of the means and the number of means is variable. Either domain knowledge is required to set the k, or an arbitrary number of the means can be selected - k clusters will be found after the algorithm converges either way. In terms of the initial mean placement, there are cases, when the resulting means get stuck in a sub-optimal configuration. There are heuristics to deal with this problem.

2.4 Dimensionality

Dimensionality refers to the dimension of the dataset or number of features in the dataset. Various problems arise with high dimensionality. The problems with their solutions are described in this section.

2.4.1 The Curse of Dimensionality

The curse of dimensionality is a term used to describe the problems caused by the increasing number of dimensions in a Euclidean space, where the volume of the space increases exponentially with an increasing number of dimensions. For the nearest-neighbour search methods in high-dimensional spaces, this means that the distances to all data points become almost identical for all samples. [8]

The possible solutions to the curse of dimensionality include using non-Euclidean distance measures such as cosine distance where applicable, or using methods of feature selection or dimensionality reduction, which some sources call feature extraction. Feature selection keeps the actual meaning of each selected feature, which makes it superior in terms of feature readability and interpretability when compared to feature extraction. [9]

2.4.2 Dimensionality Reduction

Dimensionality reduction is a term covering methods that reduce the number of features in the data by transforming the data into its lower-dimensional representation. The main goal is to reduce the dimensionality while preserving as much information as possible. The resulting lower-dimensional embedding of the input data can also be used to visualise the dataset.

The methods used in dimensionality reduction do not preserve original features but create new features based on the original features, which might later pose a problem for explaining the decision of the learning algorithm.

One of the most commonly used algorithms used to reduce the dimensionality is the Principal Component Analysis (PCA). It consists of creating a new dataset consisting of features called principal components. These new features are created to capture the variance of the data while being orthogonal to each other.

UMAP [10] is a novel approach to dimensionality reduction based on manifold theory and topological data analysis. It is non-linear, supports a wide variety of distance metrics and tries to preserve both local and global structure of the data when creating its lower dimensional representation.

2.4.3 Feature Selection

Feature Selection is an approach of dimensionality reduction used to remove irrelevant features from the data. Features not useful for the learning task are removed, which reduces the amount of data for the learning algorithm to process, while not affecting the performance of the model. Feature selection often even increases accuracy of the learned models.

The feature selection methods can be categorised into three groups:

- *Filter* – Features are ranked based on a specific criterion. Highest-ranked features are chosen, and the rest are discarded. Commonly used criteria are statistical dependence, correlation of features, mutual information.
- *Wrapper* – The subset of features is evaluated based on the performance of a learning model learned on the subset. The subset is changed iteratively until the best subset of the features is found. Heuristics help to reduce the search space of such a search.
- *Embedded* – Feature selection is made during the creation of the learning model. For example, Random Forest performs feature selection during tree construction, which can be called embedded.

According to [9], the main benefit of filter methods is their efficiency, but since these methods do not consider the learning algorithm used afterwards, they might miss the combination of the features that might be relevant to the learning algorithm.

As for wrapper methods, they perform much more thorough search of optimal feature combinations, which in turn makes them much more computationally expensive compared to filters, but generally achieving better results. [9]

Frequency-based feature selection is a simple method that can be used to rank features based on how common a feature is for a class.

$$S_t = \sum n_{t,c} \tag{2.8}$$

where $n_{t,c}$ is a number of occurrences of feature t for class c.

Mutual Information is a measure used in information theory, which is used to show the degree of mutual dependence between two random variables, for which probability distributions are known. Zero means the random variables are independent.

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p_{(X,Y)}(x,y) \log_2 \left(\frac{p_{(X,Y)}(x,y)}{p_X(x) p_y(y)} \right) \tag{2.9}$$

The output of mutual information can be used for feature selection. Each feature is evaluated for its mutual information with the target class. In case distribution of the features is not known, a nearest-neighbour-based method is used for estimation, described in detail in [11].

Chi-square test is a statistical hypothesis test used for testing of independence of two events or categorical random variables. Source [12] defines it for feature selection as

$$X^2(D,t,c) = \sum_{e_t \in \{0,1\}} \sum_{e_c \in \{0,1\}} \frac{(N_{e_t e_c} - E_{e_t e_c})^2}{N_{e_t e_c}} \tag{2.10}$$

where e_t is a value occurring in a feature t and e_c is a possible class. N is an observed frequency in D and E is the expected frequency. For example, E_{11} is the expected frequency of t and c occurring together assuming the feature and class are are independent.

Features can be ranked using the calculated value, with higher values meaning the feature and class are less likely to be dependent.

Fisher Score is similar to the Chi-square test, usable also for continuous random variables. The test consists of determining whether the variance between the means of two random variables is significantly different. For feature selection, one of the random variables is a feature, and the other one is a class vector.

Source [13] provides a formula for calculating a Fisher Score for feature t:

$$S_t = \frac{\sum_{c \in C} n_c (\mu_{t,c} - \mu_t)^2}{\sum_{c \in C} n_c \sigma_{t,c}^2} \tag{2.11}$$

where $\mu_{t,c}$ and $\sigma_{t,c}$ are mean and variance of the feature t given class c, n_c is the number of instances of the class c and μ_t is mean of the feature t.

2.5 Explaining the Classifier's Decision

When it comes to using machine learning models for making important decisions, such as disease prediction in medicine, it is important to understand why the model made a particular decision. In such cases, model accuracy is not a sufficient metric when evaluating a model. Source [14] provides an example when a better scoring classifier makes a prediction based on a set of features irrelevant to the problem it was solving, while the lower-scoring model used relevant features and its prediction made much more sense in the context. This shows that the resulting models should not be trusted blindly when an important decision is being made.

In an ideal case, inherently interpretable models should be used to model such problems, like decision trees or Bayesian classifier. Another solution to this problem is creating an interpretable model to approximate a model, which is the primary goal of LIME [14].

Chapter 3
Graph Theory

This chapter presents graph theory concepts used in this thesis, in regards to terminology, graph similarity algorithms and their application on graph comparison. Terminology in this chapter is defined according to source [1].

3.1 Graph Theory Fundamentals

Graph theory is a study of graph structures, which are used to model and analyse relations between pairs of entities.

Definition 3.1. (Graph)[1] A *graph*

$$G = \{V, E\}$$

consists of nodes V (also called vertices) and edges E (also called links) between the nodes with associated weights $w : E \mapsto [0, 1]$, for each $e \in E$. Node labels and edge weights are optional. Edges may be directed (can be traversed only in a single direction) or undirected. Edges can also have associated weights. Graph with all edges directed is called a *directed graph*, and one with all edges undirected is called a *undirected graph*. $|G|$ represents the size of the graph as a count of edges. In case the edges are weighted, $|G|$ represents the sum of all weights of the edges.

Directed edges are used to model concepts where the direction of the relation is important; for example, call graphs used in dynamic malware analysis to model which function called which.

All graphs in this thesis are labelled unless stated otherwise. Presence of weights is always specified, and graph with weights is called a *weighted graph*.

Definition 3.2. (Subgraph) Graph G is a subgraph of graph H if all nodes and edges of G are present in graph H.

Definition 3.3. (Path, ancestors, descendants)[1] A *path* AB from node A to node B is a sequence of nodes that connects A to B. That is, a path is of the form $A_0, A_1, \ldots, A_{n-1}, A_n$, with $A_0 = A$ and $A_n = B$ and each edge $(A_{k-1}, A_k), k = 1, \ldots, n$ being in the graph. A directed path is a sequence of nodes which when we follow the direction of the arrows leads us from A to B. In directed graphs, the nodes A such that $A \mapsto B$ and $B \not\mapsto A$ are the *ancestors* of B. The nodes B such that $A \mapsto B$ and $B \not\mapsto A$ are the *descendants* of A.

Definition 3.4. (Directed Acyclic Graph (DAG)) A DAG is a graph G with directed edges between the nodes such that by following a path of nodes from one node to another along the direction of each edge no path will revisit a node. In

a DAG, the ancestors of B are those nodes who have a direct path ending at B. Conversely, the descendants of A are those nodes who have a directed path at A.

Definition 3.5. (Neighbour)[1] For an undirected graph G the neighbours of x, $\mathrm{N}(x)$ are those nodes directly connected to x.

Definition 3.6. (Clique)[1] Given an undirected graph, a clique is a fully connected subset of nodes. All the members of the clique are neighbours; for a maximal clique, there is no larger clique that contains the clique.

Definition 3.7. (Connected Graph)[1] An undirected graph is connected if there is a path between every pair of nodes (i.e. there are no isolated islands). For a graph which is not connected, the *connected components* are the subgraphs that are connected.

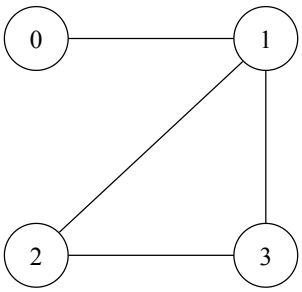
Definition 3.8. (Singly connected graph (tree))[1] A graph is *singly connected* if there is only one path from any node A to any other node B. Otherwise, the graph is *multiply connected*. This definition applies regardless of whether or not the edges in the graph are directed.

Definition 3.9. (Graph Union) *Graph union* of graphs G and H is a graph $K = \{V_G \cup V_H, E_G \cup E_H\}$. If an edge is present in both G and H and is weighted, an average of weighs of the edges is carried over to the graph K.

3.1.1 Numerically Encoding Graphs

To represent graph structures on a computer, the numerical encoding of the graph is necessary. *Edge list* lists which node-node pairs are in the graph.

An alternative is to use an *adjacency matrix* A, where $A_{ij} = 1$ if there is an edge from node i to node j in the graph. If the edges of the graph are weighted, $A_{ij} = weight((i, j))$.



$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \quad E = \{(0,1),(1,2),(1,3),(2,3)\}$$

Figure 3.1. Example of a graph, it's adjacency matrix A and edge E list.

3.2 Graph Similarity

Graph similarity serves to quantify the degree of similarity of two graphs. Depending on the type of graphs, source [15] proposes two main categories of graph similarity algorithms based on node correspondence: known node correspondence and unknown node correspondence. Known node correspondence means that the graphs being compared have the same set of nodes and associated node labels, with a possible difference in edges. When nodes of the two graphs being compared

are different (i.e. function call graphs), methods from unknown node correspondence can be used, however, with more substantial performance cost compared to the first class.

- *Known Node Correspondence*: Graph Edit Distance (GED), Maximum Common Subgraph (MCS), DeltaCon
- *Unknown Node Correspondence*: Feature Extraction, Graph Kernels, λ-distance

The similarity metrics and distance metrics are considered to be easily convertible between each other. If the distance metric produces values between 0 and 1, $similarity = 1 - distance$. If not, the outputs of the distance metric need to be normalised (e.g. by min-max normalisation).

3.2.1 Graph Edit Distance

Graph Edit Distance (GED) is one of the most straightforward but efficient graph similarity measures. The graph similarity is represented by an amount of graph edit operations needed to make the graphs to consist of the same edges and nodes. Source [16] provides an excellent and thorough definition of the method. Parts of it relevant to this thesis are contained in the following paragraphs.

Given two graphs, the source graph $G_1 = \{V_1, E_1\}$ and the target graph $G_2 = \{V_1, E_2\}$, the basic idea is to transform G_1 into G_2 using some edit operations. A standard set of operations consists of *insertions, deletions* and *substitutions* of both nodes and edges. Substitution of two nodes $u \in V_1$ and $v \in V_2$ by $(u \to v)$, the deletion of node $u \in V_1$ by $(u \to \varepsilon)$, and the insertion of node $v \in V_2$ by $(\varepsilon \to v)$, where ε refers to the empty node. Similar notation is used for edge edit operations. [16]

Definition 3.10. (Edit Path)[16] A set $\{\kappa_1, \ldots, \kappa_n\}$ of n edit operations κ_i that transform G_1 completely into G_2 is called a *complete edit path* $\lambda(G_1, G_2)$. A *partial edit path*, i.e. a subset of $\{\kappa_1, \ldots, \kappa_n\}$, edits proper subset of nodes and/or edges of the underlying graphs.

Let $\upsilon(G_1, G_2)$ denote the set of all complete edit paths between the two graphs. To find the most suitable edit path out of $\upsilon(G_1, G_2)$, one introduces cost $c(\kappa)$ for every edit operation κ, measuring the cost of the corresponding operation. Clearly, between two similar graphs, there should exist an inexpensive edit path, representing low-cost operations, while for different paths an edit path with high cost is needed. Consequently, the edit distance of two graphs is defined as follows. [16]

Definition 3.11. (Graph Edit Distance) [16] Let $G_1 = \{V_1, E_1\}$ be the source and $G_2 = \{V_1, E_2\}$ the target graph. The *graph edit distance* $d_{\lambda_{min}}(G_1, G_2)$ or $d_{\lambda_{min}}$ for short, between G_1 and G_2 is defined by

$$d_{\lambda_{min}}(G_1, G_2) = \min_{\lambda \in \Upsilon(G_1, G_2)} \sum_{\kappa_i \in \lambda} c(\kappa_i) \tag{3.1}$$

where $\Upsilon(G_1, G_2)$ denotes the set of all complete edit paths transforming G_1 into G_2. c denotes the cost function measuring the strength $c(\kappa_i)$ of node edit operation

κ_i including the cost of all edge edits implied by the operations applied on the adjacent nodes of the edges, and λ_{min} refers to the minimal cost edit path found in $\Upsilon(G_1, G_2)$.

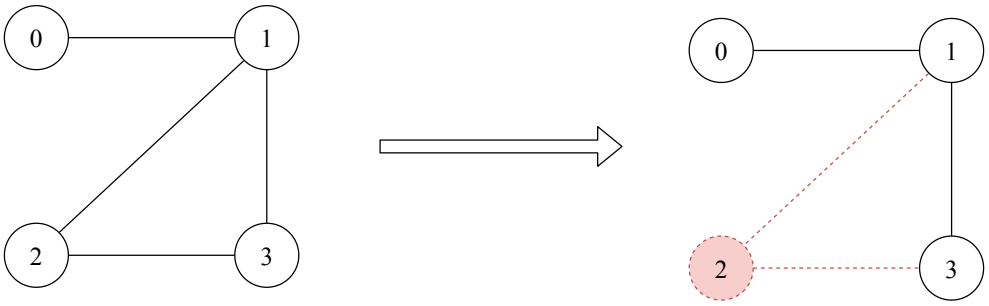


Figure 3.2. Example of Graph Edit Distance between original graph G_1 and modified G_2, $d_{GED}(G_1, G_2) = 3$. Red dashed line means the edge or node is removed.

The cost function assigns a cost to all edit operations. [16] describes how to assign a cost to the operations, so the resulting cost function had good properties. Exact Edit Distance is computed by an A*-based search technique also described by [16].

Depending on the problem being solved, simplifications of the GED are possible. First, do not consider node edit operations, i.e. graphs being compared have the same set of nodes V. This means only missing, or additional edges need to be addressed, which hugely simplifies the algorithm. If both edge insertion and edge deletion have the same cost, exact GED is given by the following.

$$d_{GED*}(G_1, G_2) = \sum |A_1 - A_2| \tag{3.2}$$

A_1 and A_2 are the adjacency matrices of graphs G_1 and G_2 respectively.

3.3 Edge Weighting

Edge weighting serves to encode the importance of an edge in the graph. The importance is either defined by domain knowledge, or domain-independent metric inspired by feature importance in feature selection.

1. *Edge Frequency.* Edges can be weighted with the number of times they occur. Frequently occurring edges might be much more informative compared to rare edges.
2. *Edge Importance.* Feature importance of a label can be calculated for edges in the graph, examples of which are described in section 2.4.3.

3.4 Graph Density

Graph density is an important aspect to keep in mind when it comes to visual interpretation of graphs. Authors of [17] define two graph density measures for, d and d_l.

$$d = \frac{m}{n} \tag{3.3}$$

$$d_l = \sqrt{\frac{m}{n^2}} \tag{3.4}$$

Here, m represents the number of edges of the graph, and n represents the number of nodes. Both measures can be used for graph with undirected edges with no self-loop edges. Measure d represents the ratio of edges of the graph compared to a complete graph given the number of nodes. d_l is a number of edges per node.

For a visual representation of the graphs, if the density is too high, it is impossible to properly lay out a graph so it can be visually analysed, according to [17]. Generally, graphs with lower density are easier to comprehend.

3.5 Graph Partitions and Modularity

Graph analysis includes community detection algorithms, which find partitions in the graph called communities. A community is a set of nodes that are densely interconnected internally, but have only a few connections with nodes outside of the community. Finding such communities helps with analysis of the graph structure by dividing it into parts, for example in case the graph is too large to be visualized whole.

There are multiple algorithms that are used to search for communities. One of such algorithms is called modularity maximisation clustering. Blondel et al. [18] describes an implementation of modularity clustering based on modularity gain of moving a vertex to a different partition. Modularity is used in graph network theory to measure quality of graph partitions, computed for a weighted graph as defined in [19]

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{3.5}$$

where $A_{i,j}$ an element of the adjacency matrix A containing weight of the graph edge from vertex i to j, $k_i = \sum_j A_{i,j}$ is the sum of the edge weights for edges attached to vertex i, c_i is the community to which the vertex i is assigned, the δ-function $\delta(u, v)$ is 1 if $u = v$ and 0 otherwise and $m = \frac{1}{2} \sum_{i,j} A_{i,j}$.

The approach introduced by Blondel et al. [18] starts with all of the vertices put into a different partition. Next, each vertex is passed iteratively, changing the partition of the vertex to a partition of its neighbour in case the modularity gain of performing the change is positive, repeated until no increase in modularity is possible.

Chapter 4
Graph-based Classification

This chapter establishes a graph-based classification method, which is the main topic of the thesis. The first part of the chapter serves as an overview of behavioural graph-based methods used for malware detection in static and dynamic analysis of malicious binaries. The second part describes the proposed graph classification method in detail.

4.1 Basic Blocks as high-level actions

Many of the sources mentioned in the following sections utilise a Control Flow Graph (CFG) structure for behavioural analysis. CFG is a directed graph in which nodes represent basic blocks of code, and edges represent relations between the basic blocks. A basic block is a part of the code with one entry point and one exit point. The model of flow relations between the basic blocks is used to simplify and optimise the high-level code structure. The concept of CFG was first introduced by Allen in [20].

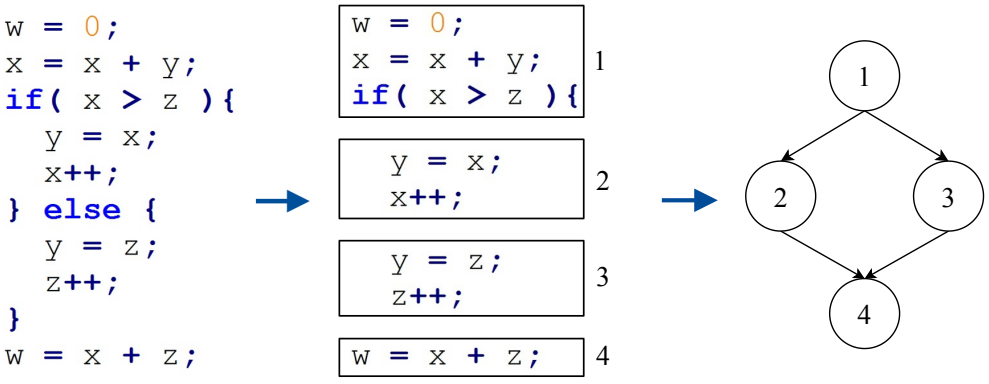


Figure 4.1. Basic block and Control Flow Graph construction. The image is based on source [21].

Figure 4.1 shows an example of CFG construction from a simple code snippet. Basic blocks are defined by their entry and exit points, with target of a jump instruction being the entry point and the jump being an edge of the resulting CFG.

Behavioural malware analysis often utilizes the concept of CFG to fight code obfuscation and polymorphism in both static and dynamic analysis. The behavioural approach utilises executable's characteristics like basic block structure, system call relations and resource usage. These characteristics do not change

even if the executable's code changes due to polymorphism. The unchanging aspects can be understood as high-level actions, representing nodes of a behavioural graph, while relations between these actions represent edges of the graph.

The high-level action behavioural model serves as a basis for graph classifier described later in this chapter.

4.2 Malware detection and usage of the graph structure

This section serves as an overview of works dealing with behavioural malware analysis.

4.2.1 Static Malware Analysis and Graph Methods

Static analysis is an approach to code analysis, where an executable file is analysed to detect malicious code without executing the binary. It is performed by disassembling an executable binary into its components, which is either done by a malware specialists or by an automated processes tailored to detect malicious code. The disassembly uncovers structural and functional information needed to determine whether the binary is malicious or benign. Reverse engineering is often applied to convert the machine code into assembly, which allows to construct a CFG.

Schultz et al. [22] introduce one of the first works utilising machine learning techniques to make malware detection more efficient. They utilise the Portable Executable (PE) structure to extract features such as a list of imported libraries or a number of library function calls. Additionally, they use strings of printable characters and byte sequences to produce more features later processed by a classification approach inspired by a Naive Bayes-based algorithm.

Malicious actors implement countermeasures against static analysis in a form of code obfuscation. Obfuscation makes the code hard to disassemble by both automated processes and human analysts by employing ambiguous language constructs without changing the intended functionality.

As an effort against code obfuscation, source [23] proposes a method of behavioural analysis based on CFG. The graphs are built for each executable sample and processed by a state automaton based on domain knowledge, modelled to be immune to code obfuscation.

4.2.2 Dynamic Malware Analysis and Graph Methods

Malicious actors employ polymorphism to change the characteristics of the malware. Polymorphism includes modifying process and file names, encrypting parts of the code to avoid static analysis and swapping the encryption keys periodically, which result in a change of the executable's signature without changing the intended functionality. As a result, the signature-based detection is rendered ineffective. However, even if the signature changes, what does not change is the behaviour of the malware.

With ever improving sophistication of code polymorphism, executable analysis independent of the way the executable is built is necessary. One of such methods is dynamic analysis. Unlike in the static analysis, the executable is executed in a controlled sandbox environment, while all of its actions like system calls, input/output events, and registry modifications are closely monitored and analysed to specify executable's behaviour. Even if the code of the executable changes to avoid detection, the behaviour cannot change easily without changing the functionality, which is what makes the dynamic analysis more flexible when compared to static analysis.

Dynamic analysis often utilises the graph structure to encode the behaviour of the analysed executable. Authors of [24] propose a behavioural analysis of data flows. A data flow is a transfer of any amount of data between two system entities, which are the nodes of the data flow graph. Directional edges encode the transfer direction. The graph is evaluated by applying high-level behaviour heuristics, which are built based on domain knowledge.

The authors of [25] utilise function calls of the executable to build a function call graph, an alternative type of CFG where functions instead of basic blocks are represented as graph nodes. Nodes of the graph represent function calls and their arguments, and directional edges encode function call relations. A labelling function is established to describe properties of the nodes. When measuring pairwise graph similarity, the labels are used to determine node correspondence between two graphs. Pairwise graph similarity is finally computed based on vertex overlap ratio, which in the author's claim is much more time efficient compared to commonly used graph edit distance.

A similar approach is used by Park et al. [26], with the main difference being that the system call relations of a process are modelled in a form of a kernel object behavioural graph. The distance between graphs is calculated by a maximum common subgraph-based distance algorithm. In their following work, Park et al. [27] improve on their idea by weighting the directed call graph and combining the call graphs in a same malware family into a single supergraph to represent the behaviour of the malware family. A subgraph of the common supergraph is used to represent the malware family. Any sample similar to the subgraph can be considered to belong to the family with high confidence.

4.2.3 Network Malware Analysis and Graph Methods

Some malware families utilise computer networks in a way that makes them stand out from the regular network traffic. Network malware analysis is very similar to static and dynamic is this regard. Signature-based detection is by far the most efficient approach to detect malware, even in network malware analysis. However, malware polymorphism hinders signature-based network analysis in the same vein, e.g. the malware can reconfigure itself to connect to different hosts periodically. Such behaviour is known for Domain Generation Algorithm (DGA) utilising malware.

Nari et al. [28] analyse network flows to extract information regarding network protocols used by a network host. The network hosts are represented by a node,

and the source host is connected to the target host by a directional edge. The graphs are not compared directly. Features are extracted first, such as graph size, average node degree, maximum node degree, root out-degree, and the number of nodes using a specific protocol. The extracted features are then used to train a decision tree model.

Besides this source, there is not much research on the application of graph structure for network-based behavioural malware classification.

4.3 Graph-based Classification

This section proposes a classification model based on graph similarity, inspired by concepts used in the static and dynamic analysis. As proposed in the source [27], a behaviour graph represents the behaviour of a malware sample, and for the malware samples in the same family, a combination of individual behavioural graphs represents the behaviour of the whole family. This can be viewed as a simplification of clustering – the family graph is essentially a cluster centroid, so only the distance to the centroid needs to be measured to determine the cluster for a new sample. For example, in k-nn classification, the distance to all samples needs to be measured to determine the nearest neighbours, which is significantly more computationally and time intensive.

The idea of a cluster centroid can be used generally as a basis of a classifier, which can consist of one or multiple such centroids. When working with graphs, a centroid graph, also called a reference graph, can be built to represent a behavioural class. Graph with an unknown class can be assigned a class by measuring its graph similarity to the reference, utilizing an adjustable decision function based on utility/loss or similarity threshold. Extension to multi-class classification is straightforward by creating multiple reference graphs for each class.

4.3.1 Terminology: Events and Transactions

An *entity* u is an object for which a behaviour can be observed, e.g. an executable in dynamic analysis, or a *user* in network analysis. High-level actions carried out by the user are called *events* e. Definition of events is domain-specific. Each event represents a unique behaviour presented by the user. Events can be defined with varying granularity, meaning a single event generally does not necessarily represent malicious behaviour. The succession of events, or a combination of events is what is essential. The act of an event causing the other to happen or two events happening at the same time means there is an *event relation* e.

Transactions $t = \{\mathcal{E}, R, l, u, \tau\}$ are made up of events and event relations produced by observing the entity u during a time period τ, where $\mathcal{E}$ is a set of events of entity u. The event relations contained within a set of relations $R = \{r(e) = e' | e, e' \in \mathcal{E}\}$. Additionally, transactions are labelled by a class label l.

In the context of dynamic analysis, executable to be analysed is an entity. Executing the executable is performing a transaction. The execution causes a chain of function calls. Each unique function call represents an event. A function

calling the other one is captured as an event relation. Transaction label originates from an anti-malware software, which if the executable is detected to be malicious either assigns a positive label in a binary classification case, or a malware family-specific label in case of multi-class.

4.3.2 From Transactions to Graphs

Concepts defined in the previous paragraphs are used as building blocks of a graph structure. Creation of a graph $G = \{V, E\}$ from a transaction $t \in T$, where T is a set of transactions, in done the following steps.

1. Use events $\mathcal{E}$ as graph nodes V: $V = \mathcal{E}$
2. Use event relations R as graph edges E: $E = R$

In case there are event relations $r(u) = v$ and $p(v) = u$ for $p, r \in R$ and events $u, v \in \mathcal{E}$, the edge between the nodes of events u and v is bidirectional.

Defining how to construct a graph from one transaction allows to define graph construction for a set of transactions. One graph per transaction in a set is constructed, and the graphs are unified using graph union to create a single graph. For example a *user graph* G_k can be created from a set of transactions $T_k = \{t \in T | u = k\}$ for a user k, or a class *reference graph* G_c from a set of transactions $T_c = \{t \in T | l = c\}$ for class c. The idea of the reference graph is analogous to the malware family graph proposed by Park et al. in the source [27].

4.4 The Graph Classifier

The graph classifier is a supervised learning algorithm made to classify user transactions. The classification is done by measuring similarity of a user graph G_u to a reference graph G_c. The reference graph is built to represent a particular class c produced by a supervised labelling process.

The classifier operates in two phases: training phase and testing phase. The reference graph G_c is created in the training phase as a core of the classifier the training dataset D^{train}. The training phase continues by calculating edge weights for the reference graph based on edge relevance to the class label.

The testing phase consists of building user graphs from the testing dataset D^{test}, trimming each user graph and reference graph according to a trimming strategy, measuring the pairwise similarity between each user graph and reference graph, and evaluating the performance of the classifier. The whole process is illustrated on figure 4.2.

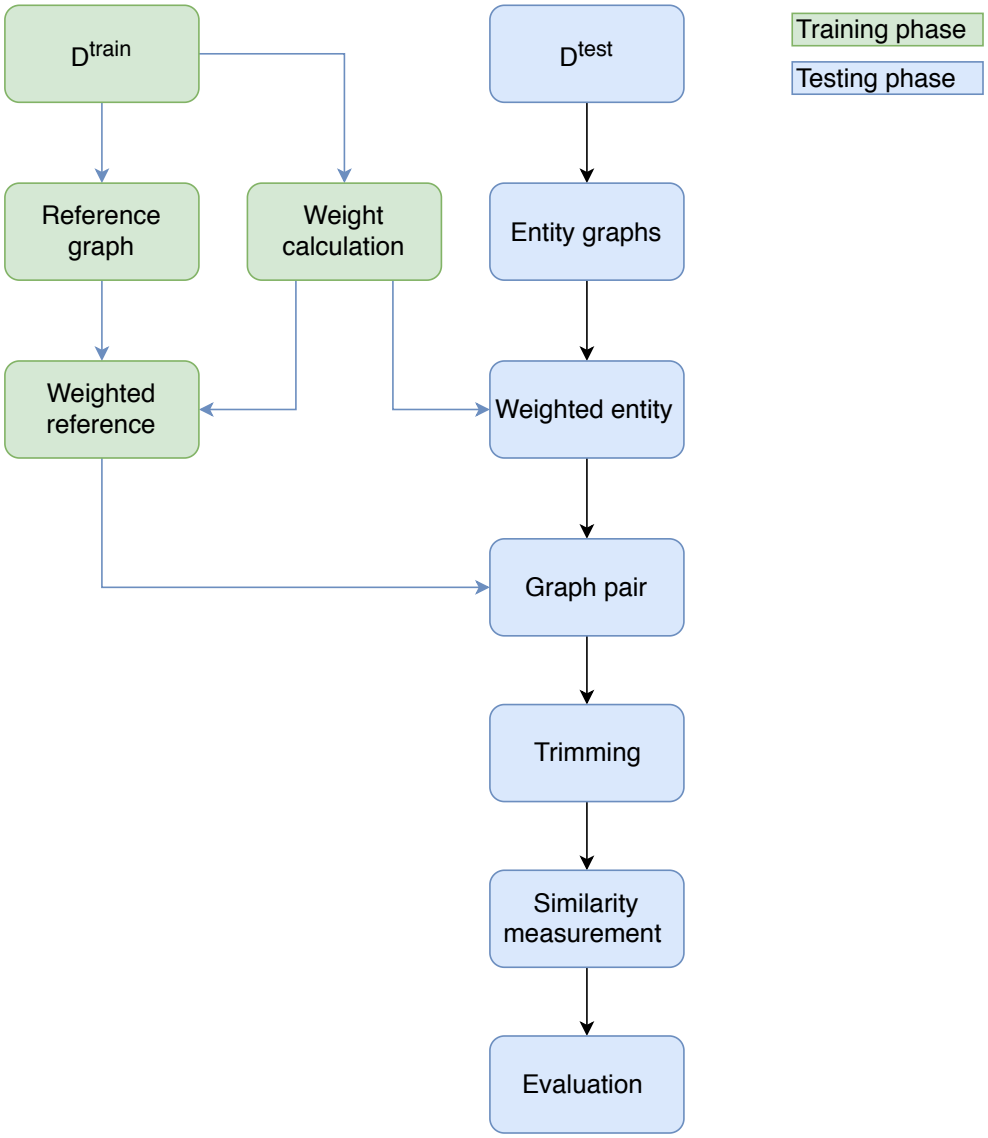


Figure 4.2. Graph classifier diagram.

4.4.1 Training Phase: Reference Graph

Construction of a reference graph is done in the following steps.

1. Filter D^{train} to keep transactions labelled c: $D_c^{train} = \{t \in D^{train} | l = c\}$
2. For each transaction $t_c \in D_c^{train}$, create a graph $G_{t,c}$.
3. Combine graphs $G_c = \bigcup G_{t,c}$ by performing a graph union.

The reference graph G_c is assumed to contain only relevant edges. The relevancy is achieved by graph trimming performed in the testing phase.

4.4.2 Training Phase: Weighting

Edge weight is used to determine importance of an edge, for example in case edge trimming is necessary. Edge weights are calculated from the training data. The primary component of the training data are transactions consisting of events. The events are defined to cover a broad range of actions an entity can carry out,

from benign to malicious. The granular definition of events means a single event may not be indicative of a class, so using the event importance of a sigle event to determine edge weight is not viable. A combination of two or more events is much more informative. Finding such combinations is a goal of associative rule mining, which is studied by Kopp et al. in [29].

The definition of a transaction includes a definition of event relation. Event relations encode pairwise event relation in a transaction and are ultimately used to build the graph structure as edges of the graph. The importance score of the event relations can be used straightforwardly as an edge weight. However, the event relation-based weights only encode importance of two-event combinations. Calculating weights for a combination of multiple events could be studied in the future work.

The calculation of the edge weights is done for each reference graph G_c separately. The labels l assigned to the transactions in D^{train} are converted to a binary case beforehand, meaning the transactions with $l = c$ are considered positive and the remaining labels are considered negative. The binary labels are required by the algorithms that later calculate the importance score.
The importance score can be computed in the following steps.

1. Transform D^{train} into $D^{train}_{u'}$, which consists of $t_{u'} = \bigcup\{t \in D^{train} | u = u', \tau = \tau'\}$ for time τ' and for each user u'.
2. Transform $D^{train}_{u'}$ into an event relation matrix X and class label vector y.
3. Calculate an importance score for each column of X according to label y.
4. Use the importance score as a score for corresponding event relation, therefore as an edge weight.

In the first step, a dataset $D^{train}_{u'}$ is created, containing transactions $t_{u'}$, which are a union of transactions $t \in D^{train}$ for each unique user u given time τ'. This is done to ensure the weights are calculated for event relations occurring for the user, since the user graph built from multiple transactions is what is classified, not single transactions.

In the second step, event relations of transactions $t_u \in D^{train}_{u'}$ are transformed into a feature vector x with label l. X is a (sparse) matrix containing feature vectors x in rows, and y is a label vector, containing a class label l for each row of the matrix X. Transforming the transactions into a sparse event relation matrix allows the usage of standard feature selection algorithms described in section 2.4.3. Those algorithms output a feature importance score for each feature, meaning each event relation is assigned a score usable as an edge weight for the corresponding edge in the graph.

4.4.3 Testing phase: User graph

The ultimate goal of the classifier is to assign class c to transactions of a user u', which is achieved by classifying the user graph $G_{u'}$ built from those transactions. The user graph can be built from one or multiple transactions produced by one user over time τ'. Compared to the reference graph, user graphs are usually much smaller in size.

User graph is constructed analogously to the reference graph:

1. Transform testing dataset D^{test} into D_U^{train} which consists of $T_{u'} = \{t \in D^{test} | u = u', \tau = \tau'\}\ \forall u' \in U$ where U is a set of unique users in transactions from D^{test} captured during time τ'.
2. For each set $T_{u'} \in D_U^{train}$ and for each transaction $t \in T_{u'}$ create a graph $G_{u',t}$ using events as nodes and event relations as edges.
3. For each user $u' \in U$, combine graphs $G_{u'} = \bigcup G_{u',t}$ by performing a graph union, producing one user graph $G_{u'}$ for each user.

   The class labels present in the transactions the user graph was built from are used as true labels for the evaluation.

4.4.4 Testing Phase: Graph Pairs and Trimming

In this stage, *graph trimming* is applied to remove irrelevant edges based on edge weights. First, pairs of G_u and G_c are created so special trimming methods which trim G_c based on characteristics G_u can be used. Each user graph is paired with each reference graph. The trimming is performed on a pairwise basis.

There are multiple approaches to graph trimming. Based on the effect of trimming, approaches are divided into two categories:

- *top k* – top k weighted edges in a graph are kept, the rest is trimmed. Alternatively, a threshold is introduced. Edges weighted below the threshold are trimmed.
- *top k%* – top $k\%$ edges are kept, main difference from top k is graph size dependence. The number of trimmed edges is proportional to the graph size.

Resulting graphs contain only the edges kept after trimming, reducing sizes and density of individual graphs by removing irrelevant edges. Graph density is the first problem graph trimming aims to solve.

The second problem emerges when two graphs, user graphs, and reference graph, are to be measured for their similarity. The size difference between user graphs results in larger entity graphs being more similar to the reference by default in case of Graph Edit Distance. This problem is present in the dataset used in experiments.

The problem can be partially solved by trimming the user graph first and keeping top k edges for the reference, where k is the user graph size after trimming. In other words, the size of the reference graph is reduced to match the size of the user graph. This can, however, result in the user graph being compared to a wrong part of the reference, as the part of the reference the user was matching could have been trimmed, falsely decreasing the similarity.

Following trimming method categories are created with the assumption that the reference graph is larger compared to user graph. If this is not the case, the larger graph is the one being reduced to match the smaller one.

- *Equal trimming* – Both user graph and reference graph are trimmed by the same transformation.

- *Adjust-to-user trimming* – User graph is trimmed first and the reference graph is top k trimmed to match the user graph size k.

Top k trimming is the simplest equal trimming method. There is a fixed parameter k. Both graphs in the pair are trimmed to keep at most top k edges by weight.

Weight threshold trimming sets a weight threshold for all graphs. Each graph has its edges below the weight threshold removed. The size of the user graphs varies from user to user, which might cause problems with distance comparability.

Adjust-to-user: top $k\%$ is based on trimming the reference graph to the size of the user graph. First, each user graph is trimmed to keep top $k\%$ of the edges. The trimmed user graph is then matched with a reference graph which is trimmed to match the size of the user graph.

Adjust-to-user: weight threshold works almost the same as the previous method, but instead of top $k\%$ trimming, weight threshold trimming is performed on the entity graph, and the reference is trimmed to match the entity in size. Doing this should eliminate the comparability problem of the weight threshold trimming.

Adjust-to-user-minmax: top $k\%$ extends the adjust to user: top $k\%$ trimming by setting minimal and maximal weight threshold for the reference graph, based on minimal a maximal weight in the entity graph after performing top $k\%$ trimming. This is inspired by a distance measure called maximal common subgraph.

Average user reference: weight threshold trims user graphs according to a weight threshold t, measures an average number of edges across resulting user graphs and trims the reference to match the average size.

4.4.5 Testing phase: Graph Similarity Measurement

The simplified Graph Edit Distance introduced in the end of the section 3.2 is used to calculate the distance between the graphs. Pure distance d_{GED*} outputted from the algorithm is not used. First, it is normalised to counteract the influence of graph size bias by performing user size normalisation, and converted into similarity S:

$$S(G_u, G_c) = 1 - \frac{d_{GED*}(G_u, G_c)}{|G_u|} \tag{4.1}$$

where $|G_u|$ denotes the size of the user graph G_u in terms of edge count, which in case of a weighted graph means the sum of edge weights. Performing this normalisation redefines the similarity as a ratio of the user graph edges matching the reference edges out of all user edges. Reference graph is considered pure, i.e., all edges of reference graph are relevant, ensured by prior application of weight-based trimming. The user graph is therefore penalised for both missing and redundant edges compared to the reference.

4.4.6 Testing phase: Evaluation

The classification performance is visualised on a class-separation histogram and a precision-recall curve per class. In case there are multiple reference graphs, the similarity of each user graph to each reference graph is evaluated, assigning a class

of the reference graph to which the entity graph is the most similar. A similarity threshold can be set to assign a label only in case when the similarity exceeds the threshold.

Along with the classification performance, visual comprehensibility of the reference graph needs to be evaluated. Graph density defined in section 3.4 is used to represent the comprehensibility.

Area under the precision-recall curve is used to evaluate the classification performance, and graph density d to evaluate the comprehensibility. When selecting the best set of classifier's parameters, one metric is necessary to find which setting performs better. However, neither AUC_{PR} nor graph density alone is sufficient.

First, the AUC_{PR} needs to be adjusted to take the graphs discarded by trimming into account. The adjustment consists of multiplying the AUC_{PR} by the ratio of the number of remaining graphs to the number of graphs before trimming. The resulting value is called *Absolute* AUC_{PR} (AUC^a_{PR}).

For the graph classifier, a combination of AUC^a_{PR} and graph density d is used to score the result of the classification.

$$s = w_1 AUC^a_{PR} - w_2 d \tag{4.2}$$

where $w_1 = w_2 = 1$ are weights, AUC^a_{PR} is a area under PR curve and d is the graph density as defined in section 3.4. Density d is selected instead of d_l mainly because d is a number between 0 and 1 so it can be used to represent the graph density without modifications. However, d_l could be considered in the future work, since it corresponds to the same visual density for different graph sizes unlike d. The d_l would require tuning before it could be used for evaluation. The pros and cons of using d_l or d to represent visual graph density are discussed in detail by the authors of [17].

The weights w_1 and w_2 can be adjusted to change the influence of the metrics on the overall score.

The visual evaluation of the graphs can follow, aided by colouring the graph nodes based on a modularity partition they fall into. This allows the analyst to see groups of events that are related as sharing the same colour and also to see the corresponding parts of the user graph that match the reference.

Chapter 5
Problem Outline and Implementation

This chapter describes how can the graph classifier be used in the domain of network traffic analysis, namely for processing network event-based data. The Cognitive malware detection system is described as a source of the event-based data used to train the graph classifier. The implementation of the graph classifier used to conduct the experiments in the next chapter is described in the last section.

5.1 Cognitive – Network-based Malware Detection

Cognitive Intelligence is a product of Cisco Systems aimed at enterprise use cases, focused mainly on network-based detection of malware and malicious communication. Cognitive processes input telemetry to discern normal network behaviour from malicious, utilising multiple layers of preprocessing, anomaly detection and classification. In case a malicious activity is detected, Cognitive creates an incident, based on which a network administrator can act to deal with the infection.

Cognitive can process different types of network communication captured, categorised into two main groups: web proxy logs and NetFlows. The web proxy logs contain information regarding the outgoing and incoming web-based traffic, i.e. communication of the local network hosts with hosts located in external networks. The collected input telemetry is proxy log-based (webflow) or NetFlow based.

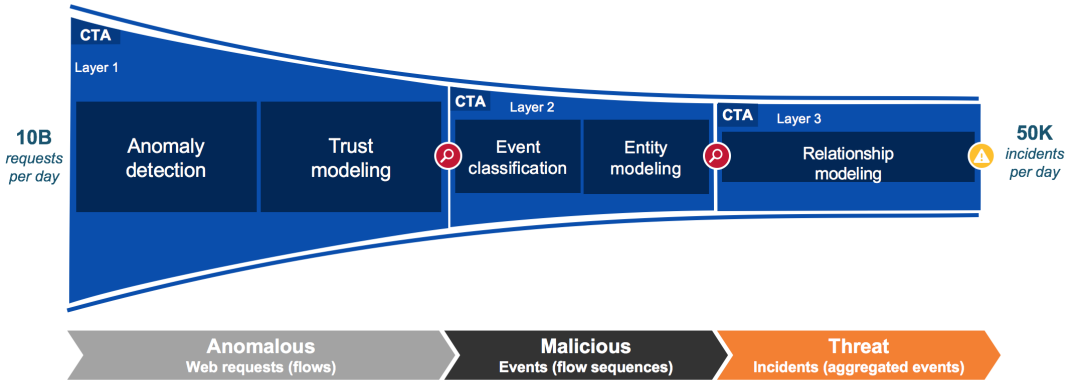


Figure 5.1. Diagram of Cognitive data processing pipeline. Data in this thesis originates from Event classification in the second layer. Source: [30].

The processes used by Cognitive to process the input data are visualised on picture 5.1. First, the data is analysed by the Anomaly detection layer. This

layer is made to pre-process the input telemetry by filtering samples that are anomalous from amounts of data exceeding 10 billion samples per day. The filtered samples are enriched with additional information from the Trust modelling and the anomaly detectors.

The second layer starts with Event classification, where the output of the anomaly detection is processed to create network events by a set of classifiers. Each event can be viewed as a type of a weak label, the presence of which is not sufficient evidence of an infection, even though the event usually stands for highly anomalous activity. The combination of these events can be viewed as a high-level behavioural indicator, which is suitable to represent a behavioural profile of a malware family. Cognitive uses approximately 350 events divided into four main categories:

- **Signature-Based** – events produced by behavioural signatures made by a domain expert
- **Classifier-Based** – events created by classifiers trained on historical data
- **Anomaly-Based** – events produced from the output of anomaly detection, consisting of more than 70 detectors
- **Contextual** – events describing activities of a host, such as a file download or access to raw IP

The Event classification outputs a set of events created by processing a set of flows collected for a network host during 24 hours. This set of events is referred to as a transaction.

After the Event classification step, further classification is performed, the result of which are class labels assigned to the transaction. Cognitive defines several high-level class labels, such as *Malicious Content Distribution, Information Stealer* or *Trojan*. These classes are not meant to represent specific malware families. Instead, they may stand for multiple malware families, each having behaviour characteristic for the class, i.e. class *Malicious Content Distribution* covers malware and malware families distributing potentially malicious adware.

5.2 Problem Outline

As stated previously, sets of events originating from processed network communication logs can be used to create a behavioural profile for a malware family. One of the approaches of creating the behavioural profile is known as rule mining, described in detail and evaluated on an event-based dataset by Kopp et al. in [29]. When applied to the event-based dataset produced by Cognitive, rule mining aims to find sets of events that occur together frequently in transactions labelled as malicious. The resulting sets of events can be used to describe malicious behaviour.

The graph structure, as proposed in the previous chapter of this thesis, can be used to model the behaviour of a malware family such as adware by processing the labelled event-based transactions. Given that a graph is built for multiple transactions of a malware family, there should be parts of the graph consisting of events which together are indicative of the behaviour presented by the malware

family. The resulting graph can be visualised, the essential parts emphasised by edge weight and coloured by graph modularity partition. Furthermore, the malware family graph can be used to classify unlabelled graphs built for network hosts by calculating the graph similarity.

In the terminology of the graph classifier, the output of Cognitive utilized in the graph classifier consists of transactions $t = \{\mathcal{E}, R, l, u, \tau\}$, where τ is the day when the event set $\mathcal{E}$ of the user u was captured with class label l. Event relations R need to be artificially created, because there are no relations between events in the provided dataset. The relations are therefore created for each transaction t using the event set $\mathcal{E}$ by adding a relation $r(e) = e'$ for each $e, e' \in \mathcal{E}$ into R, which means an event relation is added for each pair of events that occurred in the transaction.

However, the unlabelled transactions make up the largest portion of the dataset, as noted in table 6.1. There are several reasons as for why the label is missing, such as missing data or a new infection, which the classifiers are not yet trained to detect.

Among the labelled transactions, the majority are labelled with *Malicious Content Distribution* (MCD), *Malicious Advertising* (MA) and *Ad Injector*. These three labels represent a low severity infection such as adware or a fake search engine. The rest of the labels are severe infections.

Among the events defined by Cognitive, there is a set of more severe events. For example, event `ECWDGA1` means there was communication with a generated domain. Such activity is seldom deliberately done by a human user and is known for many different types of malware. The presence of the DGA event warrants a review, which needs to be done as soon as possible in case the infection is severe.

The unlabelled DGA-containing transactions are particularly interesting in this case. Assuming the distribution of infections in similar in labelled and unlabelled transactions, a large part of the unlabelled transactions is caused by adware. Filtering the unlabelled transactions by removing the ones most likely caused adware means transactions that remain fall into two categories. Either they are caused by a severe infection, or caused by an entirely new behaviour profile worth investigating. The review of remaining transactions should be prioritised in both cases. There is also a possibility that some of them are a false alarm, but the preprocessing the transactions went through along with the presence of DGA should mean the false alarm is unlikely.

Therefore, the goal is to identify and remove adware transactions from the unlabelled transaction set, so the malware analyst can focus on potentially more severe cases.

5.3 Implementation

Implementation of the graph classifier and experiments were done utilising the research infrastructure of Cognitive Intelligence within AWS. Code of the implementation was written in Python 3.7 in conjunction with PySpark for distributed parallel computing, Scikit-learn for implementation of machine learning tooling,

and NetworkX for implementation of the graph structure. Gephi was used for visualisation of resulting graphs. Development was done mainly using Jupyter notebook web interface.

5.3.1 Python

Python[1] is currently one of the most popular programming languages in the world due to its ease of use and a wide array of libraries. Python is known for its whitespace indentation to separate blocks of code and naming philosophy which makes Python code easy to read. Memory management is done with garbage collection, which further simplifies programming in this language. It is an interpreted language with interpreters available for many operating systems.

Since Python is an interpreted language with garbage collection, it is often associated with lower performance compared to languages such as C or Java. However, there are many libraries, notably NumPy and SciPy, which use vectorized implementations of functions written in C to make the computation very efficient. Because of this, Python is also very popular in the fields of data science and machine learning with a well-known library called Scikit-learn.

In this thesis, Python version 3.7 was used to implement the graph classifier and to carry out experiments. The most important libraries are described next.

5.3.2 Jupyter Notebook

Jupyter Notebook[2] is a web-based open-source web application made for interactive programming, data science, data visualisation, text annotation with Markdown support, Latex-like math notation and cell-based execution. Jupyter Notebook currently supports languages like Python, Julia, Java and JavaScript, with full kernel support listed in [31].

Jupyter Notebook was used in this thesis in conjunction with Python kernel to run the experiments and visualise the results. It was used mainly due to its cell-based execution, which allows to pre-compute computationally intensive tasks and work gradually with the result without re-running the computation.

5.3.3 SciPy

SciPy[3] is a Python ecosystem consisting of open-source libraries with a focus on mathematics. It consists of libraries like NumPy, Matplotlib, IPython and SciPy library. NumPy implements fast and efficient linear algebra operations and vectorised computation implemented for N-dimensional array objects.

Implementation in this thesis utilises a sparse matrix structure provided by SciPy sparse package, Matplotlib for plotting.

5.3.4 Scikit-learn

Scikit-learn[4] is a machine learning library implemented for use in Python, providing an implementation of commonly used machine learning algorithms for classi-

[1] https://www.python.org/about/
[2] https://jupyter.org/
[3] https://www.scipy.org
[4] https://scikit-learn.org/stable/

fication, regression and clustering, along with tooling for evaluation and visualisation. Scikit-learn is built supporting SciPy and NumPy libraries.

This thesis uses an implementation of random forest and k-means algorithms, cross-validation and PR-curve visualisation provided by Scikit-learn.

5.3.5 PySpark

Apache Spark[1] is a large-scale data processing engine. Spark provides high-level APIs in Scala, Java, Python and R, Spark SQL and Dataframes for data querying and analysis, MLlib package for machine learning and GraphX for large-scale graph processing. Spark utilizes Apache Hadoop to provide scalable distributed computing. PySpark is a Python implementation of Spark API running on a Java Virtual Machine.

PySpark was used in conjunction with Hadoop to access the data on AWS S3 and to perform parallel computation where applicable. It makes the implementation scalable for bigger volumes of data and runnable on computer clusters.

5.3.6 NetworkX

NetworkX[2] is a Python package for the creation and manipulation of graph-based networks. Graphs are implemented as objects with extensive API. Nodes and edges of graphs can represent even structured data such as images or XML records, which makes this implementation very flexible.

NetworkX is used to operate with graph structure and save and load graphs in a format supported by Gephi.

5.3.7 Gephi

Gephi is an open-source graph visualisation software. It implements graph layout algorithms along with comprehensive tools for graph formatting, editing, and filtering. Gephi is utilised to layout and visualise graphs created by graph classifier.

5.3.8 Amazon Web Services

Amazon Web Services (AWS) is a cloud-based computing platform offering services such as virtual servers (EC2), distributed computing clusters (EMR), web-based storage of big data (S3), database, developer tools, and deployment among many others. Provided services are paid based on usage time, redundancy, the volume of stored and transferred data, and availability. These services are run on infrastructure located in server farms around the world.

For experiments in this thesis, the computation was done on a `m4.4xlarge` EC2 general-purpose virtual machine instance with 2.4 GHz Intel Xeon E5-2676 v3 processor, with 16 CPU cores, 64 GB of memory, and running CentOS, a Linux-based operating system. S3 was used to store the provided dataset and to store backups of developed code.

[1] `https://spark.apache.org`
[2] `https://networkx.github.io`

5.3.9 Overall Implementation

Figure 5.2 shows how above-described software interconnects for implementation of the graph classifier. A virtual machine instance in EC2 serves as a Jupyter notebook server, running an interactive Python session. Inside of this session, Spark with Hadoop is initialised. Using Spark RDD API, data stored in S3 is accessed and loaded and processed utilising parallel data computation capabilities of RDD. The web-based interface of the Jupyter notebook server is accessed remotely through an SSH tunnel.

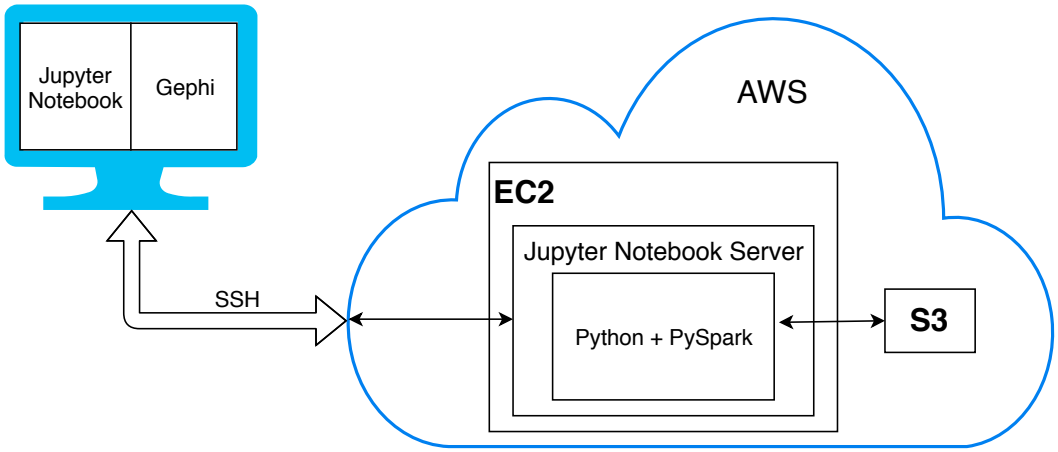


Figure 5.2. Diagram of the implementation's part interconnection

Chapter 6
Experiments

This chapter consists of dataset analysis and experiments conducted to improve the initial performance of the graph classifier measured in the second section. The best performing weighting method is selected and used in experiments with graph trimming. Afterwards, two graph pairs are visualized and analysed. The following section consist of alternative approach to build the reference graph. The last two sections contain comparison of classification performance between random forest and the graph classifier.

6.1 Dataset Analysis

The dataset used in experiments is a network security dataset produced by Cognitive by the process described in the previous chapter. It consists of flow sequences with a set of events and a cluster label, captured during January 2019. It contains 6.5 billion flow sequences, converted into day-long transactions for each user.

The dataset is split into two parts, training data and testing data. Training data consists of data captured during the first three weeks of January, and the testing dataset contains data for the remaining days of the month.

The training dataset is used to build the reference graphs and calculate the edge weights. The testing dataset is used to build user graphs. It is first filtered to only include transactions containing event `ECWDGA1`, because this event means there was a communication with a generated domain. Such activity is rarely carried out deliberately by a human user. DGA domains are known to be utilised by many different types of malware, visible in table 6.1.

class	train	train DGA	test	test DGA
no label	27 566 264	37 337	10 422 678	15 688
Adware	33 926	12 824	13 113	5 434
Ad Injector	100 839	1 036	37 772	444
Info Stealer	3 708	484	1 319	174
Ransomware	187	186	98	98
Malware Distr.	164	85	65	43
Trojan	162	23	66	8

Table 6.1. Number of class labels in the transactions of the training dataset.

The amounts of labelled transactions containing communication to a DGA domain are in table 6.1. Notably, a more substantial portion of *Ad Injector* and

Information Stealer is filtered out by keeping transactions with DGA. On the other hand, almost no *Ransomware* is filtered out, which means it almost always uses DGA to connect.

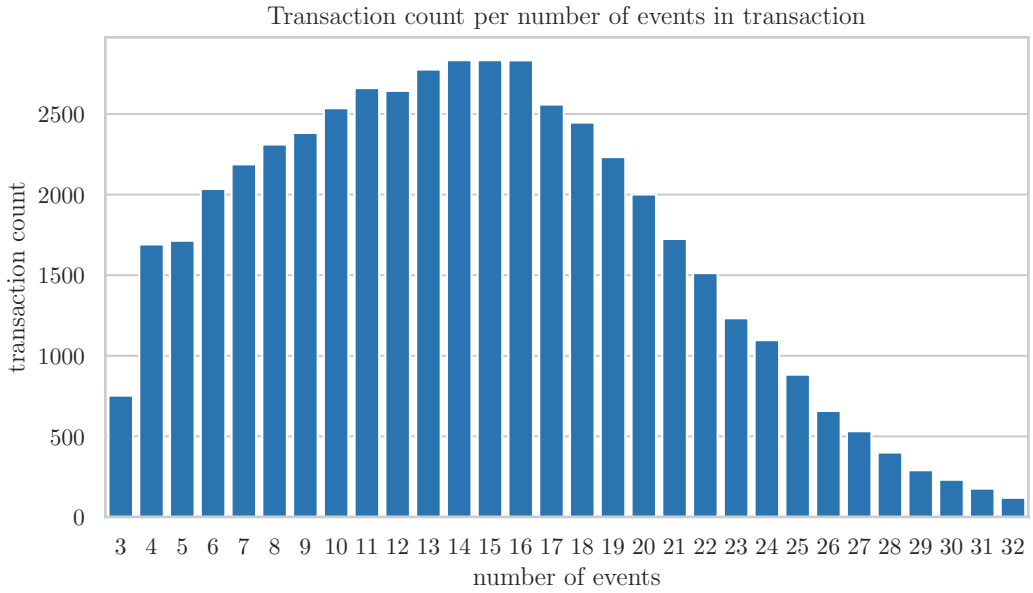


Figure 6.1. Distribution of event counts per transaction containing the DGA event.

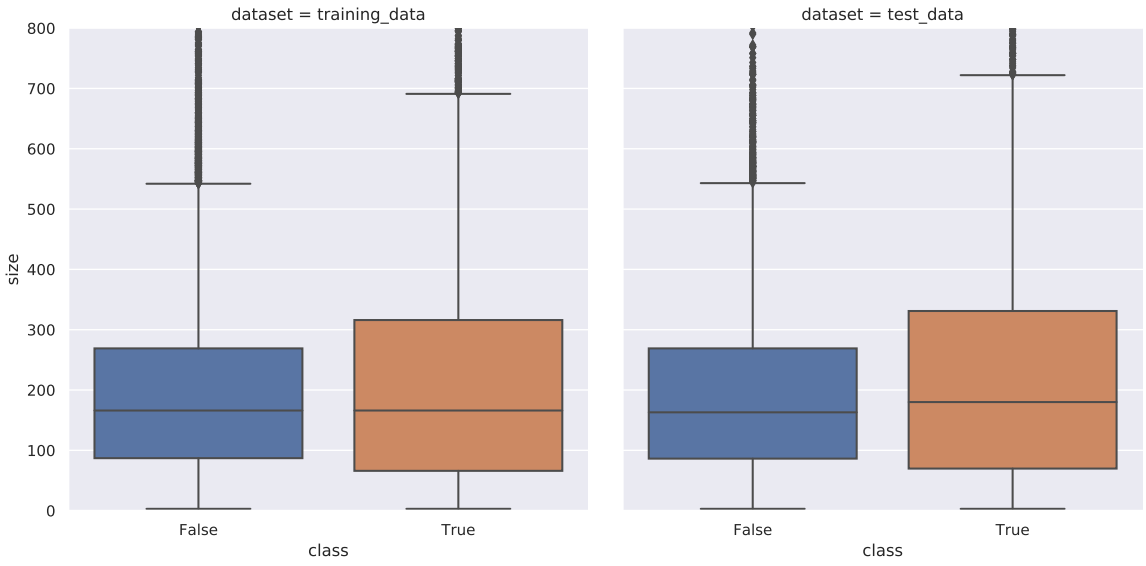


Figure 6.2. Distribution of graph sizes per class with adware as positive class.

A transaction can contain a subset of approximately 350 unique events. Usually, the transaction only consists of several events, which is visualised on figure 6.1. The low count of events per transaction means the dataset is rather sparse.

Next, the user graph size distribution should be worth investigating, mainly when it comes to the relation of user graph size of graphs with adware class label. Figure 6.2 shows there is almost no difference in the distribution of user graph sizes for positive and negative classes, which means the graph classifier should be

able to discern the adware from the rest regardless of the user graph size in order to be accurate.

6.2 Single Reference – Adware

The first experiment serves to set a performance baseline for the graph classifier. To begin with, a reference graph is built for adware. The graph is neither weighted nor trimmed to set a classification performance baseline. Figure 6.4 contains a visualization of the resulting reference graph, which is visually incomprehensible due to high edge density.

The testing phase consists of building user graphs using the training dataset, which is pre-filtered to consist of DGA-containing transactions. The user graphs are neither weighted nor trimmed.

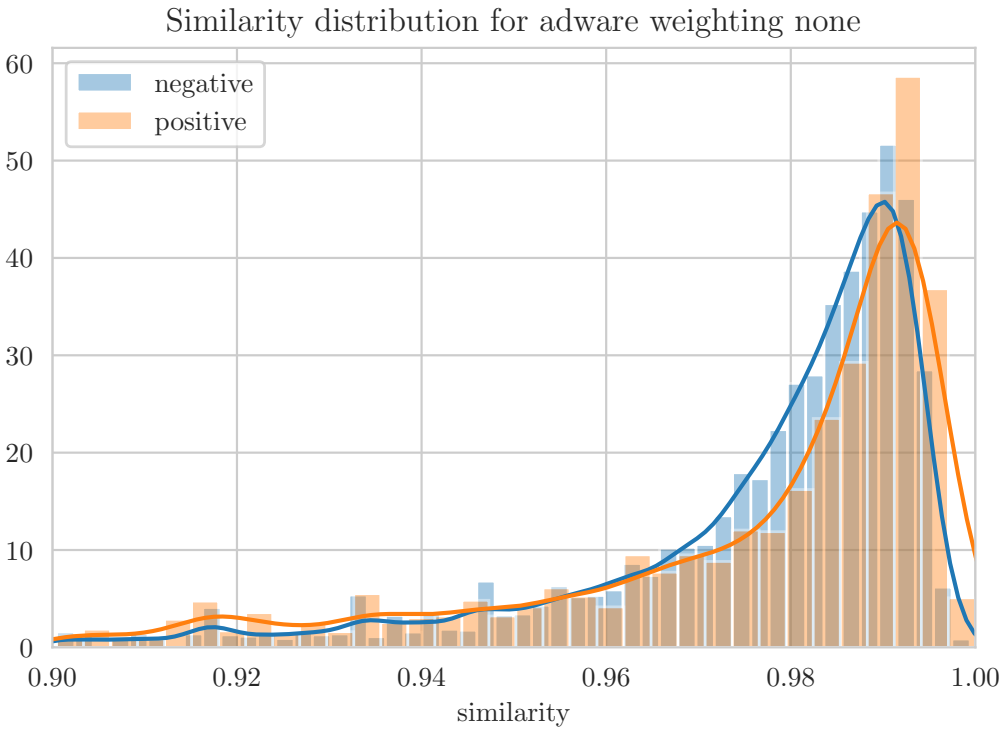


Figure 6.3. Similarity distribution of graph classifier with initial settings.

Each pair containing unweighted reference and user graph is evaluated for graph similarity. In terms of classification performance, the class separation histogram depicted in figure 6.3 shows the classes are almost entirely overlapping, which means they are not separable with the current setting. This is possibly caused by the fact that all edges are weighted equally – only the number of edges of the user graph matching the reference graph is used to determine the distance by the Graph Edit Distance. Edges missing in terms of edge importance are not accounted for, since there are no edge weights.

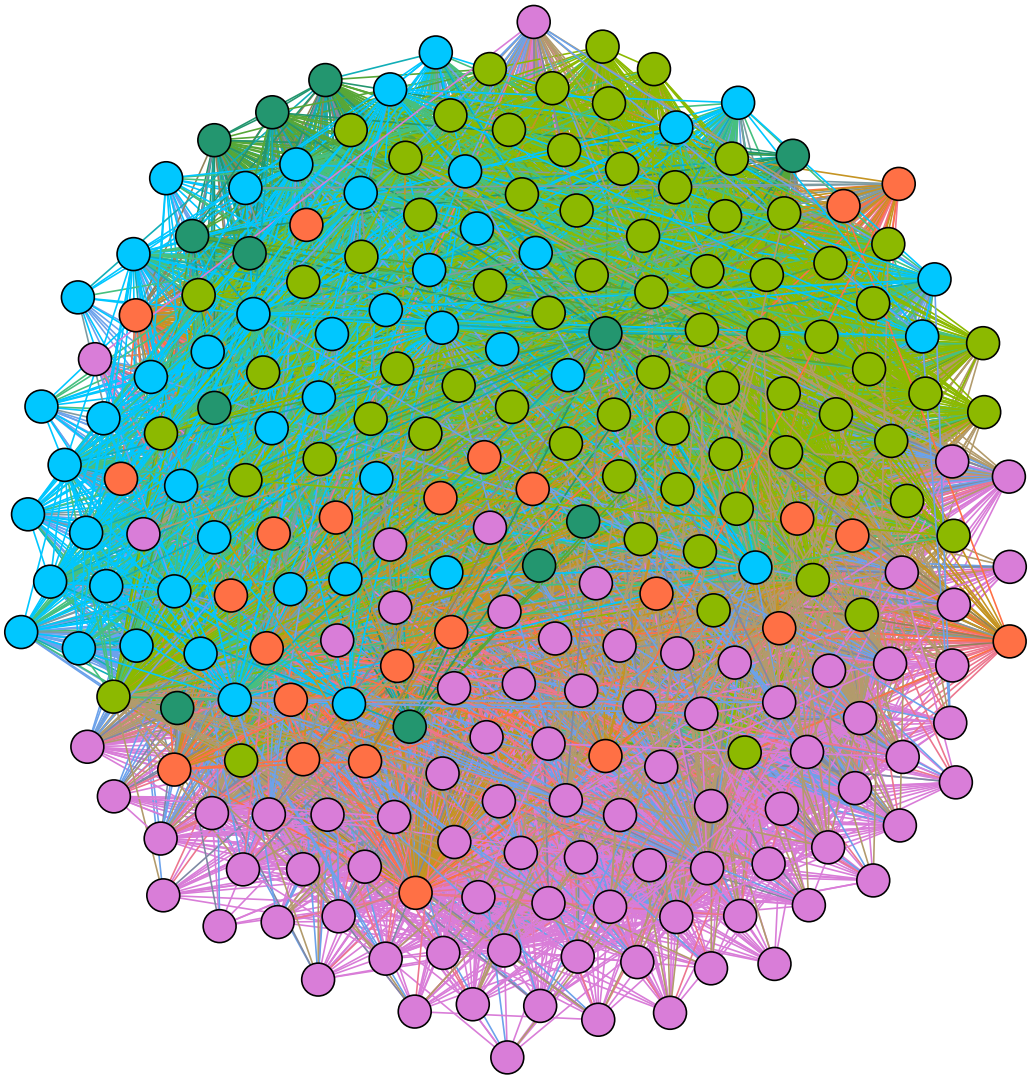


Figure 6.4. Reference graph built for adware without weighting and trimming.

6.3 Weighting

Weighting aims to resolve problems with overlapping classes and graph density by utilising edge weights in conjunction with edge weight-aware Graph Edit Distance. The tested weighting approaches are listed in table 6.2.

After building the reference graph, the edge weights are calculated as explained in section 4.4.2. These weights are used to weight the reference graph and later also to weight the user graphs.

Next, the user graphs are built and weighted, the pairwise similarity with the reference graph is calculated, and the classification performance is evaluated. Evaluation includes computing area under Precision-Recall curve (AUC_{PR}), area under PR curve penalized by the number of graphs removed after trimming as AUC^a_{PR}, which is computed as $AUC^a_{PR} = rAUC_{PR}$, where r is a ratio of remaining graphs compared to the number of graphs before trimming.

Evaluation also includes calculation of d as an average graph density of the user graphs, and d_l as an average number of edges per node of the user graphs. Resulting score is a combination of the AUC^a_{PR} and density d, as defined in section 4.4.6. Density d is used mainly for computation of the evaluation score, while d_l is easier to intuitively interpret, but not usable for score computation, because it is not directly suitable for score computation as described in section 3.4. As per source [17], values of d_l above 4 are considered too dense to be visually comprehensible. As a rule of thumb, $d_l < 2$ can be considered good density.

Table 6.2 contains the classification performance of the graph classifier with different weighting approaches without trimming. In terms of pure classification performance, the improvement of AUC_{PR} is only slight even for the best performing weighting method. However, there is a considerable drop in graph density compared to the unweighted graph, visible in the column d_l, where the density drops by almost a half. A significant amount of edges was removed after the introduction of the edge weights, caused by weighting methods valuing the irrelevant edges with zero weight.

	AUC_{PR}	d	d_l	Score
no weights	0.353	0.671	9.637	0.340
frequency	0.299	0.484	4.827	0.407
chi2	0.372	0.487	4.891	0.442
mutual information	0.320	0.487	4.891	0.416
Fisher Score	**0.374**	**0.487**	**4.891**	**0.443**

Table 6.2. Comparison of weighting methods.

The Fisher Score-based weighting performs the best both in terms of AUC_{PR} and graph density. Because of that, it will be used in the following experiments.

Looking at the density d_l, the weighting reduced the density considerably, caused by the fact that some of the edges were weighted as zero and such edges are removed by default. Still, having almost five edges per node means the weighted graph is not much more comprehensible when visualised compared to figure 6.4. Further reduction in graph density is necessary, which is achievable by performing the graph trimming.

6.4 Trimming

Because the improvement in classification performance after introducing weight was negligible and the graphs were still too dense, trimming is tested as a next step for improvement in classification performance and reduction of the graph density. The best performing weighting method, the Fisher Score is used for weighting.

Experiments with parameter tuning are done on a 10% subsample of the user graphs (1670 user graphs) to speed up the trimming, distance measurement and evaluation process. After selecting the best performing parameter for each method, the evaluation is done on the whole testing dataset.

6.4.1 Equal Trimming Approach

Section 4.4.4 introduced the distinction of the trimming methods by the approach of trimming. Trimming is performed for pairs of graphs, one reference and one user graph. The first approach of trimming is *equal trimming*, which consists of methods that trim both reference and user graph by applying the same transformation, resulting in two graphs with different sizes. Distance algorithms must take this into account not to skew the similarity by the size difference.

Two methods utilize the equal trimming approach. The first one is **constant trimming** with parameter k. This method is straightforward: top k weighted edges are kept for the user and reference graph. The influence of the value of k can be seen in table 6.3.

k	AUC_{PR}	AUC^{a}_{PR}	d	d_l	Score
2	0.375	0.374	0.374	0.259	0.499
6	0.345	0.341	0.270	0.660	0.535
8	**0.338**	**0.334**	**0.259**	**0.728**	**0.537**
10	0.310	0.307	0.254	0.793	0.526
16	0.273	0.269	0.254	0.983	0.507
30	0.272	0.268	0.280	1.415	0.494

Table 6.3. Constant trimming: parameter tuning.

The constant method does not discard many user graphs, which is reflected in a small difference between AUC_{PR} and the absolute AUC^{a}_{PR} values. Notably, the value of AUC_{PR} drops with increasing k. The drop probably means a small number of edges bring the most information while adding lower-weighted edges introduces noise.

The second method is **min. weight threshold trimming**. A weight threshold is set, and all edges weighted below the threshold are removed. This is done for both user and reference graph, resulting in graph pairs where the user and reference are generally not sized equally.

Parameter t is used to adjust the threshold. It determines the minimum weight used as a threshold similar to the top-k trimming. Weights for all possible edges given the set of all events are sorted in descending order, and t-th weight is used

t	AUC_{PR}	AUC^a_{PR}	d	d_l	Score
100	0.282	0.119	0.295	0.550	0.411
500	0.258	0.200	0.254	0.724	0.473
700	**0.265**	**0.223**	**0.245**	**0.807**	**0.489**
1100	0.262	0.234	0.261	1.077	0.486
1500	0.260	0.247	0.278	1.340	0.484
1900	0.270	0.264	0.299	1.612	0.482

Table 6.4. Min. threshold trimming: parameter tuning.

as a threshold. The influence of adjusting the value of t can be seen in the table 6.4.

The best performance is achieved by setting the value of t to 700. The AUC_{PR} is lower compared to the constant method. The drop in AUC^a_{PR} means the min. threshold method discards a larger amount of user graphs. The value of d_l remains in a reasonable range. The best score achieved by this method is, however, worse compared to the constant method. The significant amount of discarded graphs shows this method is not suitable for trimming, at least not without adjustment.

6.4.2 Adjust-to-user Trimming Approach

Next trimming approach to be tested is *adjust-to-user trimming* approach. The user graph is trimmed first with a transformation depending on the trimming method. The edges of the reference graph are removed, the lowest weighted first, until its size matches the size of the user graph.

Adjust-to-user trimming method keeps top $k\%$ of edges in the user graph or at least n edges as specified by the minimal edge count parameter n. As a result, both graphs have the same size. This can be seen as similar to the constant trimming, with the difference being that the number of edges kept also depends on the size of the user graph.

k	AUC_{PR}	AUC^a_{PR}	d	d_l	Score
0.01	0.370	0.369	0.371	0.530	0.498
0.03	0.350	0.349	0.316	0.562	0.516
0.04	**0.337**	**0.336**	**0.296**	**0.593**	**0.520**
0.05	0.321	0.320	0.284	0.628	0.518
0.40	0.272	0.271	0.330	2.065	0.470

Table 6.5. Adjust-to-user trimming: parameter k tuning.

The best score is achieved with $k = 0.04$ and $n = 8$ according to tables 6.5 and 6.6. Parameter k was tested first, with $n = 2$. After determining the best performing k, n was adjusted. According to the result, keeping more edges causes the classification performance to improve, and increasing the n brings further improvement, which starts to diminish around $n = 10$, probably due to uninformative edges being kept more often. The low value of k being the best performing

n	AUC_{PR}	AUC^a_{PR}	d	d_l	Score
2	0.337	0.336	0.296	0.573	0.520
6	0.325	0.317	0.262	0.671	0.527
8	**0.322**	**0.310**	**0.254**	**0.732**	**0.528**
10	0.299	0.284	0.249	0.795	0.517
14	0.273	0.248	0.246	0.923	0.501

Table 6.6. Adjust-to-user trimming: minimal edge count n tuning.

hints at the fact, that in the big user graphs, most of the edges are not useful for the classification.

Adjust-to-user: weight threshold is a combination of the two trimming methods. First, the user graph is trimmed in the same way as in the min. weight threshold method, but the reference is trimmed to the match user graph size.

t	AUC_{PR}	AUC^a_{PR}	d	d_l	Score
200	0.273	0.138	0.286	0.584	0.426
600	0.255	0.204	0.251	0.745	0.476
1000	**0.264**	**0.234**	**0.257**	**1.025**	**0.488**
1400	0.263	0.244	0.274	1.310	0.485
2000	0.271	0.265	0.268	1.657	0.480
4000	0.271	0.268	0.372	2.489	0.456

Table 6.7. Min. threshold adj-to-user trimming: parameter tuning.

Table 6.7 shows the best score is achieved for $t = 1000$, performing slightly better compared to min. threshold trimming, but still worse than adjust-to-user. The worse performance is mainly caused by discarding user graphs with all edges below the threshold.

k	AUC_{PR}	AUC^a_{PR}	d	d_l	Score
0.01	0.303	0.302	0.371	0.530	0.465
0.05	0.274	0.273	0.284	0.628	0.494
0.1	**0.271**	**0.270**	**0.260**	**0.807**	**0.505**
0.15	0.272	0.271	0.265	1.008	0.502
0.2	0.272	0.271	0.275	1.210	0.498
0.4	0.272	0.271	0.330	2.065	0.470
0.6	0.272	0.271	0.383	2.937	0.444

Table 6.8. Adjust-to-user-minmax trimming: parameter tuning.

Adjust-to-user-minmax trimming is an alternative to adj-to-user. The user is trimmed in the same way (top k%), but the reference is trimmed depending on two weight thresholds, min. weight and max. weight, set by values of weights in the user graph. Edges weighted above max. threshold are removed, as well as

edges below the min. threshold. Table 6.8 shows score of the graph classifier with changing value of k. Setting $k = 0.1$ results in the best performance.

Finally, **average user reference: weight threshold** method is tested. This method results in the same reference for all user graphs. The reference is top-k trimmed to contain the same number of edges as the user graphs contain on average. Table 6.9 shows the influence of changing threshold t.

t	AUC_{PR}	AUC^a_{PR}	d	d_l	Score
200	0.446	0.226	0.286	0.584	0.470
400	0.354	0.250	0.262	0.659	0.493
600	**0.308**	**0.247**	**0.251**	**0.745**	**0.498**
1000	0.269	0.238	0.257	1.025	0.490
1400	0.265	0.246	0.274	1.310	0.486
1800	0.270	0.263	0.297	1.580	0.482

Table 6.9. Min. threshold adj-to-average-user trimming: parameter tuning.

6.4.3 Comparison of the Best Performing Parameters

Table 6.10 provides a comparison of trimming methods with best-performing parameters, computed for the whole testing dataset without subsampling. Out of the tested trimming methods, the *constant method* scored the highest, with the classification performance visualised on a PR curve in figure 6.5. The *adjust-to-user* method reaches the second highest score.

The PR curve at the top right of the figure 6.5 shows that several positive graphs are classified with high precision, with few negatives before a large drop. These negatives are very likely new findings belonging to the adware class, which was one of the goals described in the problem outline.

trimming	AUC_{PR}	AUC^a_{PR}	d	d_l	Score
constant: $k = 8$	**0.342**	**0.338**	**0.261**	**0.731**	**0.538**
th: $t = 700$	0.267	0.225	0.247	0.821	0.489
adj $k = 0.04$ $n = 8$	0.328	0.314	0.255	0.739	0.529
adj-th: $t = 1000$	0.272	0.240	0.258	1.042	0.490
adj-minmax: $k = 0.1$	0.280	0.280	0.262	0.823	0.508
avg-user-ref: $t = 600$	0.296	0.237	0.252	0.757	0.492

Table 6.10. Comparison of trimming methods.

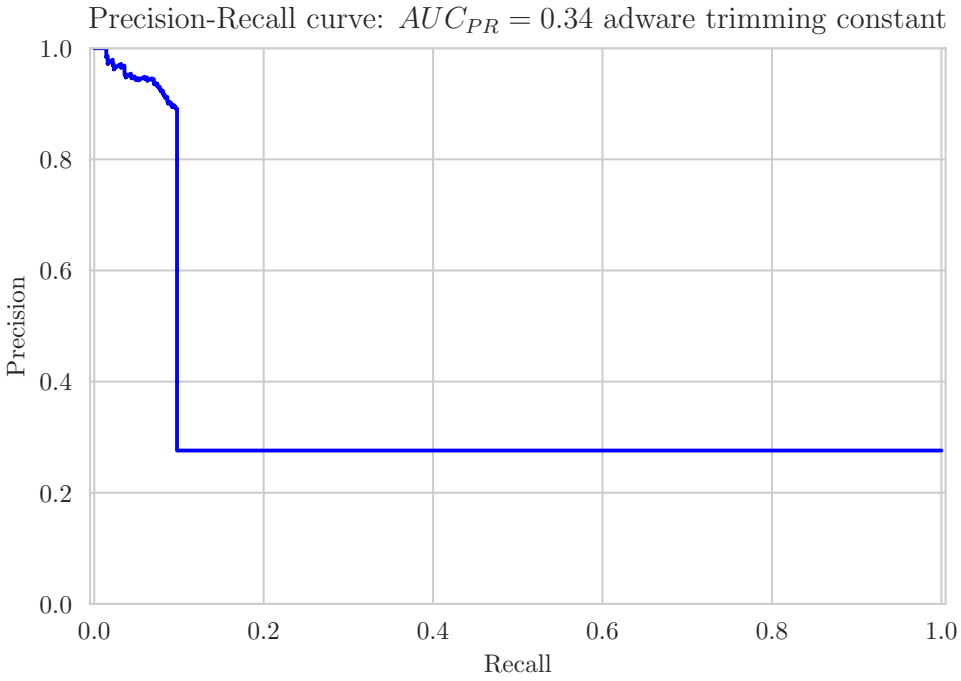


Figure 6.5. PR curve for *constant* trimming method with $k = 8$.

As for the similarity distribution, figure 6.6 shows that there is a slight improvement in the class separation compared to the separation of the initial setting in figure 6.3.

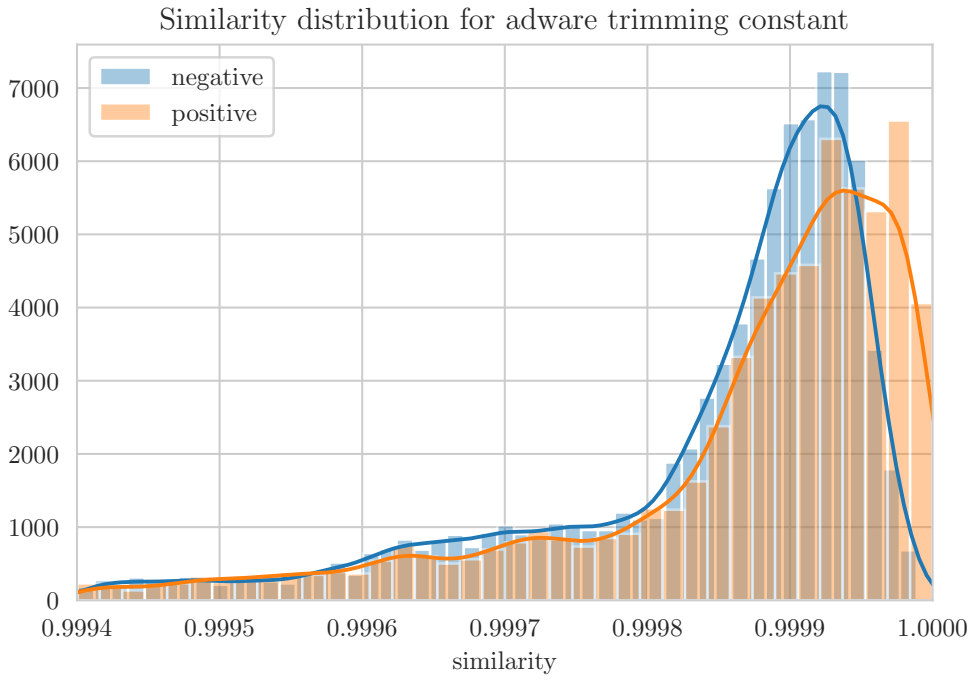


Figure 6.6. Similarity distribution of graph classifier with the best performing constant trimming setting $k = 8$

6.5 Analysis of the Trimmed Graphs

Graphs in figure 6.7 and figure 6.8 show a visualized pair of reference and user graph respectively after being processed by adjust-to-user set to parameters $k = 0.09$ $n = 6$. Both graphs are significantly smaller compared to the graph built with the initial setting depicted in figure 6.4.

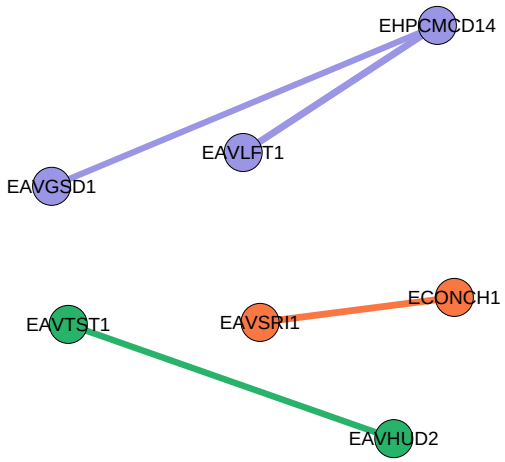


Figure 6.7. Visualization of a reference graph trimmed with adjust-to-user method with parameters $k = 0.09$ $n = 6$.

The thickness of edges represents associated edge weight. The reason the edges of the user are thin is that those edges are not relevant in the context of adware according to the edge weight originating from Fisher score. Furthermore, the user graph does not share even one node or edge with the reference, which results in zero similarity between the graphs. The graphs have become disconnected as a result of trimming.

Node colour represents modularity-based partition. For the user graph, the nodes are coloured according to the colour they have in the reference. If missing in the reference, their colour is left grey.

The reference graph in figure 6.7 is built for adware. It consists of three disconnected components. The biggest component contains nodes of three events: `EAVGSD1` meaning anomalous destination, `EHPCMCD14` meaning the communication was similar to *Malicious Content Distribution*-labeled malware. `EAVLFT1` stands for an abnormally long URL being used to access a webpage that was not accessed by the user before, meaning the access was most likely not done by the user but by an automated process.

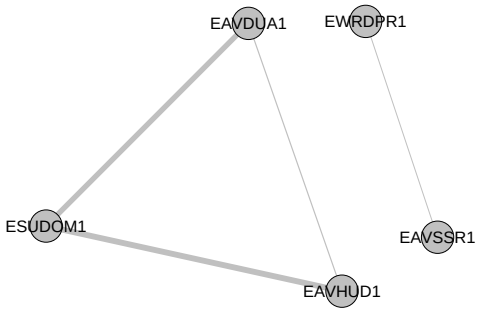


Figure 6.8. Visualization of a user graph trimmed with adjust-to-user with with parameters $k = 0.09$ $n = 6$.

The second connected component highlighted by orange colour consists of `EAVSRI1` meaning raw IP address was a target of the communication and `ECONCH1` means there was a connection check. A combination of these events, a connection check on a raw IP, shows a behaviour that is very unlikely to be caused by an average user. The third component is `EAVTST1` and `EAVHUD2`, meaning the user accessed the target at an unusual time, and the target was unusual for the user, respectively.

Due to being kept after adjust-to-user trimming, edges of the resulting reference are the highest weighted edges in the reference graph, which means they should connect events which are the most characteristic of adware. Disconnectedness might be caused by diversity in adware infection, meaning different types of adware trigger specific sets of events. In the case of 6.7, the components are not entirely convincing to be specific for adware.

The second pair of graphs is visualized on figures 6.9, and 6.10 is a result of the user graph having more edges before trimming, causing the reference to keeping more edges.

There are multiple coloured parts coloured by modularity in the reference graph on picture 6.9. The pale green part is centred on `ESUDOM1` which stands for a suspicious domain. It connects with `EAVDSU1` which means the user accessed many pages that are not normally accessed, `EAVDUA1` meaning an unexpected application based on user agent was detected and `EANEXF1` meaning an anomalous executable file was downloaded. This group of events can be interpreted as a download of a malicious executable from a suspicious domain by an application that was likely already compromised, due to the user agent inconsistency. In other words, this might mean the beginning of an adware infection.

Next, the pink partition centres on a very anomalous HTTP traffic describing event `EVAHTR2` through an encrypted connection `EENCON1`, a non-user activity `EAVDPC1`, `EDIMGS1` meaning a download of a picture and `EHPCADI03` standing for

a *Ad Injector*-like behaviour. This group of events might likely show an already established adware infection with multiple anomalous events and a download of one or multiple images, which are likely ads.

The pale blue partition consists of before mentioned `ECWDGA1` and `EAVLFT1`, meaning access to a generated domain with a long URL probably not caused by deliberate user activity. The part also includes a time of day inconsistency `EAVUEH1` and `EDATUP1` standing for data upload. Together, this can be interpreted as a data upload to a generated domain which the user did not access before, all during an unusual time for the user. Such behaviour is not exactly characteristic for adware, but more severe exfiltration infections. Furthermore, adware is often considered as a gateway for more severe infections.

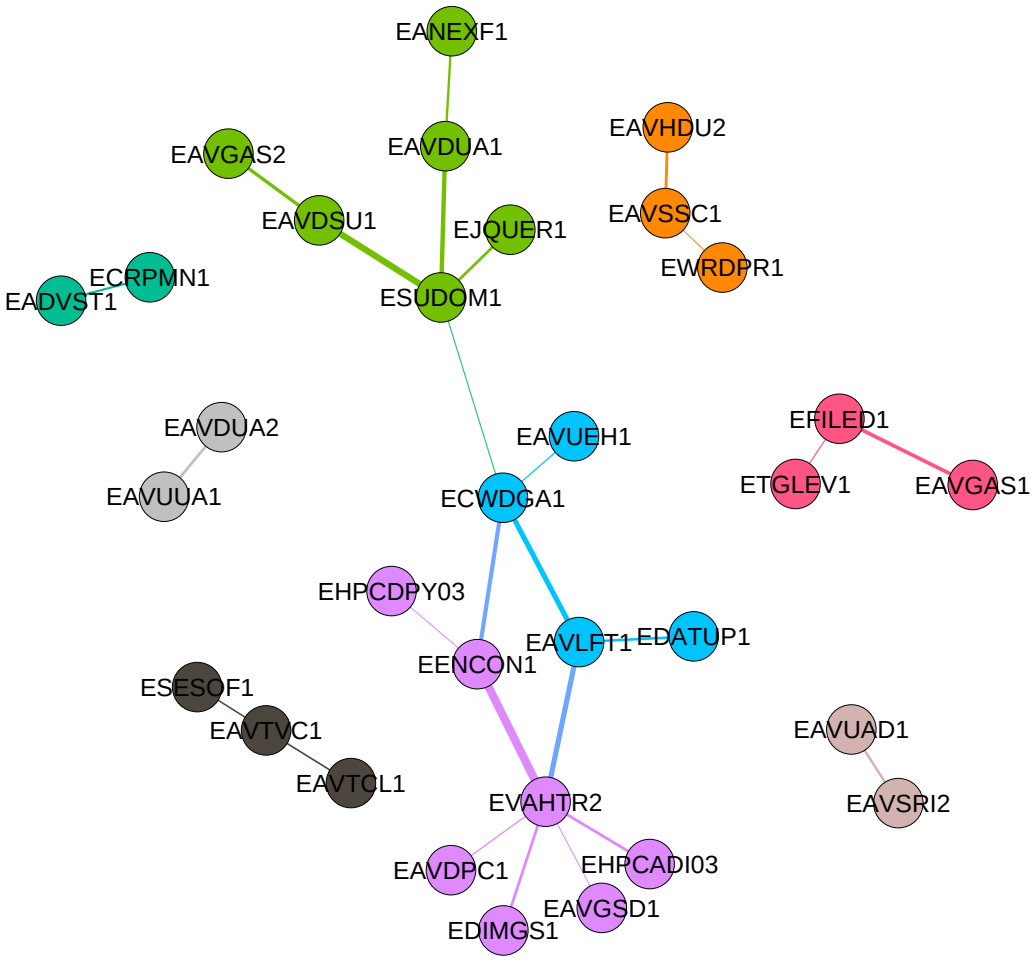


Figure 6.9. Visualization of a reference graph trimmed with adjust-to-user.

The red part contains EFILED1 connected to ETGLEV1 and EAVGAS1, meaning a malicious executable was downloaded based on Threat Grid intelligence[1] and that the destination was anomalous, respectively.

The orange partition consists of the usage of a self-signed certificate EAVSSC1 in conjunction with an anomalous destination EAVHDU2 and a Wordpress content management system EWRDPR1.

The set of events in the dark colour correspond to two TLS certificate verification inconsistency events EAVTCL1 and EAVTVC1 connected to a security software ESESOF1. This likely means adware compromises security software to use a fabricated certificate.

The last interesting part has teal colour and consists of two events. EAVDST1 means access to an advertisement site and ECRPMN1 stands for crypto mining. This might mean an ad was modified to mine cryptocurrency, which is rather common in recent time.

Above described reference graph partitions consist of activities that can be attributed to an adware infection. Such reference graph is a good representation and user graphs that are similar to it can safely be classified as adware.

As for the user graph on the picture 6.10, this time the user matches a big portion of the reference and is very likely infected by adware, if not by a more severe infection. Namely, the user matches a part of pale blue, pink, green and orange partitions of the reference. Additionally, the user graph contains events for raw IP access, suspicious HTTPS and other anomaly-based events.

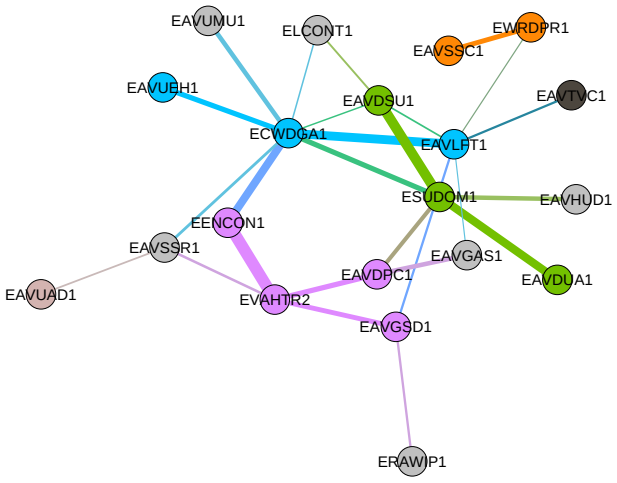


Figure 6.10. Visualization of a user graph trimmed with adjust-to-user.

[1] A threat intelligence product by Cisco. https://www.cisco.com/c/en/us/products/security/threat-grid/index.html

6.6 K-means Cluster Reference

Previous experiments improved the classification performance slightly, but there is still a lot room for improvement. The adware is still hard to separate from the rest of the data.

One of the possible causes of the inseparability might be there is too much variety in the adware class, i.e. the sets of events that make up transactions are not similar. Possibly, the labels assigned by Cognitive cover many types of adware, each presenting a different behaviour, therefore causing different sets of events to be triggered. If that is the case, using a clustering algorithm to separate the adware into clusters of similar transactions and building one reference graph per cluster might improve the classification performance considerably.

The training dataset is first transformed into vectors of event relations using the process described in section 4.4.2. Because this dataset consists of approximately 20000 features, a number which might pose a problem for Euclidean distance-based k-means due to the curse of dimensionality, a subset of top k features is selected before the clustering is performed. As an initial parameter of the k-means algorithm, k is set to 3, which means the algorithm will attempt to find three clusters of data within the training dataset.

top k features	AUC_{PR}	AUC^{a}_{PR}
20	0.333	0.330
70	0.331	0.327
150	0.332	0.329
300	**0.345**	**0.341**

Table 6.11. K-means-based reference: classification performance.

The clustering algorithm outputs a cluster label for each sample. Vectors of event relations are converted back to transactions so they can be used to train the graph classifier, and the cluster label from k-means is used as a class label for the transaction. Transactions with cluster labels are processed by the graph classifier, creating one reference graph per label. Graph weighting and trimming settings are set according to previous experiments.

An implementation of k-means from Scikit-learn is used. The rest of the parameters of k-means are set to default values.

Table 6.11 shows the classification performance of the graph classifier trained on data clustered by k-means with $k = 3$. The best performing *constant* method with its parameter $k_{constant} = 8$ was used to trim the graphs. In terms of AUC^{a}_{PR}, this method reaches slightly better performance compared to the best score of 0.342 as listed in table 6.10. However, preparation of the dataset and processing of the k-means took significantly more time compared to the single reference method.

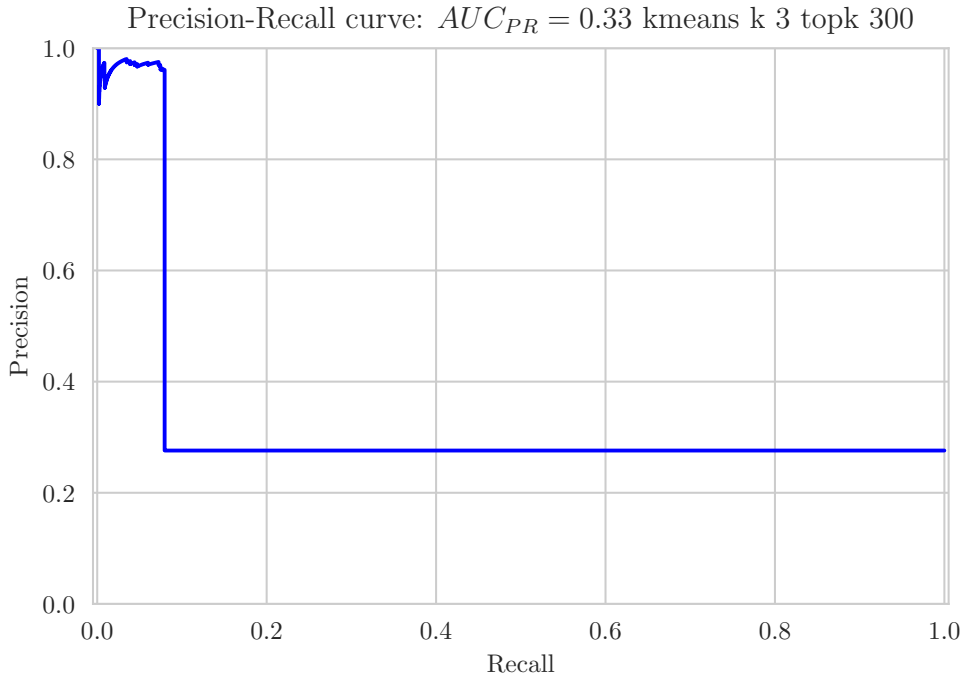


Figure 6.11. PR curve of the best performing k-means setting. $k = 3$, *constant* trimming $k_{constant} = 8$

6.7 Random Forest Classification Experiment

To compare how the graph classification method performs compared to a conventional classifier, the classification performance of random forest is measured. The random forest is trained on the same dataset used in the k-means experiments. Training is done for adware. The training data is a 10% sub-sample of the original dataset to speed up the training, which, without feature selection, is computationally expensive.

For the experiment, an implementation of a random forest was provided by Scikit-learn. A small adjustment to the default parameters of the random forest was made:

- number of estimators: 10
- max tree depth: 5
- max features: 0.7

These parameters were chosen mainly to reduce training time while keeping the estimators diverse and not too deep to prevent over-fitting.

The Fisher score from graph classification was used as a top-k feature selection. The cross-validation performance of different number of features is recorded in table 6.12. As expected, random forest performs the best without feature selection but takes significantly more time to train.

As for the performance on the testing dataset, the random forest reaches AUC_{PR} of 0.600 without feature selection.

Features	Fit time	Score time	tr. AUC_{PR}	test AUC_{PR}
top 10	11.4	1.4	0.422	0.420
top 20	11.1	1.4	0.436	0.434
top 40	11.3	1.3	0.484	0.483
top 60	11.4	1.3	0.485	0.484
top 100	11.4	1.3	0.505	0.504
all features	**339.6**	**3.6**	**0.532**	**0.529**

Table 6.12. Cross-validation score of random forest.

6.8 Experiment Result Summary

The random forest beats the highest $AUC_{PR} = 0.342$ reached by the graph classifier by almost 0.26. Figure 6.12 contains PR curves of the best performing graph classifier setting compared to the random forest. The graph classifier has better performance in the beginning of the curve, but the random forest keeps decent performance and drops off later. While the graph classifier looks for the root cause of the adware infection as a whole, the random forest generalizes, which better captures the diversity of the malware.

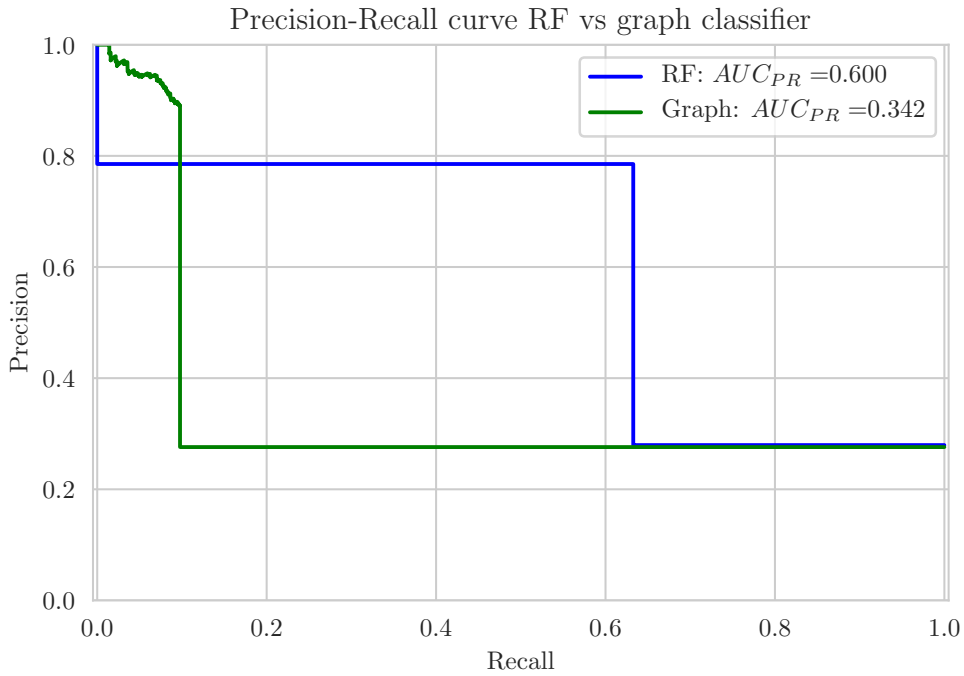


Figure 6.12. PR curve of best performing graph classifier setting compared to random forest.

The unsatisfactory classification performance of the graph classifier on the provided dataset might have many causes. As the user graphs turned out to be much smaller compared to the reference graph, the reference had to be trimmed extensively to eliminate bias towards classifying bigger user graphs as positive more often. The user graphs had to be trimmed as well to remove irrelevant

edges, which in many cases resulted in discarding too many edges and making the classification difficult.

The problems with classification might also be caused by dataset characteristics. The task was to assign a label to the unlabelled DGA containing samples, based on how much they resemble labelled samples. Specifying truly negative samples for training instead of using unlabelled samples as negative could positively impact the classification performance.

Chapter 7
Conclusion

This thesis introduced a graph-based classifier based on graph similarity. This approach was inspired by static analysis. The main aim of the graph classifier is to introduce visually comprehensible classification. The classifier was tested on a labelled network security dataset consisting of sets of high-level actions originating from network logs. Both the dataset and infrastructure for experiments were provided by Cisco Systems.

In the proposed approach, users are represented by graphs of their actions and can be classified based on their similarity to a reference graph. The immediate comprehensibility of the model is guaranteed by its graph-based nature and demonstrated on real cases throughout the thesis.

Experiments with the classifier on the dataset provided showed the impact of weighting and trimming techniques on classification performance, but more importantly they showed how they affect the intrinsic interpretability and readability of the graph. An example graph built for adware was analysed. After proper weighting and after trimming the unimportant edges, the graph was split into multiple partitions. Each partition characterises a core behaviour of adware, ranging from infection vector to monetisation.

Classification performance of the graph classifier was compared to the performance of the random forest algorithm on the same dataset. The random forest generalises better, which is reflected in higher classification performance compared to the graph classifier. However, the graph classifier achieves better precision at the beginning of the PR curve, showing that it can find root causes of the malware infection. The classifier can be considered a success as finding the root cause was one of the main goals. Generalisation improvement can be studied in future work.

Reference graph creation process has areas for improvement as well. Tested approaches include clustering transactions before building the reference, building one reference for multiple labels in data, and building one reference per label. Building a reference based on one or multiple selected user graphs could be tested. Additionally, more sophisticated wrapper or embedded feature selection methods could be introduced to improve the performance. Calculation of edge weights for a combination of multiple events could be studied in the future work as well.

In conclusion, working on this thesis has been an immense learning experience for me, as the challenges I encountered while working on the graph classifier were unlike anything I dealt with during my studies.

References

[1] Barber, David. *Bayesian Reasoning and Machine Learning*. Cambridge: Cambridge University Press, 2011. ISBN 9780511804779. ISSN 9780521518147. Available from DOI 10.1017/CBO9780511804779.
http://ebooks.cambridge.org/ref/id/CBO9780511804779.

[2] Davis, Jesse, and Mark Goadrich. The Relationship Between Precision-Recall and ROC Curves. In: *Proceedings of the 23 rd International Conference on Machine Learning*. Pittsburgh, PA, 2006.
https://www.biostat.wisc.edu/ page/rocpr.pdf.

[3] Louppe, Gilles. *Understanding Random Forests*. Cornell University, 2014. Ph.D. Thesis. Available from DOI 10.13140/2.1.1570.5928.
http://arxiv.org/abs/1407.7502.

[4] Criminisi, Antonio. Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning. *Foundations and Trends in Computer Graphics and Vision*. 2011, Vol. 7, No. 2-3. ISSN 1572-2740. Available from DOI 10.1561/0600000035.

[5] Breiman, Leo. Random Forests. *Population health metrics*. aug, 2011, Vol. 9, pp. 29. ISSN 1478-7954. Available from DOI 10.1023/A:1010933404324.

[6] Verikas, Antanas, Evaldas Vaiciukynas, Adas Gelzinis, James Parker, and M. Charlotte Olsson. Electromyographic patterns during golf swing: Activation sequence profiling and prediction of shot effectiveness. *Sensors (Switzerland)*. 2016, Vol. 16, No. 4. ISSN 14248220. Available from DOI 10.3390/s16040592.

[7] J. C. MacKay, David. *Information Theory, Inference*. 2003. ISBN 0-521-64298-1. ISSN 0263-5747. Available from DOI 10.1017/S026357470426043X.

[8] Shultz, Thomas R, and Scott E Fahlman. *Encyclopedia of Machine Learning and Data Mining*. Boston, MA: Springer US, 2017. ISBN 978-1-4899-7685-7. Available from DOI 10.1007/978-1-4899-7687-1.

[9] Suhang, Wang, Tang Jiliang, and Liu Huan. *Encyclopedia of Machine Learning and Data Mining*. Boston, MA: Springer US, 2017. ISBN 978-1-4899-7685-7. Available from DOI 10.1007/978-1-4899-7687-1.

[10] McInnes, Leland, John Healy, and James Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. feb, 2018.
http://arxiv.org/abs/1802.03426.

[11] Ross, Brian. Mutual Information between Discrete and Continuous Data Sets. *PLoS ONE*. feb, 2014, Vol. 9, No. 2, pp. e87357. ISSN 1932-6203. Available from DOI 10.1371/journal.pone.0087357.

[12] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schutze. *Introduction to Information Retrieval*. Cambridge: Cambridge University Press, 2008. ISBN 9780511809071. Available from DOI 10.1017/CBO9780511809071.

[13] Aggarwal, Charu C. *Data classification: Algorithms and applications*. 2014. ISBN 9781466586758. Available from DOI 10.1201/b17320.

[14] Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*. New York, New York, USA: ACM Press, 2016. pp. 1135–1144. ISBN 9781450342322. Available from DOI 10.1145/2939672.2939778.

[15] Koutra, Danai, Joshua T. Vogelstein, and Christos Faloutsos. DELTACON: A Principled Massive-Graph Similarity Function. In: *Proceedings of the 2013 SIAM International Conference on Data Mining*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2013. pp. 162–170. ISBN 978-1-61197-262-7. ISSN 1556-4681. Available from DOI 10.1137/1.9781611972832.18.

[16] Riesen, Kaspar. *Structural Pattern Recognition with Graph Edit Distance*. Cham: Springer International Publishing, 2015. Advances in Computer Vision and Pattern Recognition. ISBN 978-3-319-27251-1. ISSN 2191-6586. Available from DOI 10.1007/978-3-319-27252-8.

[17] Melançon, Guy. *Just how dense are dense graphs in the real world? A methodological note*.
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00091354.

[18] Blondel, Vincent D., Jean Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*. 2008, Vol. 2008, No. 10, pp. 1–12. ISSN 17425468. Available from DOI 10.1088/1742-5468/2008/10/P10008.

[19] Newman. Analysis of weighted networks. *Physical Review E*. American Physical Society, nov, 2004, Vol. 70, No. 5, pp. 056131. ISSN 1539-3755. Available from DOI 10.1103/PhysRevE.70.056131.

[20] Allen, Frances E. Control flow analysis. In: *Proceedings of a symposium on Compiler optimization*. New York, New York, USA: ACM Press, 1970. pp. 1–19. Available from DOI 10.1145/800028.808479.

[21] Kiechle, Daniel. *Basic Blocks*.
https://www.sra.uni-hannover.de/Theses/2018/BA-FI-basic-blocks.png.

[22] Schultz, M.G., E. Eskin, F. Zadok, and S.J. Stolfo. Data mining methods for detection of new malicious executables. In: *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE Comput. Soc, 2002. pp. 38–49. ISBN 0-7695-1046-9. Available from DOI 10.1109/SECPRI.2001.924286.

[23] Christodorescu, Mihai, and Somesh Jha. *Static analysis of executables to detect malicious patterns*.

[24] Wüchner, Tobias, Martín Ochoa, and Alexander Pretschner. Malware detection with quantitative data flow graphs. In: *Proceedings of the 9th ACM symposium on Information, computer and communications security - ASIA CCS '14*. New York, New York, USA: ACM Press, 2014. pp. 271–282. ISBN 9781450328005. Available from DOI 10.1145/2590296.2590319.

[25] Shang, Shanhu, Ning Zheng, Jian Xu, Ming Xu, and Haiping Zhang. Detecting malware variants via function-call graph similarity. In: *2010 5th International Conference on Malicious and Unwanted Software*. IEEE, 2010. pp. 113–120. ISBN 978-1-4244-9353-1. Available from DOI 10.1109/MALWARE.2010.5665787.

[26] Park, Younghee, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research - CSIIRW '10*. 2010, No. July, pp. 1. Available from DOI 10.1145/1852666.1852716.
http://portal.acm.org/citation.cfm?doid=1852666.1852716.

[27] Park, Younghee, Douglas S Reeves, and Mark Stamp. Deriving common malware behavior through graph clustering. *Computers & Security*. Elsevier, nov, 2013, Vol. 39, pp. 419–430. ISSN 01674048. Available from DOI 10.1016/j.cose.2013.09.006.

[28] Nari, Saeed, and Ali A. Ghorbani. Automated malware classification based on network behavior. *2013 International Conference on Computing, Networking and Communications, ICNC 2013*. IEEE, 2013, pp. 642–647. Available from DOI 10.1109/ICCNC.2013.6504162.

[29] Kopp, Martin, Lukáš Bajer, Marek Jílek, and Martin Holeňa. Comparing rule mining approaches for classification with reasoning. In: *ITAT - Information Technologies - Applications and Theory*. CEUR-WS.org, 2018. pp. 52–58.
http://ceur-ws.org/Vol-2203/52.pdf.

[30] Rehak, Martin, and Blake Anderson. *Securing Encrypted Traffic on a Global Scale - Cisco Blog*.
https://blogs.cisco.com/security/securing-encrypted-traffic-on-worldwide-scale.

[31] Wiki, Jupyter/jupyter. *Jupyter kernels*.
https://github.com/jupyter/jupyter/wiki/Jupyter-kernels.

Appendix A
Glossary

- AUC^{a}_{PR} • Absolute AUC_{PR}
- AUC_{PR} • Area Under Precision-Recall curve
- API • Application Programming Interface
- AWS • Amazon Web Services
- CFG • Control Flow Graph
- CPU • Central Procession Unit
- DAG • Directed Acyclic Graph
- DGA • Domain Generating Algorithm
- EC2 • AWS Elastic Compute Cloud
- EMR • Elastic MapReduce
- FN • False Negative
- FP • False Positive
- GED • Graph Edit Distance
- IDS • Intrusion Detection System
- MCS • Maximal Common Subgraph
- PCA • Principal Component Analysis
- PE • Portable Executable
- PR • Precision-Recall
- RDD • Resilient Distributed Dataset
- ROC • Receiver Operating Characteristic
- SSH • Secure Shell
- S3 • AWS Simple Storage Service
- TN • True Negative
- TP • True Positive
- XML • Extensible Markup Language