



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS

ÚSTAV INFORMAČNÍCH SYSTÉMŮ

**CLASSIFICATION OF POTENTIALLY MALICIOUS FILE
CLUSTERS VIA MACHINE LEARNING**

KLASIFIKÁCIA POTENCIÁLNE NEBEZPEČNÝCH ZHLUKOV SÚBOROV POMOCOU STROJOVÉHO

UČENIA

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR

AUTOR PRÁCE

PATRIK HOLOP

SUPERVISOR

VEDOUCÍ PRÁCE

Ing. VLADIMÍR BARTÍK, Ph.D.

BRNO 2019

Bachelor's Thesis Specification



21927

Student: **Holop Patrik**
Programme: Information Technology
Title: **Classification of Potentially Malicious File Clusters via Machine Learning**
Category: Data Mining
Assignment:

1. Study the topic of machine learning. Focus on models applicable for classification.
2. Get acquainted with Clusty, which is a service used in Avast for clustering of files based on their shared properties.
3. Design a service to automatically classify clusters of files created by Clusty. A classification should contain the severity of a potential threat, and in case of malicious software its type and family. Supported file types should be chosen after a discussion with the supervisor and consultant.
4. Implement the service designed in the previous step.
5. Thoroughly verify the implemented solution by creating a suite of unit and integration tests.
6. Evaluate your work and discuss future development possibilities.

Recommended literature:

- S. Guido and A. Müller: Introduction to Machine Learning with Python, O'Reilly Media (2016), ISBN 978-1449369415
- A. Géron: Hands-On Machine Learning with Scikit-Learn and TensorFlow, O'Reilly Media (2017), ISBN 978-1491962299
- Internal Avast documentation.

Requirements for the first semester:

- The first three items from the assignment and part of the fourth item.

Detailed formal requirements can be found at <http://www.fit.vutbr.cz/info/szz/>

Supervisor: **Bartík Vladimír, Ing., Ph.D.**
Consultant: Zemek Petr, Ing., Avast
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 15, 2019
Approval date: October 26, 2018

Abstract

This thesis proposes an alternative to currently used malware classification approaches on the file-level often based on the detection of specific byte sequences. The experimentation proved, that a cluster-level classification based on the shared properties of files in the cluster is possible. That was achieved by a careful selection of the properties of the three file types – PE, APK and .NET. By comparing various machine learning methods the highest scoring classifiers were selected and a web service providing API for classification was implemented, which was used for the integration with the internal clustering system of the Avast company. This thesis also discusses drawbacks of the proposed approach and suggests steps for improving the classification.

Abstrakt

Táto práca navrhuje alternatívu súčasných metód klasifikácie malvéru na úrovni súborov, ktoré sú často založené na detekcii špecifických postupností bytov v daných súboroch. Experimentáciou bolo potvrdené, že je možné klasifikovať potenciálnu hrozbu aj na úrovni zoskupení súborov založenej na spoločných vlastnostiach súborov v danom zoskupení. To bolo dosiahnuté dôkladným výberom vlastností troch typov súborov – PE, APK a .NET. Porovnaním niekoľkých metód strojového učenia boli vybrané klasifikátory s najvyššou presnosťou a implementovaná webová služba poskytujúca API pre klasifikáciu, ktoré bolo použité pre integráciu s interným systémom spoločnosti Avast zodpovedného za tvorbu súborových zoskupení. Táto práca taktiež diskutuje možné nedostatky a navrhuje kroky pre zlepšenie dosiahnutej presnosti klasifikácie.

Keywords

machine learning, clustering, classification, antivirus, analysis, malware

Klíčové slová

strojové učenie, zhľukovanie, klasifikácia, antivírus, analýza, malvér

Reference

HOLOP, Patrik. *Classification of Potentially Malicious File Clusters via Machine Learning*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vladimír Bartík, Ph.D.

Rozšírený abstrakt

Cieľom tejto práce je preskúmať možnosti klasifikácie potenciálnej hrozby na úrovni zoskupení súborov využitím metód strojového učenia. Tým sa odlišuje od aktuálne používaných klasifikačných prístupov, ktoré sa zameriavajú na jednotlivé súbory. V prípade, že je klasifikácia možná, taktiež navrhnúť a naimplementovať službu, ktorá bude schopná tieto klasifikácie vykonávať.

V úvode práce sa objasňuje potreba klasifikácie hrozieb a práca analytikov zamestnaných v antivírových spoločnostiach, ktorí sú za tvorbu klasifikácií zodpovední. Analytici manuálne analyzujú súbory a hľadajú sekcie, v ktorých sa vykonáva podozrivá či nelegálna aktivita. Taktiež sa zaoberá potrebou automatizovanej klasifikácie spôsobenej vysokými počtami analyzovaných súborov. Jedna z možností automatizácie klasifikačnej služby je využitie strojového učenia, ktorým sa táto práca zaoberá.

Nasledujúca kapitola prezentuje všeobecnú klasifikáciu hrozieb a súborov. Detailné zaklasifikovanie súboru z hľadiska potenciálnej hrozby uľahčuje antivírovej spoločnosti nie len sledovať aktuálne kampane a hrozby, ktoré sa vo svete vyskytujú, ale taktiež jednoduchšie tieto hrozby detekovať. Rovnako umožňujú užívateľom poskytnúť bližšie informácie o potenciálnom riziku. Najabstraktnejší typ klasifikácie je určiť závažnosť potenciálnej hrozby. Rozlišujú sa štyri rôzne úrovne. Čistým súborom sa označuje taký, ktorý nejaví žiadne známky hrozby ani podozrivého správania. Pokiaľ je daný súbor využívaný na konkrétne účely, ako napríklad ťažba kryptomien, označuje sa ako nástroj. Potenciálne nežiadané súbory vykonávajú podozrivé aktivity, ktoré ale neznamenajú priamu hrozbu, prípadne inak obťažujú užívateľa, napríklad pomocou zobrazovania reklám. Ak súbor vykonáva nelegálnu aktivitu alebo môže iným spôsobom ohroziť užívateľa či jeho dáta, je klasifikovaný ako malvér¹.

Klasifikácia závažnosti hrozby je prvý krok k určeniu ďalšieho postupu, ale kvôli vysokej variabilite správania malvéru je ho potrebné bližšie zaklasifikovať. Skupina nebezpečných súborov s podobným alebo rovnakým cieľom sa dá popísať určením typu malvéru, ktorý sa môže bližšie rozdeľovať na rodiny. Zatiaľ, čo typ malvéru určuje abstraktný typ hrozby, ako napríklad vydieranie užívateľa zablokovaním určitých funkcionalít jeho zariadenia či systému, rodina môže určovať konkrétnu kampaň, spôsob vydierania a podobne. Táto kapitola bližšie popisuje najznámejšie typy malvéru a uvádza aj príklady niekoľkých rodín.

Následne sa diskutujú aktuálne používané klasifikačné systémy. Medzi dôležité služby patrí Clusty, interná služba spoločnosti Avast zodpovedná za zhlukovanie súborov na základe ich spoločných vlasností. Služba rozlišuje mnohé typy súborov, ako napríklad PE alebo PDF a pre každý typ súboru vytvára zhľuky na základe jedinečných vlasností. Jednotlivé vlasnosti reprezentujú dáta získané zo statickej a dynamickej analýzy súborov. Statická analýza sa zameriava na analýzu formátu súborov a jeho obsahu uloženého v pamäti na rozdiel od dynamickej analýzy, ktorá pokrýva spustenie súboru a sledovanie jeho správania. Zhľuk súborov je reprezentovaný jedine takými vlasnosťami, ktoré sú pomedzi všetky súbory v zhľuku rovnaké. Clusty je taktiež zodpovedný za klasifikáciu vytvorených zhľukov. Na tento účel využíva rôzne klasifikačné metódy, ako napríklad manuálne hlasovanie analytikov, YARA pravidlá alebo cudzie antivírové detekcie. Na základe istoty jednotlivých klasifikácií vyberie tú najvhodnejšiu. YARA pravidlá umožňujú popísať statické i dynamické vlasnosti súborov a následne skontrolovať analyzovaný súbor, či nespĺňa popísané podmienky. V prípade, že áno, je daný súbor klasifikovaný označením, ktoré sa nachádza v metadátach pravidla.

¹pojem malvér (z ang. malware – malicious software)

Nasledujúca časť práce popisuje konkrétne vlastnosti špecifické pre vybrané tri typy súborov, ktorými sa táto práca zaoberá. Prvý z nich je PE (Program executable) špecifický pre operačný systém Microsoft Windows. U tohto typu súboru sa rozoznáva 13 vlastností, ktoré môžu byť pri klasifikácii použité. Niektoré z nich sa získali počas statickej analýzy, ako napríklad podpis alebo importované funkcie, iné počas dynamickej, ako signatúry či zoznam volaných API funkcií.

Súbory typu APK (Android package) sú typické pre mobilné zariadenia s operačných systémom Android a slúžia na distribúciu a inštaláciu obsiahnutých aplikácií. Keďže majú štruktúru archívu, je možné okrem signatúr, API tried či povolení, ktoré aplikácia vyžaduje, získať i súborové cesty v rámci archívu.

Posledný typ súborov .NET je taktiež navrhnutý firmou Microsoft. Napriek tomu, že oficiálne dodržiava štruktúru definovanú formátom PE, v rámci zhlukovania a klasifikácie sa pre tento typ uchováva špecifické vlastnosti, ktoré sa u PE súborov nerozoznávajú. Súbory definované týmto formátom umožňujú uchovať veľké množstvo pomocných informácií a uľahčujú tak prípadnú analýzu. Príklad vlastností tohto súboru sú odkazované typy vyskytujúce sa v aplikácii, prípadne deklarácie metód.

V ďalšej časti práca objasňuje problematiku strojového učenia a popisuje jeho všeobecné princípy. Strojové učenie je vedecký odbor informatiky, ktorý popisuje algoritmy a štatistické modely, ktoré sú schopné sa zlepšovať pre výkon úlohy, na ktorú sú zamerané. Objekty reálneho sveta, ako sú zhluky súborov, môžu byť popísané sadou diskretných alebo spojitých vlastností, ktoré nazývame príznaky. Formát a tvar príznakov je pre každé pozorovanie rovnaký. Ku každému popisu objektu pomocou príznakov, ktorý nazývame vektor príznakov, môžeme poznať i triedu, do ktorej daný objekt patrí. Táto práca sa zameriava na techniky učenia s učiteľom, ktoré vychádzajú z myšlienky, že pokiaľ model s aktuálnou konfiguráciou vykoná predikciu, ktorá sa dá skontrolovať porovnaním s vopred známou triedou objektu, je schopný na základe chyby, ktorú spravil, upraviť svoje parametre tak, aby sa približoval k lepším výsledkom v budúcnosti. Ďalej sa práca zaoberá spôsobmi, akými je možné transformovať príznaky reprezentované reťazcami znakov na numerické vektory, a metódami, ktoré sú schopné zobrazit existujúce vektory príznakov do priestoru s nižšou dimenzionalitou, čo môže mať pozitívny vplyv na čas, ktorý tréning klasifikátora vyžaduje, i samotné výsledky.

Medzi konkrétne metódy strojového učenia, ktorými sa táto práca zaoberala, patrí logistická regresia, rozhodovací strom a náhodný les, neurónová sieť, bayesovský klasifikátor a metóda K-najbližších susedov. Práca objasňuje princíp, na ktorom je každá metóda založená. Následne popisuje, akým spôsobom je možné validovať a porovnávať natréňované klasifikátory. Jednou z použitých metrík je F1 skóre. Dôležitou metódou je taktiež krížová validácia, ktorá umožňuje rozdeliť dáta na menšie úseky, kde sa iteratívne jeden úsek použije na vyhodnotenie klasifikátorov a zvyšné úseky na tréning. Výsledky sú potom vypočítané ako priemer hodnôt získaných z rôznych testovacích úsekov.

Pretože súčasťou úlohy nie je iba nájsť vhodné klasifikátory, ale taktiež vybrať vlastnosti vhodné na klasifikáciu a zostaviť dátové sady, bola na základe popísaných metód navrhnutá sada experimentov, ktorých účelom má byť nájdenie najlepšej klasifikačnej metódy, dátovej sady a metód na spracovanie dát či extrakciu príznakov. Jednalo sa o deväť klasifikačných úloh, pretože jednotlivé klasifikátory sú od seba nezávislé a určenie závažnosti hrozby, typu a rodiny malvéru prebieha nezávisle od seba. Pre každý typ súborov bolo navrhnutých niekoľko dátových sád, ktoré používajú rozdielne metódy spracovania dát a extrakcie príznakov a na nich porovnané výsledky všetkých popísaných klasifikačných metód. Výsledky dosiahnuté modelmi s najlepšou konfiguráciou sú zdokumentované ku každému

experimentu. Pri výbere klasifikátorov sa taktiež zohľadňoval čas, ktorý modely potrebovali k trénovaniu, a veľkosť pamäte potrebnej pre zostavenie dátovej sady a uchovanie modelu. Pretože sa experimentovalo s viacerými metódami, ako sú popísané v teoretickej časti práce, stručný popis experimentov s danými metódami je zhrnutý v samostatnej sekcii. Ako najlepšie metódy sa ukázali náhodné lesy a neurónové siete, pričom dôležitou súčasťou extrakcie dát pre klasifikáciu typu a rodiny malvéru bola lineárna diskriminačná analýza. Výsledky boli taktiež porovnané s frameworkom umožňujúcim automatizovaný návrh vhodného klasifikačného modelu, kde dosiahli podobné výsledky s jednoduchšou komplexitou modelov.

Služba, ktorá bude dané klasifikátory využívať na klasifikáciu, bola navrhnutá ako webová aplikácia implementovaná pomocou frameworku Flask v jazyku Python. Vytvorená služba sa nazýva Hamlet². Pretože často dochádza k reklasifikáciám existujúcich zhlukov súborov, je potrebné vytvorené klasifikátory priebežne trénovať. Je možné, že sa začnú rozoznávať nové rodiny a typy malvéru a niektoré zaniknú, preto je nutné klasifikátory trénovať znova. Aby nevzniklo vysoké zaťaženie existujúcich systémov, Clusty raz denne vygeneruje obraz svojej databázy, ktorý obsahuje aktualizované informácie o zhlukoch a ich klasifikácii. Tieto dáta sa následne podľa konfigurácie načítajú z pamäte a transformujú na dané dátové sady použité pre tréovanie implementovaných klasifikátorov. Pre metódy strojového učenia bola použitá knižnica scikit-learn. Služba poskytuje webové rozhranie, kde je možné zobrazovať výsledky klasifikácií, informácie o tréovaní, zobraziť grafy o štatistikách klasifikácií a zadať vlastnosti zhuku na zaklasifikovanie. Pre administrátorské účely sa všetky informácie zaznamenávajú do niekoľkých logov, ku ktorým prístup je chránený údajmi špecifikovanými správcom v konfiguračnom súbore.

Aby sa odhalili prípadné chyby v implementácii či budúcom vývoji, bola vytvorená sada jednotkových a integračných testov overujúcich funkcionality a reakciu aplikácie na prípadné chyby. Jednotkové testy sa zameriavajú na funkcionality jednotlivých tried a funkcií, kde integračné testy už pracujú aj s údajmi z konfiguračného súboru, reálnymi databázami a testujú spoluprácu modulov i výslednej služby.

V závere sa diskutujú nedostatky vytvorenej služby a navrhnutého klasifikačného prístupu a poskytuje sa návrh na ich prípadné zlepšenie. Experimentovanie bolo obmedzované výpočtovým strojom, kde by vyššia pamäť dostupná pre výpočty mala pomôcť dosiahnuť vyššie výsledky a rovnako tak využitie vlastností jednotlivých súborov než celých zhlukov. Mnohé rodiny boli zastúpené nízkym počtom zhlukov, čiže by klasifikácia malých zoskupení, ktoré sú momentálne službou zodpovednou za tvorbu zhlukov ignorované, mohla vytvoriť vyšší počet tréovacích dát. Dosiahnuté výsledky súčasnej implementácie boli uspokojivé a služba je momentálne nasadená ako jeden z čiastočných klasifikátorov v službe Clusty.

²názov Hamlet (Hierarchical automated machine learning tagger)

Classification of Potentially Malicious File Clusters via Machine Learning

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Vladimír Bartík, PhD. The supplementary information was provided by Ing. Petr Zemek, PhD. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Patrik Holop
May 12, 2019

Acknowledgements

I would like to express thanks to the supervisor Ing. Vladimír Bartík, PhD. for academic supervision of the technical report and to Ing. Petr Zemek, PhD. for consulting the clustering system and for the help with the integration of the proposed service.

Contents

1	Introduction	3
2	General malware classification	5
2.1	Severity	5
2.2	Malware types	5
2.3	Malware families	7
3	Overview of current classification systems	8
3.1	Clusty	8
3.1.1	Static analysis	9
3.1.2	Dynamic analysis	9
3.1.3	Clustering	9
3.1.4	Classification	10
3.2	YARA rules	11
4	File formats	13
4.1	PE	13
4.2	APK	16
4.3	.NET	17
5	Machine learning	18
5.1	Approaches	18
5.2	Feature engineering	19
5.2.1	One hot encoding	19
5.2.2	Feature hashing	19
5.2.3	Feature selection	20
5.2.4	Feature extraction	21
5.3	Machine learning methods	23
5.3.1	Logistic regression	23
5.3.2	Decision tree	25
5.3.3	Neural network	26
5.3.4	Naive Bayes	27
5.3.5	K-nearest neighbours	28
5.4	Model validation	29
5.4.1	Confusion matrix	29
5.4.2	Accuracy	30
5.4.3	F1 score, precision and recall	31
5.4.4	ROC and AUC curves	31
5.4.5	Learning curve	31
5.4.6	Cross-validation	32

6	Experimentation	33
6.1	Experimentation design	33
6.2	PE	34
6.2.1	Datasets	34
6.2.2	Severity	35
6.2.3	Malware type	36
6.2.4	Malware family	37
6.3	APK	38
6.3.1	Datasets	38
6.3.2	Severity	39
6.3.3	Malware type	41
6.3.4	Malware family	42
6.3.5	Data visualization	42
6.4	.NET	44
6.4.1	Datasets	44
6.4.2	Severity	44
6.4.3	Malware type	45
6.4.4	Malware family	46
6.5	Other approaches	47
6.5.1	Voting classifier	47
6.5.2	Stacking classifier	47
6.5.3	Image classification with CNN	48
6.5.4	SVM	48
6.5.5	FeatureHasher	48
6.6	Comparison with meta-learning framework	48
6.7	Summary of experimentation results	49
7	Hamlet - web classification service	50
7.1	Training and input data	50
7.2	Classification	51
7.3	Logging	53
8	Implementation	54
8.1	Used technologies	54
8.2	API	55
8.3	Classification	55
8.4	Training	58
8.5	Malware tree	58
8.6	Charts	59
8.7	Logging	59
9	Testing	61
9.1	Unit tests	61
9.2	Integration tests	61
10	Conclusion	63
	Bibliography	64
A	Contents of the DVD	67

Chapter 1

Introduction

In the last century began an era of the fast development of computer science and information technology. This phenomenon still continues nowadays. Common users are inspired by lower prices, easy accessibility of the electronic devices and intuitive user interfaces providing connection to users without previously necessary specific technical knowledge. Companies are on the other hand inspired by higher earnings, easier management, prediction tools and wild spreading their products to a high number of customers.

All of these reasons have led to everyday use and easy access to computer and mobile software, web technologies and network communication, creating space attracting unwanted attention of criminals and so called black hat hackers that are trying to exploit both user and company devices or services for their own malicious intentions such as profit, company espionage, cyberbullying.

Stopping the potential threat was and still is one of the main challenges and daily task of system administrators at companies. Increasing network traffic, employees number and technologies used by the companies made it nearly impossible to prevent all kinds of threats. Also, common users using their computers at home remained unprotected against malicious activities. This encouraged the emergence of antiviruses.

The antivirus is a set of tools providing security actions to protect the system on which it is installed. One of the most important tools is a scanner, which scans the memory or files stored in the system, warning about any potential malicious code or file, functioning as a filesystem filter. To achieve this, antivirus contains a database of rules matching the previously analysed malicious patterns, for which the scanner searches [24].

At the beginning, there was no need for automation of the file analysis and the job was handled by a few malware analysts, experts at reverse engineering. They study machine code, looking for malicious patterns. By the rising number of files that had to be analysed from hundreds to millions, this task became harder and more challenging every day [9]. Hackers discovered many new vulnerabilities and there was no time to analyse the files one by one manually, but rather grouping them to clusters. Malware analysts then might analyse the files in the cluster and find patterns in the context of the analysed cluster. The partial automation of malware analysis has also positive impact on the detection coverage.

Behaviour of the malicious files mutated over the years, but the core pattern similarities among the file clusters remained the same. To describe the malicious behaviour more precisely, it was categorized into multiple general types. Later on, each type of behaviour could have been divided into specific families closely specializing the strain of the malicious type.

The terms *pattern finding* and *classification* are one of the main focus areas of the field of the *artificial intelligence*. Classification, as the systematic placement of observed objects into categories, might be simulated by computers based on the observation of human decision-making process. Experts in statistics and machine learning, the subfield of artificial intelligence, designed multiple algorithms, which are able to automate the decision-making process based on the data with or without the supervision of human experts. This scientific field is hugely popular in the recent years, mainly due to financial sector, making predictions of the future stock prices and computer vision for high level understanding of digital images and videos [20].

Those algorithms are also applicable in the field of computer security. Some of them try to convert the files into images, making the image classifier to decide, whether the file is malicious, or to detect the infected area in the binary files [22]. Other classifiers try to make the decision based on data gained from the static and dynamic analysis of the files, either categorical, or continuous. In both cases the classification system has have access to the files themselves and are often unable to provide closer information about the threat.

This thesis proposes an alternative approach to currently used file-level classification systems. It tries to simulate the work of analysts and classify the clusters of files as accurately as possible based on shared properties of files in the cluster and data gained from both types of analysis. It would be implemented as a web service providing an interface for making classification requests. The classification results will be mainly used as suggestions to malware analysts, whether the files in the cluster are malicious and if so also specify the type and family of the malicious behaviour. The implemented service would not need access to the files themselves, just to the data extracted and gathered by the clustering system. It would also provide closer information about the threat than just its possible existence and would be able to calculate the confidence of its own decision. The confidence might be used as a metric for comparison with classifications from other systems and help to choose the best and final classification.

The text is structured as follows. In Chapter 2 is described the overview of classification hierarchy created by antivirus companies, followed by the description of the currently used classification approaches by these companies in Chapter 3. That chapter also provides closer description of the process of gathering the cluster properties from various types of analysis used for obtaining them. The following Chapter 4 describes the file formats that would be classified and their specific properties also representing the clusters. The next Chapter 5 sums up the base of machine learning methods, pre-processing of the properties and approaches to validate the created models. Design of the experiments and results of experimentation are described in the next Chapter 6. General description and design of the service is presented in the next chapter followed by description of used technologies and implementation details in Chapter 8. In Chapter 9 are described tests implemented for automated verification of the service functionality. Chapter 10 closes the text by summarizing the goals and achieved results.

Chapter 2

General malware classification

This chapter describes the threat categories that the antivirus companies have to deal with the most often. Each classification system or analyst tries to classify the observed files as precisely as possible and estimate, whether it represents any risk and also closer describe its category. This information might be used for detecting new types of threats, for other automated detection systems or to inform the end users, what threat are they facing. The more information can user obtain from the represented classification, the better he would be informed, what steps should be taken in order to remove the threat.

2.1 Severity

Severity is the top-level classification of the potential threat and it represents the overall threat risk of the file. Files and programs are created and modified on the daily basis. It does not take any more necessary deep knowledge or higher education to develop simple programs that access the filesystem or operating system interface on the high level basis. The goal behind those actions might be to invoke any kind of malicious behaviour, like to harm the potential user, steal his personal data or even hide a certain behaviour that the user was not acknowledged about, or would not like to be executed. If the motivation behind those actions is malicious, those programs are referred to as *malware* (malicious software), which represents the first type of severity [12]. Those goals might also be achieved by using tools like *Metasploit* to automatically generate malicious files [23]. One can even inject his own malicious part of the code into existing file making the legitimate file malicious.

When the file does not have any malicious intentions, it is labelled as *clean*. Those might be popular web browsers or products of the well-known and trusted companies. Clean files are the most prevalent severity type. There exists also a gray area, when the file does not have any strictly harmful behaviour, but there is a high chance that the user does not know about its presence in the system or can harass the user otherwise. File like this is referred to as *PUP* (potentially unwanted program) [37]. This might include some types of advertising software, information gathering programs, etc. The last category are automated generators or *tools* often used for specific tasks, like mining tools of cryptocurrencies.

2.2 Malware types

Many potential weaknesses were discovered due to the enlarging community taking interest in the computer science throughout the years. Malicious files were often created for a certain

specific task that was easily discovered during the analysis. Some of them were trying to infect other files, others to lock certain actions in the system or to restrict user's access to the system until he has paid a ransom. *Malware type* represents a category of malicious files with common malicious behaviour. With the deployment of the antiviruses detecting and stopping the potential threats, malware creators have started using various ways to hide the malicious part of the software, making them seem like legitimate software for both the detecting systems and users. Even then it is often possible to detect the malicious part and classify its type. The malicious behaviour can be divided into several high level areas. The most common malware types are described below [9, 30, 40].

1. *Trojan*: It is inspired by the Greek tale, where the citizens of Troy were tricked by the statue of wooden horse and after moving the statue into the city of Troy, enemy soldiers hidden in the statue started to rob them at night. This kind of malware tricks the user by pretending to be the legitimate software, but after installation or execution the software instantiates hidden connection with the attacker's system allowing him to perform malicious tasks on the target system. Trojan often represents a general type of malicious behaviour in case that the file was not classified otherwise.
2. *Worm*: The goal of the worm is to spread over the network. Worms might contain a malicious payload, leaving the infected hosts harmed or just increase the traffic of the targeted network.
3. *Fileinfector*: Fileinfectors try to infect other files in the targeted system. Those might be files necessary for system operation or just user's content. They either try to append the malicious part of the code to the targeted file or overwrite existing sections of the file partially or completely changing its behaviour.
4. *Adware*: This name stands for *Advertising software* and the purpose of adware is to show unwanted adverts to the user. This might be realized in many ways like showing the advertising bars in the web browser or popup windows at the targeted system.
5. *Ransomware*: Software often threatens the user by locking the system or by encrypting and deleting his files, unless he pays a financial ransom. This malware type has become very widespread a few years ago. Example of detection rate of this type is shown in Figure 2.1.
6. *Dropper*: Droppers also contain malicious payload and their purpose is not to perform the malicious activity themselves, but to deploy other malicious files to the system. They might contain encoded executable files or code in them as a payload or to download them directly from the network.
7. *Coinminer*: Mining tools were introduced with the emergence of cryptocurrencies, using the user's computer to calculate transaction operations in exchange for a cut in the cryptocurrency. Many websites try to execute the coin mining tools on the victim's computer or the executable files secretly running the mining software in the background without the permission of the user.

Those are the most common types, but there exist many more others. There is no standard describing the malware classification hierarchy and this task belongs to antivirus companies that are responsible for naming the kinds of malware. Antivirus companies use

their own dialects with regard to classification, which leads to the synonyms describing the same malware type with a different name.

Because it is often easy to detect whether it is the purpose of a specific section of software to show an advertisement or to inject other files, it is not easy to classify a file if it contains malicious sections that focus on multiple areas. It could append the payload into other files as well as show ads. In this case, the analyst decides whether the software is classified as a fileinfector or adware. The final malware type is often chosen based on the most significant impact on the file's behaviour. This phenomenon has caused that the classifications of automated systems may vary depending on the final type chosen by the service.

2.3 Malware families

To classify malware types it is useful to know, with what kind of the threat is the system or analyst dealing with, but those areas are too general to provide any closer information. This results in the classification of malware types into smaller specific areas referred to as *malware families* [9].

Malware family is the description of a certain strain of the malware type. It may refer to the specific software, time period or just specify closer area of the software's behaviour. Malware types are established by the community of experts and potential changes are minor. Malware families, on the other hand, are completely left for the classification of a specific antivirus company. They name the families for their own internal comfort of detecting them and are not as widely shared. This results in multiple antivirus companies naming the same family with completely different names or using the same name for slightly different software behaviour. Antivirus companies might try to find those synonyms, but different names are often treated as different classifications. Just for the PE file type there exist more than 4000 unique malware families [40]. Examples of malware families are described below.

1. *Dealply*: Adware that installs the advertisement popups to a targeted browser.
2. *WannaCry*: Type of ransomware introduced in May 2017 that infected more than 200 000 computers [4].
3. *Zeus*: Type of Trojan that tries to steal user credentials and confidential information from the victim.

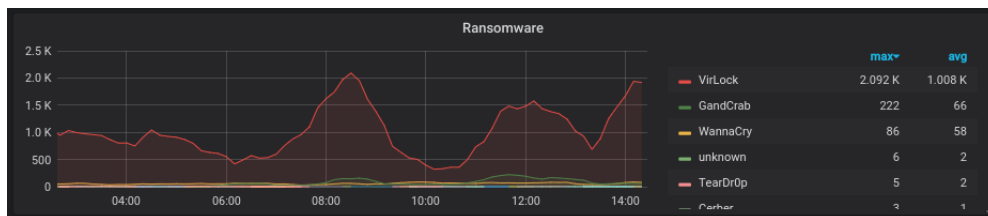


Figure 2.1: Detection rate of ransomware and its families (internal Avast system)

Chapter 3

Overview of current classification systems

Because this project tries to find an alternative to current classification methods, it is important to understand how they work. This chapter contains detailed information about the clustering service Clusty, which provides the necessary data for classification and clusters the files based on their shared properties, the process of gathering and extracting the file properties, analysis types and current classification methods.

3.1 Clusty

Clusty is the internal clustering system used at Avast to cluster files based on their shared properties. It is a real-time service, which collects approximately 500 000 file samples from a variety of sources every day, grouping them into clusters and makes classifications over the created clusters [40]. The file samples are being constantly collected using external services from users, other antivirus products, etc. When the sample is collected, it is sent to multiple independent services responsible for the sample analysis. List of systems, to which the file would be sent, depends on the file type and other high level attributes gathered during collection. Figure 3.1 shows a web interface of Clusty and the classification of the PE cluster based on the external detections.

Clustering Results [Login]

Category: PE [SHA-256 hash or cluster ID] Search [Advanced search]

Show any clusters with any classification having severity "malware" clustered by Import table hash created anytime ordered by sample count limited to 50

clusters Apply filters Reset filters

Number of shown clusters: 50

PE cluster 1 (2 255 981 samples, prevalence: 0) by import table hash		5b0863b7cbd3d0f6b5991f5 2018-05-25 21:27:51
Entry point address:	0x49add5	malware Login to vote Type: trojan Comment: by AV detections Author: clusty Confidence: 88% Created: 2019-02-17 13:57:40 [View history]
Entry point bytes:	550bec6a1f68e0554d06034dd490064a1000000050648925000000003ec585356578965e8ff1580b14b0033d28ad48915	
Import table hash:	28178deeb23ca335978bb93418aba95	
Imports sim:	100% (all 434 imports in common) [Download]	
Languages:	C++	
Rich header:	090c1c7b09000005090b1f6f00000090900a1f6f0000000000e1c830000002c00131f560000002c00131f620000001800010000000002a3005d8883000000	
Section table hash:	03000a2636000000d4090b26360000005f090b1fe80000005e000a1fe80000001b000000000000035000666c700000001	
Section table hash:	509b8254	
Sections:	.data, .rdata, .rsrc, .text	
AV detections (summary):	59% (min), 81% (avg), 90% (max), 0% unknown	
AV detections (top):	Kaspersky: HEUR:Trojan.Win32.Generic (100%), DrWeb: Trojan.DiskFill.41072 (99%), Microsoft: Trojan:Win32/Tescript!rfn (99%), Avira: TR/Downloader.Gen (98%), Symantec: SMG.Heur!gen (97%)	
AV detections (Avast):	100% detected (10 unique detections); Top detection: Win32:Evo-gen [Susp]!+EG704008!egpe (99%)	
Classifications (Avast):	98% infected, 1% clean, 1% unknown	
Classifications (BEC):	99% malware, 1% unknown	
Classifications (Scavenger):	100% malware	

[+] [SHA-256 hashes]

Figure 3.1: Web interface of Clusty showing cluster information

3.1.1 Static analysis

Static analysis involves information gathering from a binary file without its execution. Analyst or automated system parses the binary file by using the *decompiler* (software used for reverse engineering to assemble higher level information from the binary files) or *disassembler* that translates the binary code into assembly language, the symbolic machine code [9]. For non-binary files exist specific parsers. During the disassembling file format dependent data are shown. The data include file format headers, information about used API and libraries or even identifiers named by the author. Then the analyst tries to find malicious patterns or shared patterns among the files.

When a new sample is analysed, firstly its format is recognized. Clusty supports multiple file formats like PE, APK, ELF and is able to retrieve those information automatically with the specific parsers for each file format. For this purpose the tool named *Fileinfo* is used, which is able to analyse most of the formats and it is part of the *RetDec*¹ (Retargetable Decompiler). Once the static analysis is completed and all obtainable information are gathered from the Fileinfo, the clustering process is initiated [40].

3.1.2 Dynamic analysis

Dynamic analysis involves running the tested sample in the *sandbox* environment [9]. Sandbox environment is an isolated virtual machine, which based on how the program behaves in the emulated environment generates an analysis report containing sandbox signatures. Signatures generated during dynamic analysis are short labels generally describing the triggered behaviour. They are mostly high level tags describing a wide area of functionality, for example, *connected_to_internet*. Others are more specific, like *trojan_behaviour*. Alternative to sandbox environments are the *emulators*. While the software in the sandbox is running natively, emulator simulates most of the machine instructions and runs the software step by step [26]. Avast uses its own implementation of the popular sandbox environment *Cuckoo* and emulator *GVMA* (Grisoft Virtual Machine AMD64). GVMA has its own filesystem, registers, emulated processor and virtual memory. During dynamic analysis can also be triggered certain YARA rules described in Section 3.2.

When the file is sent to Clusty, it is also independently sent to other services, few of which are the sandboxes and emulators providing API for dynamic analysis. The dynamic analysis is in comparison with static analysis significantly slower, because it involves running the file in real time and collecting information during the program's execution. Because the program's execution might take a long time, each analysis is limited to a certain duration, after which the program is terminated and analysis completed. Because Clusty often completes the static analysis sooner and the clustering process already began, those services automatically send event information through an agreed channel to Clusty about the completion of analysis and Clusty can gather the necessary information from the dynamic analysis later [40]. When obtaining the event message, it is considered also as a change of the cluster information, thus the reclustering process based on the new properties is triggered.

3.1.3 Clustering

Once all the important data about the file from the static analysis are gathered, Clusty tries to find the most proper cluster for the sample based on file similarity. One sample can be

¹<https://retdec.com/>

clustered only into one cluster if the clustering conditions are met. The sample properties that are considered during the clustering are sorted by their settled priority and Clusty tries to find a proper cluster for the file. Once the sample was clustered, shared properties of the cluster that obtained the new sample have to be recalculated. For example, the sample might not use some dynamic libraries and even though all other samples in the cluster do, these libraries have to be removed from the shared properties. The sample might remain unclustered if Clusty failed to cluster it by all of its properties. During the clustering process multiple sample properties have a different priority. New obtained information about any file of the cluster results in reclustering of all the samples in the cluster [40].

File priorities on which the clustering is dependent are carefully chosen. Retrieved absolute file path or signing authorities has higher priority than the section names or hashes of byte sequences. To overview these priorities, example of the top four properties used during the clustering of PE file is listed below:

1. Corruption
2. Cuckoo YARA rule
3. Uncommon mutexes (100% match)
4. Uncommon named sections (100% match)

From the list is obvious that the highest considered priority has the information about the integrity of files. The corrupted files are then clustered into same clusters and do not introduce noise into clusters with compact files. As is shown in the case of uncommon mutexes, the files have to have some properties exactly the same to be clustered in the same cluster. File formats and their properties are described in Chapter 4.

3.1.4 Classification

Once the cluster is created or changed, it has to be classified. Because the files are being constantly changed and updated, for performance purposes the cluster triggers the re-classification process if any of the first 1 000 samples in the cluster has changed. Clusty allows malware analysts to submit a manual classification, which has the highest priority over all the classification approaches, because the experienced analyst manually analysed the created cluster. YARA rules have lower priority than manual votes and classification results of other automated systems have the lowest importance [40].

Clusty uses the API of *Virus Total*², service assembling the most common antivirus services into one and allowing a user to get the classification summary of the submitted files or web sites. Virus Total currently supports dozens of different antivirus products. These classifications are then internally parsed by Clusty, because each antivirus company uses different convention of naming malware types and families. If any other previous method failed to classify the cluster, those parsed information are used as the final classification. To get the final decision from all supported antiviruses by the service, Clusty calculates simple ratio between those minor classifications.

To estimate how much Clusty believes its own classification results, each cluster contains additional percentage information about the confidence of the final decision. Manual classifications have the highest confidence of 100%. YARA rules have different confidence

²<https://www.virustotal.com/#/home/upload>

ranges from 95-98%. Confidence of other antivirus detections is calculated as a ratio of the most prevalent classification result compared to all other classifications. This number is in the range from 1 to 95%. The highest possible confidence is being purposely limited, because it was obtained from the external sources. Example of those detections can be found in Figure 3.2.

48 engines detected this file			
EXE		SHA-256	000280cf4b53c125a6f07be65693e7c4b1b1b02cc5e1842a028e4a4135b56f30
48 / 68		File name	000280cf4b53c125_winst-8.0.exe
		File size	275.9 KB
		Last analysis	2018-04-23 21:07:41 UTC
Detection	Details	Behavior	Community
Ad-Aware	⚠ Trojan.GenericKD.40198618	AegisLab	⚠ Troj.W32.Gen.IXRx
ALYac	⚠ Trojan.GenericKD.40198618	Antiy-AVL	⚠ Trojan/Win32.Cosmu
Arcabit	⚠ Trojan.Generic.D26561DA	Avast	⚠ Win32:Malware-gen
AVG	⚠ Win32:Malware-gen	Avira	⚠ PUA/ICLoader.uzeg
Baidu	⚠ Win32:Trojan.WisdomEyes.16070401...	BitDefender	⚠ Trojan.GenericKD.40198618
Bkav	⚠ W32.LalaND.Worm	CAT-QuickHeal	⚠ W32.Morefi.A3
Comodo	⚠ Worm.Win32.Viking.jet	CrowdStrike Falcon	⚠ malicious_confidence_100% (D)

Figure 3.2: Subset of AV detections supported by VirusTotal

3.2 YARA rules

YARA (Yet Another Recursive Acronym) is the language used by many antivirus companies to describe the patterns for malware detection [6]. It provides expressions to precisely describe the properties of the analysed files. These properties might be either binary patterns, strings, regular expressions or basic logical conditions, which are also supported by YARA. The situation when a sample matches the properties described by the YARA rule is called the *YARA hit*. Rules created by the analysts can be used both during the static and dynamic analysis and the rules would hit only the described properties or behaviour like malware type. To store the classification information of the rule, YARA allows to specify the *metadata*. In this section can the analyst specify any information like the name of the author, version of the rule or the classification itself.

```
rule AdwareDetection
{
  meta:
    severity = "malware"
    type = "adware"
  strings:
    $string_1 = { A5 FF B1 4C }
    $string_2 = { FD 12 AF 10 11 }
  condition:
    $string_1 or $string_2
}
```

Figure 3.3: Example YARA rule

YARA rules are also used for clustering. When the sample triggers a YARA rule hit, it is clustered with other samples that triggered the same YARA rule [40]. When the cluster is described by the YARA rule, as classification are simply used metadata containing information about severity, type and family as shown in Figure 3.3. Rules have to be constantly updated to minimize the number of unwanted hits on the clean samples and keep track with the newest malware traits.

Chapter 4

File formats

Each file has a defined structure and layout containing necessary information for operating system or any software that needs to parse it, extract the data or execute it. Based on the file format can user easily identify the purpose of this file, like image or document. Some files target specific platform or operating system, for example, PE or APK files are intended for specific platform (Microsoft Windows and Android respectively), but document file formats like PDF (Portable document file) are multi-platform, which means that they are supported by multiple platforms with different architectures. Executable files often depend on the specific platform [9].

YARA rules and malware analysts are looking for a specific pattern when analysing samples. Each file type has it own unique descriptive properties. For the most malware classification systems it is necessary to recognize the file format at first to run the correct parsing subroutines and classifiers [40]. Also, the techniques used by malware creators for exploitation are often completely different across the file formats. The file types that are used in this project are described below. Even though the proposed classification solution does not need access to the files themselves, it uses specific file format properties extracted by the clustering system Clusty shared by all files in the cluster. Within each file type description in this chapter is the list of properties recognized by Clusty.

4.1 PE

PE (Program Executable) is the file format created by Microsoft Corporation and describes the structure of executable files of the operating system Microsoft Windows [5]. This includes simple executable files or dynamic-link libraries (DLLs) that contain functions and interfaces usable in the other executables. Files in the format .NET are also a subset of the PE file format description. However, Clusty tries to extract different properties, which will be described in Section 4.3 as a separate file format. Each PE file consists of DOS header, PE header, Optional header and Section table followed by the specific sections [13].

DOS Header is used mainly for compatibility with the DOS operating system. PE Header contains information about the main structure of the file, for instance information about platform machine or number of sections defined in the following parts. Optional header contains much useful information about the integrity of the file, like a checksum or size of the stack and heap necessary for allocating memory during the execution.

Data are separated into many specific sections. Executable code is often stored in the section *.text*, but constants and literal strings represent read-only data stored in section *.rdata*. Information about section locations and their sizes are stored in the Section table.

Common way of obfuscating the files is the usage of packers. Packers are programs that based on multiple compression algorithms change the structure of PE file and their sections. Parser, which does not support packed files or the specific algorithm used for packing, thinks that the file is corrupted, making it much harder for analysis [9].

Most of the files processed by Clusty are of this file format, supporting the fact that Windows is the most targeted platform by the malware creators, mainly caused by the market share [2]. The attributes that might be used for classification obtained during the static and dynamic analysis are described below [9, 36, 40].

1. Imports: Section containing *Import address table* stores information about functions that the application imports from other modules and serves as a lookup table when the external function is called. Because the compiler does not know the addresses of those functions, this table is filled by the *loader*, program responsible for running and loading the binary before execution, after it has imported external libraries. *Import name table* contains a list of the names of these functions that might be used to classify files, because imported functions might specify the programs behaviour and its purpose. For instance, ransomware might use file system interface functions in comparison with worms that focus on networking. Example of retrieved imports using IDA Free¹ disassembler is shown in Figure 4.1.

Address	Ordinal	Name	Library
000000000407000		Sleep	KERNEL32
000000000407004		FindClose	KERNEL32
000000000407008		FindNextFileA	KERNEL32
00000000040700C		FindFirstFileA	KERNEL32
000000000407010		GetModuleFileNameA	KERNEL32
000000000407014		GetLogicalDriveStringsA	KERNEL32
000000000407018		ExitProcess	KERNEL32
00000000040701C		TerminateProcess	KERNEL32
000000000407020		GetCurrentProcess	KERNEL32
000000000407024		HeapAlloc	KERNEL32
000000000407028		HeapFree	KERNEL32
00000000040702C		GetCommandLineA	KERNEL32
000000000407030		GetVersion	KERNEL32
000000000407034		GetLastError	KERNEL32

Figure 4.1: Imports retrieved during PE file analysis

2. API calls: API calls are related to the same functions as imports, but these are the functions that were really called during the program's execution. Malware authors try to obfuscate the software, so that the detection systems fail to extract the properties from the binary and thus do not recognize its malicious behaviour or indented purpose. For this reason, functions can be dynamically loaded and being listed in the PE import tables, but they are never called during the programs execution. These functions are not detected by the sandbox during dynamic analysis, but other functions might be called very frequently. Clusty contains only the names of the called functions, not the frequency of their calls.

¹<https://www.hex-rays.com/products/ida/>

3. Cuckoo and GVMA signatures: Cuckoo is the name of one of the sandboxes used at Avast, GVMA is emulator of PE files described in Section 3.1.2. Generated signatures describe program behaviour during the execution and cover mostly high level areas, such as networking.
4. Entry point address: Entry point is the relative address within the binary of the first instruction, which will be executed, when the operating system hands control to the application. For dynamic load libraries is specifying of the entry point optional. Very simple technique of executing the injected malicious part of the code is a change of this address in the PE header, so it points directly to malware code. This is easily detected, so the better obscuring techniques use more complicated code redirection.
5. Entry point bytes: The first 100 bytes located at the entry point. Those bytes represent the sequence of machine instructions executed after the application gains control from the operating system. This might be used for detecting the malware containing the malicious code directly at the entry point or generated program patterns.
6. Resources: These values represent additional resource data used by the application, like string values, icons, cursors or images stored in the section *.rsrc*. Each resource is represented by the previously described resource type and language. Language might be useful for localization, because certain languages are more prevalent in the malware than in the clean files.
7. Sections: Each binary file is divided into sections and each section contains different type of data. For example, *.text* section often contains the executable code of the application. Those sections can be modified and packers often encode or modify them to hide malicious behaviour. This process often changes the number and names of the sections and might be detected. Number of sections is information stored directly in the PE header. Clusty can retrieve the list of section names.
8. Rich header: It is a small section of binary data stored after DOS Header created by Microsoft linker. There is no official documentation describing the structure. Rich header might be manipulated in order to cause false detections, as was proven by the malware named OlympicDestroyer in 2018 [3].
9. Signer: When the file is signed by the signer authority, it provides a certain information of trust. For a certain amount of money, companies can buy a digital certificate trusted by the Windows OS. Using the certificate and file hashes can be calculated the final signature called *Authenticode*. However, during the validation of Authenticode are used hashes of only three parts of the PE file, allowing the malware creators to store the malicious code in the other non-validated sections.
10. Exports: As opposite to PE imported functions, when the file also provides a objects to be used in other files, for instance dynamic load libraries, it has to contain information about the functions they export. Exports are extracted from the *Export name table* and represented by the names of exported objects.
11. Languages: Computer programming languages that were used to create the software. This is important, because recent trends show that malware creators tend to focus on a specific set of languages like Delphi. Also, languages like C and C++ require more complex memory management by the programmer and can be more vulnerable to exploitation than others.

12. PDB path: *PDB* file (Program Data Base) is produced by the linker and contains additional information about the executable required to run the program in debug mode. This is very useful for analysis of the crashes caused by invalid memory management or unexpected behaviour of the process. This property is in the Clusty represented by absolute filesystem path to this file.
13. Size: File size of the executable in bytes.

4.2 APK

APK (Android Package) is the file format used by Android operating system mainly on mobile devices. It is typically recognized by the *.apk* or *.xapk* extensions and contains all necessary data for distribution of the mobile applications. APK files are used for application installation and also serve as an archive of the application files [19].

APK package has a strict structure. Each file has to contain *META-INF* directory containing the certificate and the *manifest file*. Manifest is a file describing the main archive attributes. There can be modified the entry point of the application, paths to classes and other important properties. *Lib* folder contains compiled code sorted into directories based on the target platform. Directory *res* stands for resources that are used by the application. The additional Android manifest file named *AndroidManifest.xml* describes the top-level package information in XML format. This manifest file contains information about application permissions, used sensors or supported screen sizes [1, 19].

The Android programming language used to create Android applications is mainly based on the Java. During the compilation is the code transformed into byte code typical for Java and then stored in the DEX file format (Dalvik Executable). File *Classes.dex* in the package contains compiled classes that are being interpreted during runtime. Dalvik execution environment was later replaced by the ART (Android Runtime). ART supports native code execution meaning that the DEX files are compiled into ELF file format and the whole application is managed as a native executable. This process requires additional time for compilation but it is compensated with faster execution. Properties of this file type are described below [1, 40].

1. API classes: Names of the of classes that are implemented in the analysed application. The classes are organized in the package hierarchical structure. Classes having similar functionality or design sections can be grouped into packages. Top-level package is usually *Landroid* that contains all other packages and classes. In the class paths are the packages separated by the dot characters, for example, *Landroid.app.Activity*. They are stored in DEX file format as the part of package content.
2. Sandbox signatures: Labels describing behaviour generated by a sandbox during the dynamic analysis. They are similar to PE signatures, but focused on the Android device services like sending SMS.
3. Archive members: Relative paths of files stored in the package.
4. Android permissions: Each Android application needs user's permission for execution of certain actions. This feature was introduced to restrict access to files, sensors or the camera by suspicious applications. When a simple music player application requests permission for camera, there is a chance that it is a malware taking photos without the

user's knowledge. Those permissions are typically granted during package installation but can be later modified.

4.3 .NET

Files of this file format represent a subset of PE files and they share similar traits. Framework .NET was developed by Microsoft Corporation and serves as a platform for execution of these files [29]. Even though they can be considered as PE files having the same structure, they are recognized by the analysis systems as a different format with unique properties due to their separate traits and behaviour. The most popular programming language for development of the .NET applications is C#. Clusty recognizes the following properties [29, 40].

1. Classes: They represent the classes declared in the application and their format is similar to API classes extracted from the APK package. They are structured in the tree hierarchy, but they might also have the form of a hexadecimal code. Interface of this framework for C++ language was used for developing the application in uncommon cases. In that case C++ elements like *StandardIterator* could be found within the classes.
2. Referenced types: References are in the context of .NET platform variables storing addresses of the actual objects stored in the memory rather than raw data. Those are represented not only by classes or interfaces, but can also be declared with keywords *delegate*, *dynamic*, *object*, *string*.
3. Cuckoo signatures: Labels describing software's behaviour in the Cuckoo sandbox during the execution.
4. Methods: Function declarations reconstructed from the binary. The declaration of the method contains information about the return variable type or class, library hierarchy and parameters. Those information are usually discarded during the compilation. However, C# permits to include the declarative information into the binary. Information about object's visibility, base classes, interfaces, its members and methods, security permission and many others can be retrieved during the static analysis. Examples of the reconstructed method declarations are shown in Figure 4.2.

```
void JARVIS.Thread.EntryPoint.EndInvoke(System.IAsyncResult result)
void JARVIS.Thread.abort()
bool JARVIS.Thread.isAlive()
int JARVIS.Thread.length()
```

Figure 4.2: .NET reconstructed method declarations

5. Properties: Information about the type and name of the properties contained in the .NET assemble, for example, *string MusicExpress.MusicReader.Path*.

Chapter 5

Machine learning

Machine learning is the scientific branch of computer science studying algorithms and statistical models improving their performance on a specific task. This procedure is called learning, because the models consequently change their own decision-making attributes based on new observations they encountered and based on the error they've made either update or confirm their decisions [14].

The most common types of problems are classification and regression. While regression models attempt to predict future values in time series (used in financial sector for stock price prediction), this project focuses on classification. Classification is defined as the systematic placement of observed objects into categories (classes). The observed sample can be any object that is being classified, like the cluster of files. The classes represent groups, into which would be the objects classified. If the task is to estimate the severity of the clusters, then the labels of the classes would be *malware*, *clean*, *tool* and *pup*.

Each sample is identified by features, carefully selected traits describing the object. Data represented by the features are often divided into training and testing set [28]. Training set is a subset of original data space used by models to iteratively increase the model's approximation of the mathematical relations over data resulting into better classification accuracy and other statistical attributes. Testing dataset is used to verify model's ability to classify unseen data.

5.1 Approaches

There exists multiple approaches of improving the model's performance based on the information that are already known during the model's training [14].

1. Supervised learning: The basic principle of this approach is that the correct labels of the dataset samples are already known. When the model iterates through the data samples, it can modify its inner attributes based on the error between the current model's classification of the sample and the expected result.
2. Unsupervised learning: This method is often used for clustering. Nothing other than features is known during the learning and the model tries to find relations between samples based only on features, for example, clusters by their similarity.
3. Semi-supervised learning: This is a special case of learning that combines the unsupervised and supervised approach. It is used when the labels are known, but only for a small subset of the original dataset. Based on these classified samples are by

the unsupervised methods automatically predicted labels for the rest of the samples. Then the model is trained under supervision.

4. Reinforcement learning: This type of learning is an approach using the challenges for the trained model (agent) and correctly fulfilling the challenges leads to a reward for the agent, failing the task to punishment. The goal of the agent is to gain the highest possible score. For example, it is used for the training of automated bots in computer games.

5.2 Feature engineering

Choosing the right descriptive features representing the objects is one of the crucial tasks for high classification accuracy. Most models work mainly with numerical features. Features can be discrete or continuous. Discrete features are often referred to as categorical. Categorical features represent the mapping of vocabulary (a set of possible attribute values) into the space of numbers [28]. For example, when a person is a man or a woman, this might be described by only one feature named *gender* acquiring two possible values, 1 for the woman and 0 for the man (or vice versa). Continuous features might represent values like height or weight of the person and assume a value within a specified interval. The number of features n and their order has to be unified for all samples. The sample X is then described by the *feature vector* of length n , in which the x_i represents the numerical value of the feature i (equation 5.1) [14]. A set of independent observations represented by the features is called the *feature matrix*. The number of observations j is equal to the number of rows in the feature matrix and the number of features n is equal to length of v -th observation X_v [33]. For supervised learning approach, when the class C is known for each observation in the feature matrix, a set of ordered pairs, where X_v is an observation and C_v is its true class, a *dataset* has form described by equation 5.2.

$$X = (x_1, x_2, x_3, \dots, x_n) \quad (5.1)$$

$$D = \{(X_1, C_1), (X_2, C_2), \dots, (X_j, C_j)\} \quad (5.2)$$

5.2.1 One hot encoding

Attributes of the real objects might be described by strings. Names, types and many other attributes are sequences of characters. Because many machine learning models expect numerical features, those values have to be transformed. Let the attribute have n possible values represented by strings. A set of all possible values that the feature might gain is called the *vocabulary* of the feature. Then we can transform this attribute into n features acquiring only boolean values 1 and 0. Each feature represents one possible value of the attribute's vocabulary. The only feature with the value 1 is the one representing the attribute's value. All other features have value 0 [28]. The example of this encoding is shown in Table 5.1.

5.2.2 Feature hashing

When the size of the vocabulary of the features is too big, for example, when just by transformation of one attribute using the One Hot Encoding would be created several thousands of binary features, it is necessary to lower the dimension of this space, because any computation over those data would require extremely high computational resources.

Vocabulary of the attribute $faculty = \{FIT, FEKT, FAST\}$

Features	FIT	FEKT	FAST
faculty = FIT	1	0	0
faculty = FEKT	0	1	0
faculty = FAST	0	0	1

Table 5.1: Example of One hot encoding

This might be achieved by the method called *Hashing trick*, which is often used in the field of *Natural language processing* when the vocabulary is a set of all words of the certain human language. Because the number of all possible words is high and representation by binary feature for each possible word would exhaust the resources, we can transform this vector into vector with lower dimension by proper hashing function like *Murmurhash3* [33]. Hashing function takes the value of the feature and calculates the index in the lower dimensional vector. Depending on the hashing value the calculated number might be negative, so it is important to use its absolute value. To ensure that the index is valid and in the range of the lower dimensional vector, the operation *modulo* can be used. Each value in the hashed vector represents the number of features mapped into its position [7]. This way is memory efficient but often results into lower accuracy than using the original features. The process of the hashing is shown in Algorithm 1.

Data: Feature vector of length v

Result: Transformed vector `hashed_vector` of length d , where $d < v$

initialize `hashed_vector` with zeros;

foreach *feature* in *features* **do**

`index = HashingFunction(feature);`

`index = absolute_value(index) mod d;`

`hashed_vector[index] += 1;`

end

Algorithm 1: Example of feature hashing

5.2.3 Feature selection

Data collected by the companies are often too wide and it is not easy to choose the proper set of features that are useful for the represented problem like classification. One approach is using every information that is available. This often results in a highly dimensional feature vector. Number of features might reach thousands or millions and can far exceed the number of observations, which makes for many machine learning methods an obstacle to be trained properly. Many features might be correlated, meaning that the values of one feature might be estimated based on the values of the correlated feature. This decreases the advantage of having the values of multiple features and using only one of those might not only lower the necessary amount of memory to store the feature matrix but also allows the models to focus on other more discriminative features. *The curse of dimensionality* is the term used in machine learning describing the fact that the more highly dimensional feature space is due to number of features, the harder it is for most models to find a relations between the observed data and the models tend to learn a noise, meaning that they focus on relations that are not increasing the general classification performance of the

model [14]. With so many features the classification models might even ignore the values of a few features, based on which the classification accuracy might be even higher. The idea behind the *feature selection* is to decrease the number of features by the removal of those that might either not provide any new information due to high correlation with other features or by the empirical background. Some models are able to estimate the *feature importance*, meaning that they can numerically express, on which features they tend to focus more during their calculations [27].

5.2.4 Feature extraction

Feature extraction is the process of transforming the features into a completely different form based on proper mathematical methods. The motivation might be either dimensionality reduction such as with the feature selection (reducing the number of features resulting into lower dimensional space) or the assumption that the transformed features would represent more discriminative space [14]. Visualization of dataset with a chart or by plotting the decision regions of the classifiers in the form humans can understand and easily interpret is also possible when the data are in two or three-dimensional space, in special cases also four. Those values might be thought of as coordinates in the space and transformed into the image. Projecting the features into proper feature subspaces is computationally more demanding than the feature selection.

Principal component analysis

Principal component analysis is statistical method that transforms the N dimensional data into lower D -dimensional space, where $D < N$. The main idea of the algorithm is in preserving the highest variance of the data while decreasing the number of features necessary to describe it [14]. The motivation behind preserving the highest variance is based on the changes in distances of the data points. When the data are projected into the lower dimensional space, the relative distances between the original data points may vary. It is presumed that the observations of the same class are close to each other in the original space and randomizing the distances during the transformation may destroy this relationship between data. By projecting the data into the space with the highest variance is reduced the number of cases when the data that were distant in the original space are by the transformation close to each other and vice versa [21]. Data represented in this new space are linear combination of the original data. This algorithm counts with only the data variance, so there is no guarantee that the separation of classes in the new projected space would be higher than in the original one.

Linear discriminant analysis

Linear discriminant analysis (also called *Fisher's discriminant analysis*) is similar to the PCA. However, the algorithm calculates not only the variance within each class when estimating the projection space of the data, but also the variance between the classes. For that purpose it is necessary that the algorithm has access to the labels of the observations to identify them and it is considered a supervised approach in comparison with the unsupervised PCA [33]. From the covariance matrix are later calculated eigenvalues λ and eigenvectors v representing the projection space similarly to the PCA. Eigenvectors represent a new axes of the feature subspace. Those vectors have always a size of 1, meaning that they represent only the directions of the projection. Eigenvalues, on the other hand,

represent the variance in the new subspace. With the highest value of the eigenvalues will be chosen the corresponding eigenvectors as the new axes. In this space due to nature of the algorithm would not only the data have the lowest variance within the classes, but also the highest distance between classes, which often tends to help the classification by the separation of classes in the space [38]. LDA is a very memory and computational complex algorithm, limiting the resources when the number of dimensions and observation is high.

First it is necessary to calculate the Between class separability or variance S_B . Each class can be represented by mean values of its feature vectors. The distance between the classes can be represented by the sum of distances between the overall mean value \bar{x} and the mean value of the i -th class \bar{x}_i , where c is equal to the number of classes (equation 5.3).

$$S_B = \sum_{i=1}^c N_i (\bar{x}_i - \bar{x})(\bar{x}_i - \bar{x})^T \quad (5.3)$$

The within class matrix represents the distance of the observations belonging to the given class from its own mean value (equation 5.4). Within class variance is in comparison to the between class variance minimized, because we want the data of the same classes to be closer to each other. In this equation the $x_{i,j}$ represents a sample of the j -th class, where the \bar{x}_j is the mean value within the class and n_j represents the number of samples of the j -th class.

$$S_W = \sum_{i=1}^c \sum_{j=1}^{n_j} (x_{i,j} - \bar{x}_j)(x_{i,j} - \bar{x}_j)^T \quad (5.4)$$

Based on those 2 values is calculated the projection matrix representing the transformation into the lower dimensional space based on the LDA criterion. The eigenvalues are in this method calculated from the $S_W^{-1} S_B$, because we need to both maximize the between class variance S_B and minimize the within class variance S_W [33]. Once are the *discriminant* components calculated, the data are projected into new subspace.

The difference between the PCA and the LDA is shown in Figure 5.1. The projection of the classes in the direction of the highest variance might not always be a good technique to transform the data and the LDA provides a solution to find a better transformation space. In production problems, the LDA does not often beat the PCA and for the problems where the speed of both training and prediction has to be very high, PCA is more popular option for dimensionality reduction [33].

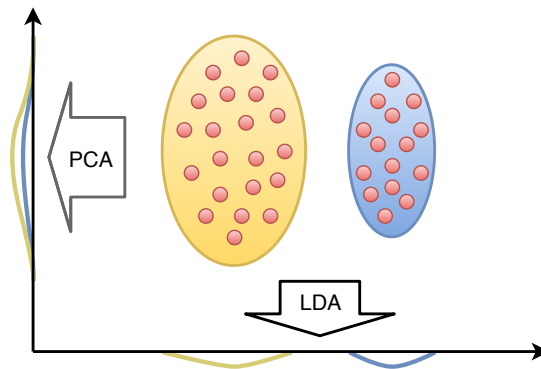


Figure 5.1: Comparison of the PCA and LDA

5.3 Machine learning methods

This section describes the most commonly used machine learning algorithms and methods for solving the real classification problems. There are no strict rules for choosing the best method and finding the best performing method is often left on supervised experimentation, educated guesses and assumptions about the data.

Common classification of the machine learning methods is by their ability to classify linearly separable and non-linearly separable data [16]. Linearly separable data are considered the data that are separable by any linear object in the space described by the features. For example, if it is possible to measure the height of flowers with respect to the number of their leaves and from a plot representing this relation, where each axis represents numerical value of those attributes, be precisely estimated whether the plant is muscatel or rose just by drawing a line representing the decision line into this plot, it represents linearly separable classification problem. On the other hand, some problems are not linearly separable and the decision function needed to correctly classify data is not a linear object, for example, circle as shown in Figure 5.2.

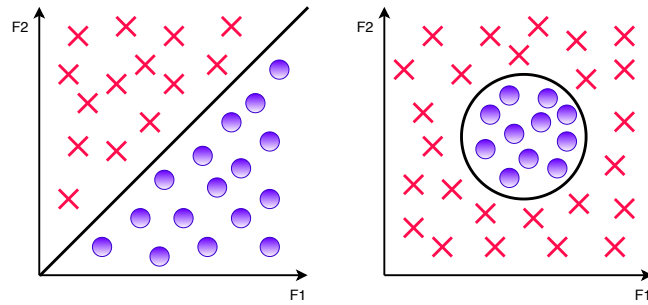


Figure 5.2: Linear vs non-linear classification problem

5.3.1 Logistic regression

Logistic regression is a statistical method used for binary classification. This method results in binary dependent variable. It uses similar idea as the Perceptron designed in 1957 as the simplest neural network and it is based on assigning to each feature i a weight w_i representing the dependency of outcome on the given feature [14]. The higher is the value of the weight, the higher impact is estimated in making the classification prediction.

For that calculation the concept of *odds* is used. Odds represent the ratio between the probability P of a certain class given the certain features x versa the probability of a different class, which in the binary classification is an opposite probability $1 - P$. Then a natural logarithm \ln of the odds is calculated [33]. This function is called *Logit* (equation 5.5).

$$\text{Logit}(P(x)) = \ln\left(\frac{P(y = 1|x)}{1 - P(y = 1|x)}\right) \quad (5.5)$$

Because the relationship between the probability given the features and the logit is linear, it can be also represented as the linear function based on the weights and values of the features (equation 5.6).

$$\text{Logit}(P(x)) = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \dots + \text{bias} \quad (5.6)$$

Logit does not represent the final decision, because it only maps the probability values to the range of real numbers. The proposed decision function is the inverse of the logit function called the *Sigmoid* or logistic function (equation 5.7). This function removes the significant impact of the feature value exceeding the expected value once the weights are estimated and also maps the calculated values in to the range of $\langle 0, 1 \rangle$ [14].

$$S(x) = \frac{1}{1 + e^{-x}} \quad (5.7)$$

The algorithm iterates over dataset and values of the weights change based on the processed samples. The value that each feature provides for the calculation of the final result is represented as the multiplication of weight and feature. This value is then transformed by the sigmoid function transforming the weighted value into the range of $\langle 0, 1 \rangle$. For value in this interval we can estimate a threshold for determining the output value. For example, if the threshold is set to 0.5, any output of the activation function lower than 0.5 is classified as the first class, higher than 0.5 as the second class. To be able to shift the activation function itself, the bias has to be added to the final sum of the weighted features and it is represented as the additional feature always having the value 1 [14].

Error function and gradient descent

To estimate how to change the weights based on the prediction error, the error function has to be used. The main purpose of changing the values of the weights is to minimize the overall error made during the training. The error function representing the relation between the value of the weight and the predicted error (equation 5.8) is a quadratic parabola, which means it has only one minimum value. If the predicted class y_pred and the correct class y_true are represented by numbers and can gain only values 0 and 1, the error can be easily calculated [16].

$$\text{error}(X) = \frac{1}{2} \left(y_pred - y_true \right)^2 \quad (5.8)$$

Each weight has an initial value. This might be a random number or zero. The direction, in which we would like to move the value of the weight is an opposite direction of the *gradient increase*, which is a vector representing the direction of the highest increase of the function in the given point (Figure 5.3).

The changes in the weight should not be too big, because the needed change in the value might be lower and the weight might be changed in the direction of the gradient increase. For that purpose a *learning rate*, which represent the percentage in the changing the weight value, is chosen [33]. New value of the weight would be then calculated as shown in the equation 5.9.

$$w_i = w_i + \text{learning_rate} \cdot \text{error} \cdot x_i \quad (5.9)$$

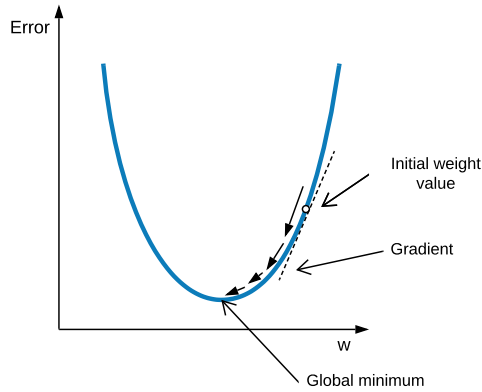


Figure 5.3: Gradient decrease method

With more features this function becomes the function of multiple variables equal to the number of weights. The more complex the error function is, the harder it might be to find minimum of the function. It might contain more local minimums and with the incorrect learning rate it might never converge to the global minimum.

5.3.2 Decision tree

Classification and detection might be viewed at as a sequence of following decisions based on the features, each partial decision more specifying the classified sample [35]. This allows us to split the one classification as a result of more partial decisions. Those decisions might be represented by the tree structure called *decision tree*. Decision tree is a flow-chart like structure, where each node in the decision tree contains specific conditions describing and separating observed samples, for example, whether the height of the man is higher than 1.7 meters. Each node has to also contain references to at least two other nodes, one of which represents the next decision node in case the conditions are met, the other when they are not. The leafs of the decision tree represent the final decisions, in classification case the labels [28]. Example of the decision tree is shown in Figure 5.4

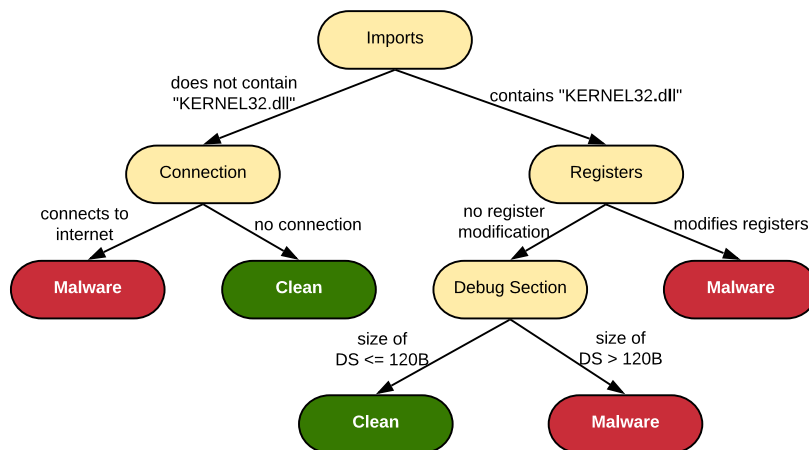


Figure 5.4: Example of the decision tree

Decision tree can be constructed using multiple algorithms, for example, Iterative Dichotomiser 3 (ID3) or Gini index [14]. Both algorithms describe which features to choose and how to create rules necessary to split the observed data into smaller chunks based on feature values. At the start the root node needs to be chosen. With each descending rule node in the tree we expect to separate as much observation as possible, so the root node should contain a rule that splits the highest number of observations.

Algorithm ID3 calculates *entropy* of the dataset. The decision, which features and which values to use in the condition selection is determined by the information that can be obtained from feature (equation 5.10). It describes the fact that the less probable an occurrence of the value c is, the more information it has for the classifier [33].

$$Entropy = \sum_{c=1}^N p(c) \cdot -\log_2 p(c) \quad (5.10)$$

Random forest

Decision trees represent an intuitive approach. However, they tend to overfit meaning the rules constructed by the decision trees are too strict to be able to generalize and describe not observed samples. For this purpose is often used an ensemble method called Random Forest. Random forest is a set of decision trees. Each tree was constructed on different partial set of features, which helps the algorithm to generalize [28]. Prediction is done by each partial decision tree followed by a vote among all trees. The class with the most votes is chosen as the final decision. Construction of random forests is easily parallelized, but the model needs more memory. The main attribute that influences the final decision is the number of trees, but many algorithm support specification of the maximum height of the partial decision trees, number of features that are randomly assigned to each tree, etc.

5.3.3 Neural network

Neural network is a model representing neural activity in the human brain. When humans make decisions, the neural system of human consisting of neurons spreads the electricity through the brain and the body. Each neuron has two types of threads. The first type *dendrites* represents the input of the neuron and based on the value of electric signal received by them either generates or not generates the electric signal. This electric activity might be described by the feature values. The single output thread *axon* is connected to one or more dendrites of other neurons, which connects the neurons into the net and the signal can be transmitted through the body. Neural network simulates this approach and connects multiple neurons in several layers to represent the neural system [33].

The single neural cell *perceptron* is very similar to the logistic regression. The main difference between the original design of perceptron and the logistic regression is that while the output of the logistic regression is a continuous number in the range of $\langle 0, 1 \rangle$, the output of the perceptron is either 0 or 1, which means that a single perceptron is not able to express the probability neither confidence of its decision, because of the activation function used. There exist many types of neurons based on their activation functions [31]. For example, *rectifier* function (equation 5.11) propagates the output only if the neuron's calculation was positive.

$$relu(x) = \max(0, x) \quad (5.11)$$

When more than one layer of neurons are connected between each other, they are able to approximate more complex functions. Neural networks consist of several layers [25]. The input layer consists of neurons which are connected directly to the input values. Each of those neurons might be connected to the neurons the next layer and so on until the final layer. The final output layer then represents the final decision and the probability of the given class. This architecture is called *feedforward neural network* (Figure 5.5).

Layers between the input and output layer are referred to as the *hidden* layers. The neural network without any hidden layer can be trained to solve only linear classification problems. To solve the complex non-linear problems neural net has to have at least one hidden layer [28].

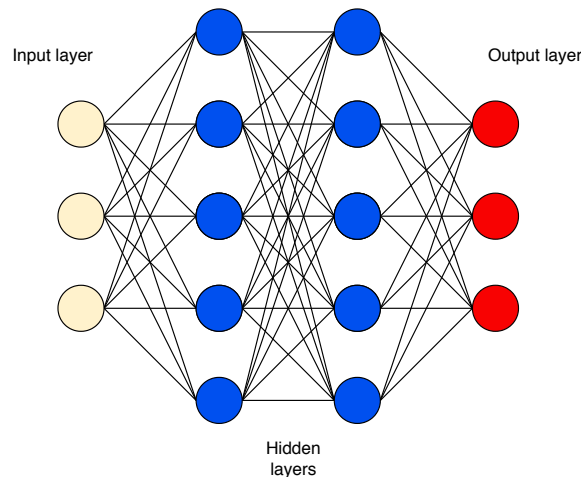


Figure 5.5: Simple feedforward neural network

There exists no rule for choosing the type and the number of neurons and layers in each layer of the neural network called the *configuration*. However, common practice is to choose the number of neurons in the following layers smaller than the previous one. Therefore, relu neurons are very useful in the first layers of the neural network, where the number of input values and neurons is high, but for more precise calculation in the later layers are often chosen different types of neurons. If each neuron of each layer is connected to each neuron in the next layer, we call this configuration the *fully connected* neural network.

Backpropagation

The general error of the neural network is not so straightforward as with the logistic regressions. To update each weight there has to be an assumption of the impact of the current weight value on the error of the given neuron. After the prediction was made, we know the error of the last output layer neurons. The error of neurons in the previous layer has to be calculated with the respect to the error of the neurons in current layer and the weights can be updated. This process can be repeated until all weights of the network were updated [16].

5.3.4 Naive Bayes

Naive Bayes classifier is also a statistical model, which is suitable and often used when there exists an assumption about the observed data that they do have a specific form of

distribution. With that assumption it applies the Bayes's theorem (equation 5.12) [10]. It describes the conditional probability calculating the probability of sample belonging to the certain class y given the feature vector x . The $P(x)$ usually normalizes the data but does not change the distribution itself, so it might be omitted. $P(y)$ is the probability of the response variable and is represented by the proportion of the dataset belonging to the given class y . Finally, $P(x|y)$ represents the likelihood of the training data for the given class. It would be complex to compute, but the algorithm uses an assumption to decrease the complexity. The assumption is that the features are conditionally independent to the given class, so the final distribution is proportional to $P(c_i) \prod_{j=1}^n P(x_j|c_i)$. The goal of the algorithm is to estimate the most likely category [34].

$$P(y|x) = \frac{P(y) \cdot P(x|y)}{P(x)} \quad (5.12)$$

$P(y|x)$ is also called *posterior probability*, where the output variable depends on the evidence data x . For the *Gaussian naive Bayes* is the expected distribution of the of features $P(x_i|y)$ a *Gaussian* or *Normal* distribution (equation 5.13) [39]. This distribution is based on the assumption that the most data would be around the mean value μ and spread around based on the standard deviation σ as shown in Figure 5.6. The second distribution that the feature might follow is the multinomial distribution more typical for the natural language processing problems.

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \quad (5.13)$$



Figure 5.6: Examples of normal distributions

Maximum a posteriori is a decision rule, which describes the process of choosing the most likely class based on the highest probability of the given data points values in the distribution calculated based on the expected distribution (equation 5.14).

$$y = \operatorname{argmax} P(c_i) \prod_{j=1}^n P(x_j|c_i) \quad (5.14)$$

5.3.5 K-nearest neighbours

K-nearest neighbours is a method based on different metrics than the previous classifiers. It is not based on prediction of a certain distribution. However, it assumes that the feature values might be looked at as n -dimensional space coordinates, where n is the number of

features. Based on the correct distance metric used for calculation of distance in the given space, samples as data points of the same class should be located close to each other [33]. When two points in this space are both labelled with the different classes, we assume that the third unlabelled point's class is the same as of the point that is closer to it. This represents a decision based only on the one nearest neighbour. The more neighbours are chosen for the class estimation, the lower variance is expected. This method is very sensitive to not normalized data, because if the features have very different feature ranges, it tends to ignore the feature with the smaller values, because they are insignificant from the view of the distance metric [15]. For example, this may happen when one feature is represented in centimeters and the other in kilometers.

The important step is to choose the correct distance metric. For the boolean value features the proper distance is called the *Hamming distance*. Hamming distance represents the minimum number of substitutions to make the two sequences the same. The non-binary data often use the common *Euclidean distance* as the proper metric. The Euclidean distance between two points x and y , both represented by n -dimensional coordinates, can be calculated as in the equation 5.15.

$$euclidean(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2} \quad (5.15)$$

When the point is classified, a distance to every point of the training dataset must be calculated, then are chosen K points with the lowest distance between those points and the classified one, followed by the vote. The most common class between the K chosen points is set as the prediction.

5.4 Model validation

Once the model is trained, it has to be validated, because it is not known whether the model is not overfitted or underfitted and how it would adapt to new data, which were not used for the training. For that purpose there exist many metrics, which might be used [18]. Those might be used not only for estimating the model's performance, but also to compare more models in order to find the best for the given problem. When trying to estimate the best classification method and configuration of the model, it should not be trained on the whole dataset rather than splitting it into two parts. The first part called *training dataset* would be used to train the classifier and find the necessary patterns. The second *validation part* would represent the unseen data on which the model has not been trained on and we can observe, whether the model is generalized enough to make valid predictions on those data. Proportion of testing data is often chosen between 10 to 30% based on the amount of collected data [14].

5.4.1 Confusion matrix

Confusion matrix offers a closer description of the classification results. For the binary classification problem, it is shown in Table 5.2. It is based on the relation between the actual classes of the testing observations with the predicted ones by the tested model. Confusion matrix visualizes missclassifications between the classified classes [28]. It can be used to spot the classes that are often misclassified between each other.

	Actual class 0	Actual class 1
Predicted class 0	True positives	False positives
Predicted class 1	False negatives	True negatives

Table 5.2: Confusion matrix

1. True positives - Number of samples of the class 0 that the classifier predicted correctly
2. True negatives - Number of samples of the class 1 that the classifier predicted correctly
3. False positives - Number of samples of the class 1 that the classifier predicted incorrectly.
4. False negatives - Number of samples of the class 0 that the classifier predicted incorrectly.

In malware classification and detections problems, *false positives* often represent the number of the *clean* files that were classified as *malware*. The motivation of antivirus companies is to minimize those values, because the antivirus could mark as *malware* system files or other legitimate files resulting into dissatisfied customers and in the worst case also system crashes. Example of the confusion matrix of the file severity problem is shown in Figure 5.7.

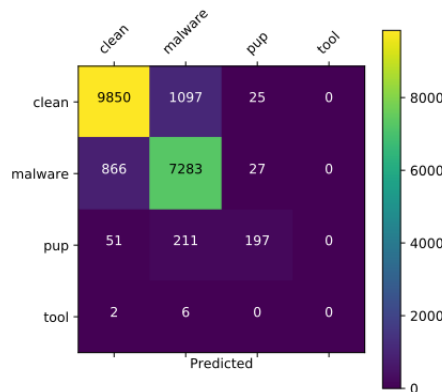


Figure 5.7: Example confusion matrix for file severity

5.4.2 Accuracy

The most intuitive metric is the *accuracy*. It is the simple metric which represents the ratio of the correctly predicted samples c to the number of samples n (equation 5.16).

$$Accuracy = \frac{c}{n} \quad (5.16)$$

The accuracy might be a useful metric when validating the balanced classification problem and all classes have the same or similar proportions in the dataset. However, when

the classes are imbalanced, it might be misleading [28]. For example, in the binary classification problem with the 100 samples where the first class is represented by 99 samples and the second class by only one sample, many classifiers would just ignore the one sample belonging to the second class. This results into accuracy 99%, what would be an excellent result, but the model's ability to discover nor predict the second class is insufficient.

5.4.3 F1 score, precision and recall

Precision and recall are the metrics that represent the facts that accuracy ignores. Precision represents how well the classifier predicts the labels from the chosen ones with respect to the given class (equation 5.17). High precision means that for every class the model predicted, the majority of samples of the classes were classified correctly. Recall, on the other hand, represents how well can model identify the class with regard to other classes (equation 5.18). High recall means that from all the classes in the dataset the model was able to identify most of the samples correctly [14].

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \quad (5.17)$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \quad (5.18)$$

To combine precision and recall into one metric is used F1 score, which merges them into one common metric (equation 5.19). The higher are precision and recall values, the higher is F1 score and the performance of the model [14].

$$F1\ score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (5.19)$$

5.4.4 ROC and AUC curves

Receiver operating characteristics is a metric used to visualize the relationship between the *true positives* and *false positives*. This required the application of multiple thresholds from the interval $\langle 0, 1 \rangle$. For these thresholds is then estimated the *true positive rate* and *false positive rate* [28]. If the classifier would be dummy and in the binary classification problem would guess a random class for each prediction, the ROC curve would be linear relationship between false positives and true positives. Every classifier trained better than the random decision has the ratio between true positives and false positives higher. If the ROC curve values would be under ROC curve of the random classifier, the classifier was trained to classify incorrectly caused most likely by an error in the implementation. Very similar metric is the *Area under curve* (AUC) basically representing the area under ROC curve. The higher are this values, the better classification performance the classifier has [17].

5.4.5 Learning curve

Learning curve is in machine learning used for detection of overfitting and underfitting of the model with respect to the number of training data. Learning curve represents the achieved accuracy with the continually increasing amount of the training data [32]. The plot visualizes values of accuracy of the model with regard to size of the training dataset

for both training and testing data. Expected behaviour is that the accuracy on the training dataset iteratively decreases, because the model is generalizing during the training and with more samples it cannot make any longer as accurate predictions on the training data [28]. This is caused by the higher bias. But this should result in the accuracy increase on the validation set. If the model is not able to learn any new information from more data, the curves would become horizontal meaning that with the increasing number of observed data the accuracy does not change. For the better result is then necessary a change in the model configuration or preprocessing. The distance between the curves on the training data and testing data represents the variance of the model. The goal is to find the best trade-off of the bias and variance.

5.4.6 Cross-validation

When the dataset splits into training and testing set, there is a chance that the training set was formed by the best representative samples for training and the validation score would be high. However, the model should be able to train properly on any subset of the dataset with the chosen proportion, because the next time the model would be trained might be the training set composed of different observations and the higher score might be bound just with this specific distribution. To ensure that the model handles these situations properly, an approach called *cross-validation* is used. The main idea is to choose a proper number k representing the number of folds, into which would be the whole dataset split [33]. One fold is chosen as the testing set and all other folds form the training data during each iteration of the algorithm. The model is trained each iteration and for all iterations are saved the results. The final metrics are calculated as the mean values of all collected results.

Chapter 6

Experimentation

Using the methods and algorithms stated in the previous chapter we try to find the file format properties described in Chapter 4 that have impact on classification performance of the models. The experimentation includes various data pre-processing approaches and configurations of the models themselves. The goal is to design representative datasets and the best performing classification models that would be used by the implemented service. During experimentation for the evaluation of the models would not only be considered achieved classification results, but also memory requirements of the trained classifiers and the time necessary to train them. Because the experimentation involved many approaches, methods with lower accuracy or other reasons not to select them as suitable for the given classification problems are summarized in Section 6.5.

6.1 Experimentation design

The inability to uniquely estimate one specific malware type for each malware family affects the view on the malware classification as a strict tree hierarchy. One malware family can belong to multiple malware types that might vary. Classification of severity, malware type and family is considered as three different problems. Every file format also requires unique classifiers based on different properties. Because the classifiers are independent, the experimentation involves nine classification problems. Data required for experimentation are gathered from database dumps generated by Clusty every day.

Experimentation would be performed using *scikit-learn*¹, a high level experimentation framework for machine learning in Python. This framework provides a highly optimized implementation of all classification methods described in the previous chapter. If a need for a change in the implemented algorithms occurs during the experimentation, we would use our own implementation. Experiments are performed on internal company server with 128 GB of RAM and two Intel Xeon processors with 48 threads. The implemented service would be deployed on the same server. All nine classifiers and other modules that would be required to pre-process the data need to be stored in memory simultaneously. During experimentation would be multiple methods trained in parallel using multiprocessing supported by scikit-learn. However, the number of processes used during training of the model and predicting must remain the same and it cannot be changed later. The implemented service would require multiple instances of the trained classifiers to create predictions simultaneously, so the total number of running processes intensively consuming the time of

¹<https://scikit-learn.org/stable/>

the processor should not be higher than the agreed threshold to keep additional resources for other services and tasks.

Experimentation with each file format firstly involves a high level analysis of the data. There is estimated whether the classification problem is balanced or imbalanced, the number of classified classes, prevalence of missing data, etc. This helps to design a set datasets used for training the models. On each dataset would be trained a set of classifiers based on the classification methods described in Chapter 5.3 with the default configuration of the models. Learning curves of the models with cross-validated results are generated during the training together with other metrics calculated using the test dataset. Based on the results are then repeatedly changed configurations of the models. Grid search method is used for estimation of the best value of certain parameters. The classification methods with best results would be compared and the best method would be chosen. To further validate the method performance meta-learning framework *auto-sklearn* described in Section 6.6 is used. Model validation based on only one dump might not consider the changes of class prevalences and the number of data in time. For that purpose are the results averaged over five dumps, where the difference in time of creation between the dumps is one month.

A set of datasets used for experimentation with each file format described in the following sections is only the best scoring subset of all tested datasets. The experiments also included comparison between one hot encoding and feature hashing, various ranges of hashing function and hashing the properties into the same or separate sections of feature vector, eventually their combination.

Only data that have assigned confidence of current classification stored in Clusty higher than 30% should be used in order to remove the samples with higher probability of potentially wrong labels. Each cluster with lower confidence would be ignored.

Malware family classification problem differs from others, because the number of malware families is very high in comparison with severities or malware types and in the dumps are often only one or few samples representing its family. This problem partially requires the approach called One shot learning, which makes it difficult to properly validate trained models. For that reason is the testing dataset composed only of samples representing families with prevalence higher than one, otherwise it would be impossible to split the data. The samples with unique or low prevalent class would be still used for training.

6.2 PE

PE file format is represented by the highest number of properties that might be used for classification. Most of them are stored as collections of string values, but others originally stored as strings, like entry point address, should be converted into hexadecimal number. By the initial analysis of the dump and the cluster information generated by the Clusty we observe that the dumps contain the highest number of samples among all file types exceeding 300 000. This number is a combination of all clusters including various severities and malware types.

6.2.1 Datasets

Feature hashing and one hot encoding were used to generate the first proposed datasets A and B. The original data represented by strings are before hashing or encoding not pre-processed in any way. The number of imports, API calls and symbols exceeds multiple millions so those features are being hashed into lower dimensional vector. Experimentation

includes hashing them separately into multiple concatenated lower sized feature vectors and hashing them together into the same space. Signatures have lower vocabulary with multiple hundreds of unique values, so they are one hot encoded. Multiple ratios of the hashed vector sizes were tested during the experimentation.

Dataset C contains the hashed values of properties and also their numbers as features. Only four selected properties were used for hashing: imports, signatures, API calls, resources. The sizes of hashed vectors were initially selected based on the vocabulary of the given property and were consequently changed based on the classification results.

Dataset D was generated using the obtained values and extracting potentially useful data. From resources was extracted a name of the language and type of the resource. Languages are one hot encoded and types are hashed. Programming languages that were used for application development are one hot encoded, because in dumps were present only few languages, mainly C++, Java, Delphi, VisualBasic and C#. API calls are already represented by pure function names, which are also extracted from the imports. An additional feature contains boolean value whether the file was signed or not.

Datasets E and F represent the application of PCA and LDA applied to previously mentioned datasets, respectively. This set of datasets uses either PCA or LDA to extract the important features. The proper number of components and discriminants was estimated experimentally.

Dataset G is composed out of function categories. Functions described by imports and API calls are divided by Microsoft Corporation into 96 unique categories based on their purpose or use case, for example, *registry* or *networking*. The number of functions from each category can be used as a feature and then used for training together with other pre-processed attributes. To them one hot encoded signatures and the rest of hashed attributes are being appended. This method was tested by various malware research groups focused on malware classification via machine learning [36].

6.2.2 Severity

Each PE cluster in dump has a valid severity label. By comparing the ratio of all severity labels we observe that the classification problem is imbalanced. The most prevalent are clean clusters with total number over 150 000, followed by malware, PUP and lowest number of samples have tools represented by only few hundreds of clusters. The highest achieved scores among various methods differ. The summary of achieved results is shown in Table 6.1.

A high number of samples resulted into exhaustion of memory during application of LDA and PCA. To test those approaches under-sampling of random samples from the original datasets was used up to 7 000 samples per class and on those datasets was the application of LDA and PCA possible. However, this approach has not outperformed the classifiers trained on the original datasets.

K-nearest neighbours classifier performed best with lower number of neighbours (2–4), increasing this number have led to lower model performance. *Euclidean* distance used as a metric have outperformed the default *Minkowski* distance. This method performed best on the dataset B.

The best performing classifiers were random forests. Time of the decision trees construction was much lower than the time necessary to train neural networks and together with neural networks was this the only method that achieved higher accuracy than 0.9. The well chosen number of decision trees was set to 50 with no constraints regarding the depth of decision trees, the gain in scores achieved by higher number of trees was insignificant.

Lower number of trees also helped to keep lower size of the trained model, which was high in comparison with other methods. Maximum number of features assigned to each decision tree was set to $\log n$, where n is the total number of features.

This method had similar classification results as neural networks. Experimentation involved various configurations, numbers and types of neurons as well as the number of layers. Configurations with one hidden layer containing 55–60 neurons achieved the best results. Increasing the number of hidden layers had a negative impact on classification results, because based on the learning curves of the models they required more data to train properly. However, models with multiple hidden layers and low number of neurons achieved similar scores. The best performing type of neurons was *relu*.

In terms of recall, linear regression achieved similar results as K-nearest neighbours or naive Bayes. However, its precision was significantly lower. Slightly lowering the default regularization helped to improve the F1 score. This approach was outperformed by naive Bayes, where the precision was higher by 0.11. However, the training time was longer.

Method	Dataset	Precision	Recall	F1 score
KNN	B	0.859	0.833	0.846
RF	C	0.968	0.932	0.950
NN	C	0.954	0.921	0.937
LR	B	0.721	0.849	0.800
NB	C	0.832	0.867	0.849

Table 6.1: PE severity – achieved results

6.2.3 Malware type

Information about cluster classification retrieved from dump should contain valid malware type, but it might not be further specified by malware family. To use as much information as possible for training, those clusters are not ignored and used for malware type training. Dumps provide information about 26 unique PE malware types. The number of malware types contained in the latest generated dump was higher than in the oldest one. Because the number of malware samples is lower than the total number of samples with any assigned severity, experimentation with LDA and PCA did not result in memory exhauster. Achieved results are shown in Table 6.2.

K-nearest neighbours classifier did not achieve high accuracy when trained on the original datasets and the application of LDA was necessary to achieve higher classification scores of the models. K-nearest neighbours had the best performance when the number of neighbours was chosen between five and seven together with Minkowski distance as active metric.

Random forests had the best results among the tested methods with F1 score above 0.85. The proper number of decision trees for classification was estimated to be 30 with number of features assigned to each tree equal to \sqrt{n} . The application of LDA was also necessary for this method to achieve the highest score. However, application of LDA on datasets with highly dimensional vectors necessary to achieve F1 score 0.880 did use most of the available system memory and would not be applicable in the classification service. For this reason was chosen as the final classifier random forest trained the dataset F, which

contains hashed values of properties into lower dimensional space. The best results were achieved when the number of LDA components was 30.

Neural networks achieved the second best score among the tested methods on the dataset F. The best performing configuration contained one hidden layer with 45 hidden *relu* neurons. Logistic regression performed similarly to neural network regarding the datasets. However, naive Bayes classifier achieved the worst result among all methods.

Method	Dataset	Precision	Recall	F1 score
KNN	B	0.726	0.702	0.714
RF	F	0.850	0.852	0.850
NN	F	0.824	0.865	0.844
LR	F	0.781	0.776	0.778
NB	F	0.696	0.672	0.684

Table 6.2: PE malware type – achieved results

Although most methods have achieved the highest scores on the dataset F, dataset G consisting of numbers representing prevalence of function categories has achieved better results than during severity or malware family classification. The importance of the most relevant categories generated using trained random forest is shown in Figure 6.1.

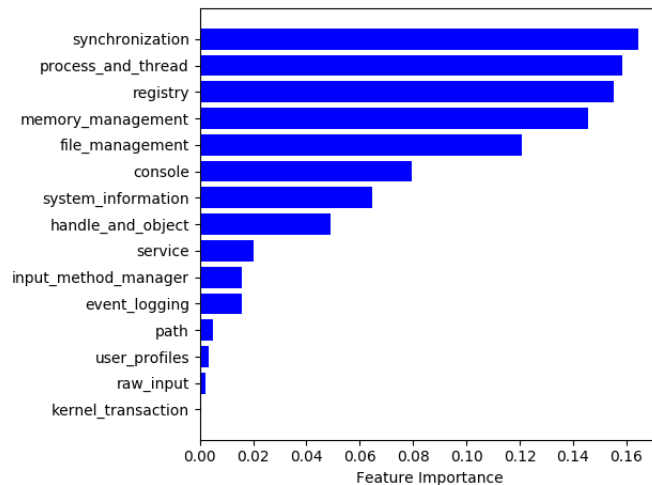


Figure 6.1: Feature importance of PE function categories

6.2.4 Malware family

Not all clusters with malware severity have to have assigned a type or family, but there are cases when the cluster classification information contains only the severity *malware* and family. High number of PE malware families (3814 in the latest dump) is not common in machine learning classification problems. The number of families also slightly varies among the multiple generated dumps. As was stated in Section 6.1, many families are represented by only one or few clusters. Classification results are shown in Table 6.3. The average achieved precision was generally higher than recall.

K-nearest neighbours performed well on the randomly over-sampled dataset A. However, the highest number of neighbours must have been chosen up to three. The oversampling has caused longer training time of this method in comparison with previous types of classification problems. As one of the few methods, the application of LDA have not resulted into better results.

Random forest achieved the highest results on dataset F. Similarly to malware type, larger feature vectors with application of LDA resulted into the highest classification accuracy. Limitation for decision tree depth or increasing the number of features used by each tree did not have positive impact and the best performing classifier was the forest with 30 trees, where each tree uses \sqrt{n} sized subset of features. Application of LDA increased the performing accuracy by $\approx 12\%$ in comparison to the original dataset. The number of discriminants was estimated to be 150.

In the previous types of classification problems were the results achieved by neural networks similar to the random forests. However, in this problem has the random forest significantly outperformed neural network. This might be caused by the high number of classes and higher number of missing data among the clusters than in the previous classification problems. The best type of neuron was *sigmoid*. To properly classify the malware families was required one hidden layer of 60 neurons.

Logistic regression and naive Bayes performed poorly in comparison with other file types or severity classification. Logistic regression with lower regularization performed better, but neural networks has outperformed it significantly.

Method	Dataset	Precision	Recall	F1 score
KNN	B	0.812	0.799	0.805
RF	F	0.845	0.810	0.827
NN	F	0.814	0.802	0.808
LR	A	0.769	0.756	0.762
NB	F	0.756	0.693	0.723

Table 6.3: PE malware family – achieved results

6.3 APK

APK dumps contain lower number of clusters than are contained in dumps of other file formats. In comparison with PE and .NET dumps where the number of missing values is noticeable, APK clusters often contain most of the properties and therefore the number of missing data is significantly lower. This might have positive impact on classification results and allows us to experiment with all properties. The second main difference in comparison with other file formats is that APK dumps do not contain any clusters with *tool* severity, so only *malware*, *clean* and *pup* would be used as severity labels.

6.3.1 Datasets

Experimentation with APK clusters involves various combinations of all four cluster properties described in Section 4.2. Not only properties in raw format as stored in the dump without any pre-processing were tested during experimentation, but also various techniques to change their format or extract partial information from them. All APK properties are

represented by the lists of string values. In the case of APK information extraction only two attributes had impact on the final classification results – API classes and Android permissions. Name of the class can be extracted from the signature of API class. In order to do this, it has to be removed template information from the signature identified by pointy brackets if the class is generic. Class can be identified by the last name in the hierarchy. It is also necessary to remove additional identifier information recognized by the character '\$'. Finally the name is converted to lowercase.

To lower the memory requirements necessary to store one hot encoders it is possible to remove the permission structure and extract only the name of permission action (last in the hierarchy, also lowercased). Modifying the paths of archive members was not necessary, because removing the relative path of the file inside archive and keeping only its filename did not result into any improvement from the experimentation.

The first proposed dataset A contains hashed values of original properties contained in the dump. The only pre-processing step was converting the string values into lowercase characters. Experimentation involved testing different sizes of the vector and ranges of the hashing function. Experimentation also involved comparison of hashing the properties into the same space and separately.

The next step involves comparison of dimensionality reduction techniques. The second dataset B applies PCA to the best performing instance of dataset A with and without additional data pre-processing. The third dataset C applies LDA to the same feature vector. Experimentation involves finding the best number of principal components and discriminants as well as regularization properties of the algorithms.

Datasets D, E and F represent customized combinations using different feature ranges for specific properties that also contain the total numbers of specific properties representing the cluster as independent features. Detailed description of composition of feature vectors is shown in Table 6.4. Summary of all datasets used for learning of APK classifiers is described in Table 6.5.

Dataset	Features			
	No. archive members	No. signatures	No. API classes	No. permissions
D	Hash(300) API classes	OHE permissions + signatures	–	–
E	Hash(10000) API classes	OHE permissions + signatures	Hash(10000) archive members	–
F	Hash(20000) API classes	OHE permissions + signatures	Hash(20000) archive members	–

Table 6.4: Custom datasets – APK

6.3.2 Severity

Severity classification of APK clusters is a three class classification problem, because APK Clusty dumps do not contain any tools. The average ratio of class prevalence in the five observed dumps is as follows: Malware \approx 28%, clean 67%, pup 5% with total number of

Dataset	Expected No. features	Used methods
A	1000-20000	Hashing
B	30-200	Hashing, PCA
C	10-200	Hashing, LDA
D	800-1000	Hashing, OHE
E	10-200	Hashing, OHE, LDA
F	10-200	Hashing, OHE, LDA

Table 6.5: Summary of tested APK datasets

samples around 195 000. The variance of those numbers is up to 3% among the various dumps, so we can estimate that this distribution is relatively stable. To weight the prevalence of the classes equally during the training, so the classifiers would not overestimate certain classes, the significance of each sample is changed accordingly to the prevalence of its class using *class_weight* parameter when fitting *scikit-learn* models.

The best results of methods for certain dumps are shown in Table 6.6. As the best classification method was chosen neural network with 100 neurons in the first hidden layer on dataset D. Worst results on the other hand achieved naive Bayes classifier.

K-nearest neighbours did perform well with low number of neighbours (2–3), further increasing the number of neighbours lowered the performance of the model. The best metric used for distance calculation was *Euclidean* that performed better than the default *Minkowski*.

Logistic regression performed best on the simplest dump A using only feature hashing with no pre-processing of dump properties. This was achieved by lowering the L2 regularization parameter and using the default stopping criteria to be 10^{-4} .

Neural networks took the longest time necessary for training among all methods, because other methods were trained using in parallel using multiple processes. Best performed configurations were composed of *relu* neurons. One hidden layer with 100 neurons was enough to achieve expected results. Increasing the number of hidden layers nor changing the number of neurons did not have any positive impact on models performance, only on the increase in training time. Neural networks had the best classification results.

Random forest had slightly lower its best estimated performance than neural networks. However, the number of tested datasets where random forests outperformed neural network was higher. It significantly outperformed single decision tree with the increase in accuracy over 20% when using 30 to 40 decision trees. The best performance achieved the models that have the number of assigned features for each decision tree calculated as the square root of the overall number of features.

Naive Bayes had the worst results among all methods. They were outperformed regarding the training time and final accuracy and this method did not have the best performance on any dataset.

Datasets that use feature extraction techniques were not tested completely, because application of LDA or PCA resulted into exceeding available memory. To test those methods the first 7000 samples of each type were selected to lower the number of samples used for pre-processing calculations, while the testing set was composed out of the rest of the samples. However, this approach had lower results than other approaches.

Method	Dataset	Precision	Recall	F1 score
KNN	A	0.947	0.950	0.948
RF	D	0.965	0.969	0.967
NN	D	0.970	0.970	0.970
LR	A	0.961	0.962	0.961
NB	D	0.855	0.654	0.741

Table 6.6: APK severity – achieved results

6.3.3 Malware type

In average APK dump there are $\approx 54\,000$ malware samples. The number of samples for each class is highly unequal. Malware types like *exploit* have only few dozens of samples while the most prevalent type *trojan* $\approx 10\,000$. The total number of malware types in APK dumps is 19. Class weights need to be balanced properly for balanced classification.

As is visible from the results of malware type classification (Table 6.7), the most significant impact on the performance of the models had an application of LDA increasing the F1 score more than 8% in comparison with the training on the original datasets. PCA did not improve achieved classification results and neither did the not previously pre-processed dataset A. Highest results were achieved when the number of discriminants was close to the number of classes, the best was estimated to be 10 using random forest with 30 trees. Changing this number had a negative impact on the results. The time of LDA calculations exceeded the time of training the models on dataset without application of LDA.

Neural networks have slightly lower accuracy than random forests possibly caused by higher number of classes resulting into higher amount of output neurons. Also, a higher number of neurons in the hidden layer was needed. However, changing the number of hidden layers had no positive effect. Equally as with severity classification for neural networks had to be slightly lowered regularization parameter.

Linear regression achieved lower classification results. K-nearest neighbours classifier performed similarly to severity classification with the best results when using low number of neighbours (2–4). Both methods performed best on dataset C.

For better separation of malware types were needed longer feature vectors than for severity classification. The best results achieved the random forest on the dataset E. Its results are better than other methods and its construction was performed faster than training the neural network.

Method	Dataset	Precision	Recall	F1 score
KNN	C	0.955	0.954	0.954
RF	E	0.962	0.960	0.961
NN	E	0.959	0.962	0.960
LR	C	0.950	0.959	0.954
NB	E	0.948	0.945	0.946

Table 6.7: APK malware type – achieved results

Performance of some models like logistic regression was improved by using semi-supervised learning approach. Only manually classified samples were used as the initial labels and the

rest of the labels were calculated through label propagation method. As shown in Figure 6.2, it had a positive impact on the performance of logistic regression. However, this phenomenon was achieved only on worse performing datasets.

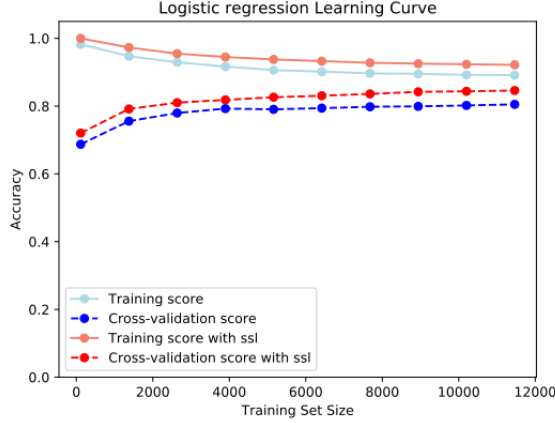


Figure 6.2: Semi-supervised approach improvement

6.3.4 Malware family

More than 1100 malware families are present in APK dumps. That is a high number not common for machine learning problems. Further data analysis revealed that majority of the families is in the dump present only several times and part of them only once, which makes the malware family classification problem partially a One shot learning problem.

Linear discriminant analysis was again significant for creating a dataset that correctly separates classes. The best performing classification method was random forest similarly to malware type, easily handling the high number of classes opposing to logistic regression and naive Bayes. The application of LDA results into small number of features and low number of trees was sufficient enough to classify the malware families, when the height of the trees was not reduced by any constraints. The number of LDA discriminants was 350, what is the highest among all other classification problems.

Logistic regression achieved the lowest results together with naive Bayes and any tested configuration of those methods did not increase the classification accuracy.

K-nearest neighbours did also not respond well to One shot learning problem. For that reason an oversampling method increasing the number of samples using SMOTE approach was applied [8]. While it slightly improved classification accuracy of linear classification methods and K-nearest neighbours, it did not outperform random forest trained on the original datasets using LDA.

For testing of LDA on the oversampled dataset we did not have sufficient computational resources, mainly enough memory for such computation. Summary of the best results is shown in Table 6.8.

6.3.5 Data visualization

A visualization of the most prevalent types and families in the APK datasets is shown in this section (Figure 6.3). Images were generated using dataset A (not pre-processed

Method	Dataset	Precision	Recall	F1 score
KNN	E	0.878	0.874	0.875
RF	C	0.896	0.908	0.902
NN	C	0.874	0.872	0.873
LR	E	0.739	0.778	0.758
NB	E	0.779	0.632	0.698

Table 6.8: APK malware family – achieved results

hashed features) and then LDA for dimensionality reduction into three-dimensional space was applied. Axes represent values of the extracted features. Images confirm results from confusion matrices about the classes that are often miss-predicted between each other. Classes are represented with different colors. Overlapping colors of the classes in those images predict a missclassification possibilities between those classes in real classification.

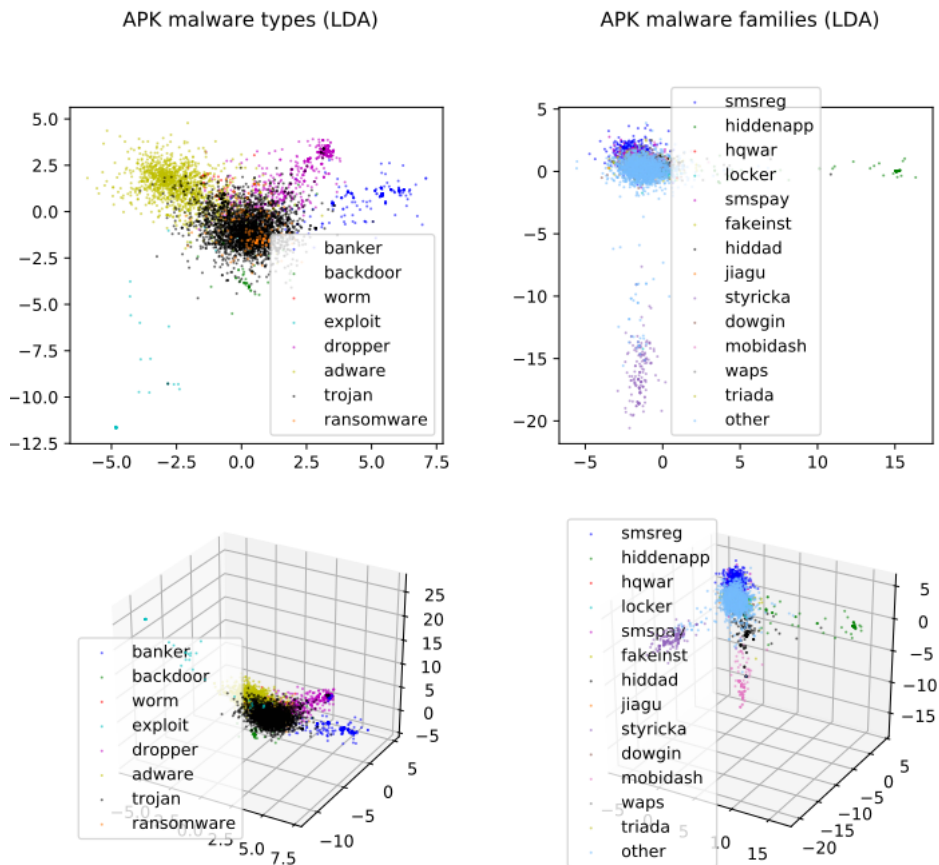


Figure 6.3: Visualization of APK malware types and families

6.4 .NET

Dumps of this file format require the most disk space to be stored, because they contain long function signatures and properties represented by a high number of string values. They also contain a high number of clusters, slightly lower than the largest PE dumps. Most of those clusters are clean, so amount of data used for malware and family classification is comparable with other file formats. Properties of .NET clusters also contain higher number of missing data than APK clusters. All properties are represented by the lists of strings.

6.4.1 Datasets

Clusters of this file format are represented by five properties described in Section 4.3. Signatures do not require any pre-processing. However, from other properties it is possible to extract additional information. A return type, number and types of parameters can be extracted from method signatures. Return types are then one hot encoded as well as short vector of selected extracted types as parameters that are observed. The hierarchy might be omitted from the class information and only the relevant class name should be hashed. Referenced types are also represented by the full hierarchy of the given type, so the same method is applicable on them as well. Properties are represented by their type and name. Types might be one hot encoded or hashed and property names are hashed.

The first proposed dataset A does not contain any additionally pre-processed data and all features are hashed into vectors, where the experimentation involves length of those vectors and hashing combinations. Datasets B and C apply LDA and PCA on best vectors of approach A achieving the highest results to compare dimensionality reduction techniques and their impact on class separation.

Datasets D, E and F are designed to combine various features and pre-processing techniques, experimentation with them involves data pre-processing as described above, using feature hashing and one hot encoding and application of LDA and PCA. Custom feature vectors are described in Table 6.9.

Dataset	Features			
	No. classes	No. signatures	No. properties	No. methods
	No. ref. types	–	–	–
D	Hash(1000) classes	Hash(200) signatures	Hash(1000) methods	Hash(1000) referenced types
	Hash(1000) properties	–	–	–
E	Hash(10000) classes + methods	Hash(100) signatures	Hash(100) referenced types	–
F	Hash(50000) all properties	–	–	–

Table 6.9: Custom datasets – .NET

6.4.2 Severity

The structure of .NET dump is similar to APK dumps, because significant part of the dump consists of clean clusters. The average ratio in the dumps between the classes is as

follows: malware $\approx 14\%$, clean 85% and the rest are *tool* and *pup*. To weight the prevalence of the classes equally during the training so that the classifiers would not overestimate certain classes, the significance of each sample is changed accordingly to the prevalence of its class. Classification results are shown in Table 6.10.

Classification results were compared during experimentation when the training set was constructed out of all original samples and when under-sampling of the most prevalent *clean* class was applied. Results have shown that under-sampling had a negative impact on classification performance of the models implicating diversity of .NET clean samples. This fact remained true even after application of LDA on subsampled dataset.

The best performing models were instances of random forests followed by K-nearest neighbours. Increasing the number of decision trees in random forest had a significant impact up to 50 trees, after this number the results did not change significantly. Positive effect did also have a change in number of features for each decision tree from \sqrt{n} to $\log n$. K-nearest neighbours performed best when $K = 6$, what is slightly higher number than in severity classification of other file formats.

Neural network with one hidden layer with 35 *sigmoid* neurons did outperform logistic regression. However, the achieved result was lower than the F1 score achieved by random forests. Usage of *relu* neurons in the hidden layer resulted into slightly lower score. The worst performing method was naive Bayes with F1 score only 0.585. On the other hand the precision of this method was comparable with other approaches.

Method	Dataset	Precision	Recall	F1 score
KNN	E	0.948	0.952	0.950
RF	D	0.961	0.962	0.961
NN	D	0.948	0.949	0.948
LR	D	0.941	0.945	0.943
NB	A	0.901	0.433	0.585

Table 6.10: .NET severity – achieved results

6.4.3 Malware type

Twenty different malware types are present in the latest .NET dump, what is a higher number than the number of APK malware types. The number of malware samples is however lower than PE format ($\approx 30\,000$). This allows us to use longer feature vector for feature hashing as well as application of LDA on the whole dataset.

All achieved results except K-nearest neighbours performed best on the dataset C. The best result was achieved when all properties were hashed together and not into separate ranges of final feature vector. This dataset must have been transformed by LDA of 20 discriminants, which supports the theory proposed during experimentation with APK clusters, suggesting that the number of components required for the best classification is close to number of classes for general malware type classification. The best performing dump and results are shown in Table 6.11.

Random forests and neural networks achieved similar results that outperformed other methods. Best configuration of neural networks contained hidden layer with 30 *relu* neurons. Figure 6.4 shows learning curve visualizing the phenomenon that similar results might have been achieved also by training random forest using higher number of decision trees on

under-sampled dataset. Neural network required less memory to store the trained classifier than the constructed decision trees. However, the training time was higher, so the random forest was chosen as the final classifier.

Logistic regression outperformed K-nearest neighbours and naive Bayes, which had the lowest overall score. Logistic regression required increase in the number of iterations to converge. Since the LDA was necessary for all classifiers to achieve F1 score above 0.90, K-nearest neighbours was the only method than performed better on dataset E.

Method	Dataset	Precision	Recall	F1 score
KNN	E	0.911	0.895	0.902
RF	C	0.925	0.920	0.922
NN	C	0.920	0.923	0.921
LR	C	0.912	0.917	0.914
NB	C	0.913	0.907	0.909

Table 6.11: .NET malware type - achieved results



Figure 6.4: Learning curve of random forest

6.4.4 Malware family

Clusters of this file format were identified by 419 malware families. This value had changed over time and multiple dumps contained different number of families. However, this number is still lowest in comparison with APK and PE clusters. Application of LDA had again impact on increasing the overall classification results and the best results were achieved using 30 components.

Even though usual experimentation results of severity or malware type classification resulted into similar F1 scores among top performing classifiers, experimentation with .NET clusters has shown that random forests are the best classification methods with higher difference between the best and the next method. Also, usually the best performing datasets for malware family classification were the ones with longer feature vectors than necessary

for malware type classification and this was also true in this classification problem, because the classifier performed best on dataset F. Results are shown in Table 6.12.

K-nearest neighbours performed best with only 1 neighbour using *Euclidean* distance as a metric. Neural networks required higher number of *relu* neurons in the hidden layer (65–70), but it outperformed logistic regression in both precision and recall. Neural network also used adaptive learning rate, which was constant until the error sequentially decreased, but in the later phases of training it started to slightly decrease.

Naive Bayes classifier was the only method that achieved F1 score over 0.9 together with random forests. However, its training time was slower.

Method	Dataset	Precision	Recall	F1 score
KNN	B	0.844	0.858	0.851
RF	F	0.916	0.924	0.920
NN	E	0.867	0.884	0.875
LR	E	0.678	0.731	0.704
NB	E	0.913	0.907	0.909

Table 6.12: .NET malware family - achieved results

6.5 Other approaches

Overview of used approaches and experimentation results described in the previous sections focuses on the models applicable in the final implemented web service. However, experimentation was done using many other techniques, but there was not enough space to describe them. This section represents a high level overview of other classification approaches that did not achieve expected results or were not applicable in the classification service due to different reason.

6.5.1 Voting classifier

Voting classifier is an ensemble method that combines multiple weak classifiers into one, trains them separately and the final prediction is based on voting among them. The final chosen class is the one with the highest number of votes among them [14]. This approach is supposed to suppress the disadvantages of individual classification methods and achieve higher results. Experiments have shown that this approach achieved the best results during classification of severity and malware type of APK clusters with slightly higher accuracy ($\approx 0.5\%$). The voting classifier was composed out of random forest, neural network and KNN. However the significantly increased training time and higher model size were reasons for not selecting this method as the final classifier.

6.5.2 Stacking classifier

This is a meta-learning method that adds a new meta-layer into the training pipeline. Predictions of multiple classifiers can be considered as independent features for another classifier, which tries to learn, when certain classifiers miss-predict samples and corrects those errors [14]. Stacking classifier had generally lower final accuracy than proposed solu-

tions described in previous sections and required a second layer that generated dataset out of existing classifiers to be trained, which required necessary time and memory.

6.5.3 Image classification with CNN

Malpedia² is a website that focuses on malware and cybersecurity. It uses interface called Apiscout that allows to generate images of malware samples based on their properties. Those colourful 32x32 pixel images might be used as dataset for further classification by *Convolutional neural networks*, special kind of neural network used in computer vision and image classification [22]. Multiple common CNN configurations like LeNet-5 were tested. The experimentation also involved black and white image classification, various feature map sizes in convolutional layers and layer sizes. Achieved results were significantly worse than methods described in previous sections.

6.5.4 SVM

SVM (Support vector machines) is a classification algorithm maximizing the margin between the separating hyperplane and samples [33]. A general rule³ proposed by scikit-learn creators suggests that their implementation of SVM is suitable when the number of samples is lower than 100 000. Otherwise the time necessary to train the classifier might be extremely long. Experimentation with SVM had an overall classification performance similar to K-nearest neighbours and slightly lower than the best found configuration of neural networks. The time necessary to train this classifier with current library implementation exceeded other methods even on smaller datasets.

6.5.5 FeatureHasher

Scikit-learn implements its own feature hashing class called *FeatureHasher*. This method uses Murmurhash3 as hashing function equally as our own implementation. However, it does not calculate absolute values out the function output to ensure that the index is positive, but if the resulting index is negative, the value 1 is subtracted from the value stored on the given index. This approach should penalize the hashing collisions with a certain probability. However, only increasing the value on the calculated index in our implementation achieved slightly better classification results.

6.6 Comparison with meta-learning framework

Auto-sklearn is a high level framework built as extension of scikit-learn to provide automated search for the best classification methods [11]. User can select memory and time constraints as well as methods that should be used for creation of the final classifier. Auto-sklearn also supports data pre-processing methods and feature manipulation. This framework was trained for each classification problem on the best performing dataset after evaluation of proposed experiments described in the previous sections. Training time necessary to find the best models was set to three days for each problem. The difference in F1 score between our proposed models and the ensemble models found by this framework was not higher than 0.02, but the structures and configurations of the generated ensemble models were far more

²<https://malpedia.caad.fkie.fraunhofer.de/>

³https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

complex and resource consuming regarding the training time and memory requirements. Our classifiers proposed for PE and .NET format classification outperformed the classifiers generated by this framework.

6.7 Summary of experimentation results

The purpose of the experimentation was to find the best classification models of the cluster severity, malware type and malware family. Various feature pre-processing approaches and machine learning methods were tested in order to achieve the highest classification results. The experiments were conducted for nine imbalanced classification problems involving three file formats – PE, APK and .NET.

The best results for cluster severity classification were achieved with datasets composed of hashed properties and one hot encoded data. Application of linear discriminant analysis had the most significant impact on the classification results of malware types and malware families. LDA has not only increased the class separation that resulted into higher classification accuracy, but it has also reduced the dimensionality of feature vectors. Principal component analysis did not achieve similar improvements in terms of accuracy. Because the application of LDA on the whole datasets was impossible for severity classification due to memory limitations of the server, experimentation also involved application of LDA on under-sampled datasets, but this approach did not achieve a sufficient positive impact on classification performance.

The results have shown that our implementation of feature hashing not allowing the collision elimination during the data transformation has slightly improved the classification results in comparison with the default implementation of feature hashing in scikit-learn.

Random forests were chosen as the best classification method (except for the severity of APK clusters) due to their superior accuracy, training time and ability to handle missing data. Severity of APK clusters should be classified by neural networks. In the case of APK classification, additional extraction of data from the stored properties also helped to achieve higher classification results. Classifiers of PE malware types and families achieved lower scores than classifiers of other file formats.

All classification results were validated among multiple dumps generated by Clusty and later compared with classifiers generated by automated meta-learning framework that did not achieve higher results in the most cases.

Chapter 7

Hamlet - web classification service

Based on the results of classifiers described in the previous chapter, the best performing datasets and classification models should be used in the implemented service. Because Clusty needs to be able to get the classification results of those classifiers, an independent internal web service was designed. The official name of the service is *Hamlet* (**H**ierarchical **a**utomated **m**achine **l**earning **t**agger). Clusty generates the necessary updated data for training of the classifiers and Hamlet provides an interface that Clusty can use to request cluster classification and get classification results or additional information. This chapter describes a general design of the service.

7.1 Training and input data

Clusty saves all the information about active clusters in the internal database. For performance reasons it generates a partial dump of this database once a day. This dump contains only necessary properties about clusters that Hamlet needs for training. The extracted data are serialized using the JSON serialization format.

Each cluster is identified by unique *cluster_id* as shown in Figure 7.1. Hamlet can later gain information about the cluster classification directly from Clusty. Labels that represent classes during the supervised learning are saved as string values of the corresponding classification type (severity, malware type or family). Information, whether the classification was manually submitted by the analyst and was not generated by other automated classifier, is stored together with the classification labels. If the tag was generated by an automated service, a confidence provides closer percentual information about how confident is the selected classifier with the current classification. Confidence of the manual vote is 100%. Properties of clusters are mostly represented by the lists of attributes.

A separate file is generated into shared folder on the hosting server for each classified file format. Because the storage capacity of the hosting server is limited and everyday creation of dumps would consequently exhaust the memory in few days, Hamlet should automatically remove dumps over time.

Hamlet loads data from the generated dumps each day at the same time and retrains the classifiers. For the training Hamlet ignores clusters with classification of lower confidence than empirically estimated threshold 30%. This threshold is used to ensure that low confident classifications would not be used for training, because they might lower the classification results by introducing missclassified clusters. The time of training together with other essential configuration variables of the service such as supported file types, properties,

```

{
  "cluster_id": "5ab0afdecabd30d42147041f3",
  "classification": {
    "severity": "malware",
    "type": "worm",
    "family": "Xindl",
    "manual": true,
    "confidence": 30
  },
  "properties": {
    "imports": [
      "MSVBVM60.DLL:DllFunctionCall",
      "MSVBVM60.DLL:EVENT_SINK_AddRef",
      "MSVBVM60.DLL:EVENT_SINK_QueryInterface",
      "MSVBVM60.DLL:EVENT_SINK_Release",
      "MSVBVM60.DLL:_CIatan"
    ]
  }
}

```

Figure 7.1: Example of the JSON cluster

number of loaded samples, etc., can be specified using a configuration file. When the service is initiated, it tries to load stored classifiers from the memory based on the implementation of the classifiers. If any partial object like one hot encoder or scikit-learn classifier necessary for file format classification is corrupted or the stored file does not exist, new instances of those objects are automatically generated. Until the scheduled training Hamlet would be unable to classify clusters of the specific file format. Figure 7.2 shows an overview of the training process and classification flow.

For the functionality of the web service are responsible multiple processes, instances of the service able to simultaneously fulfil client requests, both the API and web interface. These processes (also called *workers*) are managed by the web server, but the main master worker is also delegated and responsible for retraining of the classifiers. When the training is done, it sends a signal to itself that triggers updating of other web workers.

Each new encountered malware family, malware type or severity from the dump is stored into Hamlet's internal database deployed on independent database server. Users could then access the results from the web interface and see which classes is Hamlet not able to classify by highlighting all of the currently classified families and types extracted from the classifier objects.

7.2 Classification

Because Hamlet is an independent service and not a direct part of Clusty itself, it has to provide an HTTP API that other services like Clusty can use. For this purpose the application supports multiple endpoints to which Clusty can send POST data representing JSON containing information about cluster. The response is also a JSON object containing generated classification tag and additional information about classification. Each JSON representing classification contains exact time of the classification (standard ISO 8601¹),

¹<https://www.iso.org/obp/ui#iso:std:iso:8601:-1:ed-1:v1:en>

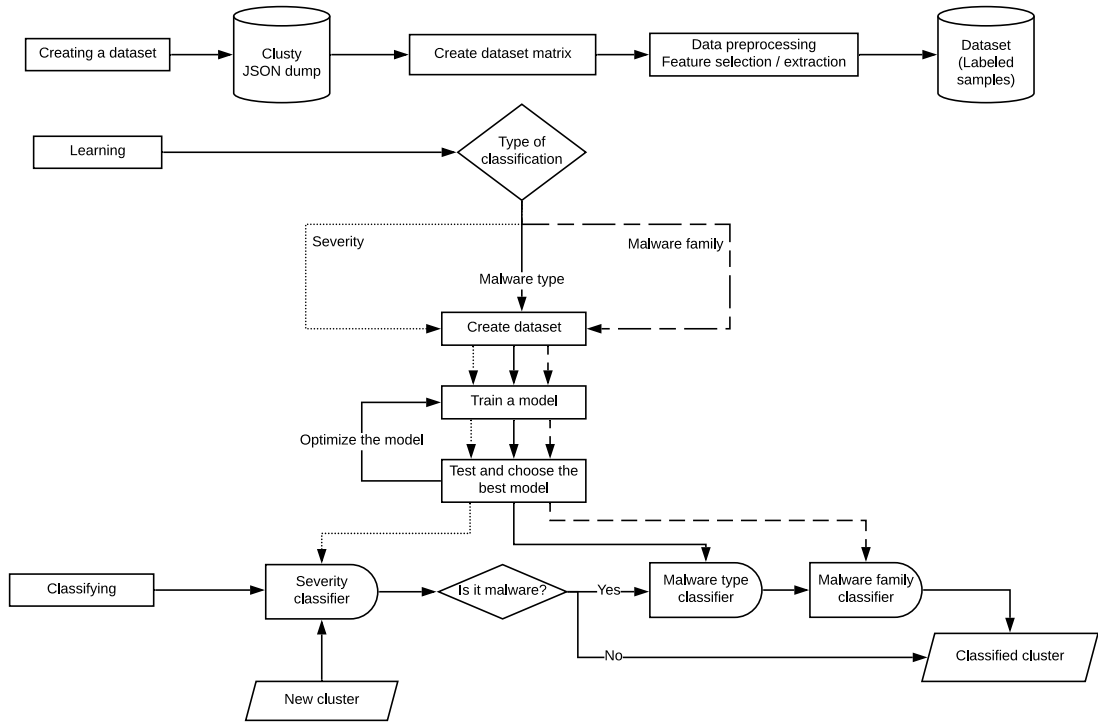


Figure 7.2: Initial design of the action flow

confidence of each partial classification and also the names of the estimated classes. Once Clusty gets the Hamlet’s response, it uses the partial confidence information to calculate the overall confidence of the whole tag and compare it to other classification methods such as decisions based on other antiviruses. Classification with the highest confidence is selected as the final classification and stored into Clusty’s database. To ensure lower number of false positives and proper comparison with foreign antivirus classifications, the confidence of the whole classification is recalculated by the formula stated in equation 7.1.

$$confidence = \frac{7 \cdot severity_conf + 4 \cdot type_conf + 3 \cdot family_conf}{14} \quad (7.1)$$

Each made classification decision has to be saved into Hamlet’s internal database. If there is a problem with missclassifications or specific cluster, it must be possible to find all or just the latest classifications generated by Hamlet for the given cluster by using the cluster ID. Also, it must be possible to generate plots representing the latest classification trends based on the aggregated information.

Observation of the implemented service in time and analysis of its classification results in production environment have led to implementation of secondary classification rules and additional change in original design. Multiple times has occurred the same missclassification pattern. When the cluster was missclassified as *clean* and the confidence of severity classification was low, both malware type and malware family classifiers would agree with high confidence on the result that would be correct. Also, if the confidence of malware type classification is low and confidence of malware family is high or vice versa, the analysed classifications were often correct in favor of the higher confidence. For the change of predicted malware type or family Hamlet needs to know which malware families belong to a certain

type and also confidences of the lower probable classes. When Hamlet needs to change the type or family, it chooses the class with the next highest probability that also belongs to the certain family or type based on the information from dump. Hamlet then might also change the severity to *malware*. Those rules must be applied after all machine learning estimations are done. This means that the cluster needs to be classified by all classifiers and Hamlet must not finalize the tag when the severity is not estimated as *malware*. This slightly changes the initial classification flow shown in Figure 7.2.

7.3 Logging

To monitor the functionality of application, performed activities and to catch any unexpected behavior or error, web service needs to log its activity. Because logging of all information into one log would make it harder to easily find needed information, Hamlet separates the logged data into four different logs, each focused on different aspects of the overall behaviour of the service described in Table 7.1.

Log name	Examples of logged events
SERVER	Main service flow, loading or generating of classifiers
CLF	Performed classifications, IP addresses of requesting clients
DB	Database access, information handling
TRAINING	Encoding progress, application of LDA

Table 7.1: Service logs

Chapter 8

Implementation

In this chapter are described implementation details based on the design presented in the previous chapter, used technologies and description of the provided API that other services use for communication with Hamlet. It also presents designed web interface of the service.

8.1 Used technologies

For each functionality an appropriate existing technological solution that fulfils the necessary demands or provides usable interface was chosen. As the base of machine learning was chosen Python library scikit-learn. This machine learning framework for Python provides high level model implementations and most of the data pre-processing methods. If the methods were either not implemented or the behavior differs from the expected one, they were implemented from scratch. For example, this was used for implementation of custom feature hashing.

Because the scikit-learn was implemented in Python and many other internal Avast systems like Clusty are as well, Python 3 was chosen as the main language for implementation of the web service using the Flask¹ framework. Flask is useful web development framework often used if the demands on the web interface should be simplicity and easy maintenance with focus on back-end functionality, what meets the demands on Hamlet. Flask provides default development WCGI library called Werkzeug. However, it is not recommended for production deployment. For this reason was used Gunicorn² - Unix WSGI HTTP server. Gunicorn also supports multiple workers for hosting the web application.

Basic structure of the web interface was designed using HTML and its style using CSS. For creation of dynamic web pages a templating language for Python called Jinja2 was used. By inserting pseudo Python commands directly into HTML template can then Flask, which supports Jinja2, based on attributes passed to generating function execute the template commands and generate the final HTML page. This is useful for generating HTML documents with dynamic content, for example, visualization of classification results. For chart creation was used Python package called Pygal³, which allows an easy integration with web frameworks like Flask.

Because Hamlet needs to store information without specific scheme, like the list of families for each dump, NoSQL database MongoDB was chosen as the main database to

¹<http://flask.pocoo.org/>

²<http://gunicorn.org/>

³<http://pygal.org/en/stable/>

store the data. Furthermore, Clusty uses the same type of database, so Hamlet can share the same database server. This is shown in Figure 8.1.

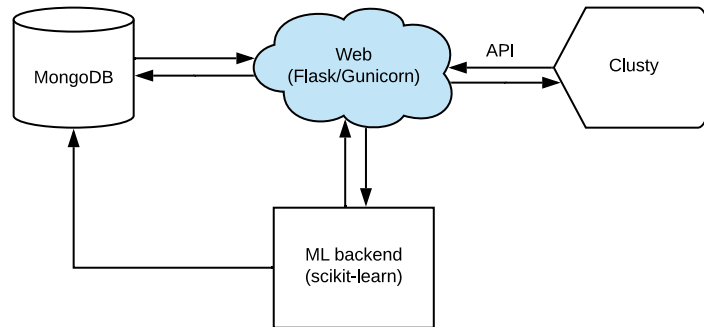


Figure 8.1: MongoDB access flow

8.2 API

Hamlet provides API for other services that require its classification results, mainly focused on the needs of Clusty. Response of Hamlet is JSON containing requested information. API covers all demanded use cases of the service. All implemented endpoints are described in Table 8.1. For accessing private information like logs the user needs valid credentials that are stored in the configuration file of the running service. The JSON representing cluster has the same format as the one generated into dumps and used for training. However, it does not have to include current classification information and if so, they would be discarded.

Endpoint	Description
<code>server/api</code>	Overview of API endpoints
<code>server/api/log/<log_name></code>	Content of requested log
<code>server/api/classify_cluster/</code>	Classification of JSON cluster sent via HTTP POST request. Returns JSON representing classification tag.
<code>server/api/cluster_info/<id></code>	Returns the list of classification results of cluster (JSON data) for the given <i>id</i> . If the parameter <i>last_only</i> has assigned value True, only the last classification result is returned.

Table 8.1: Overview of API endpoints

8.3 Classification

The core of classification service is represented by classifiers that are implemented as an independent set of classes for each file format. They all share the same base class `GeneralClassifier`. This class should serve as an abstract base for all classification models implemented also in the future. It contains the property `classifiers` together with all

necessary constraints for them. From the class are derived main production classifier classes PE, APK and DOTNET_FeatureHashingClassifier. Although they share similar methods and traits, those are not covered by the base class to highlight the independence of classifiers and to allow easier changes in the future models without dependency on other formats. Such a classifier contains multiple dictionaries where each key represents type of partial classification like severity and contains models like encoders or feature extractors ensembling the final classifier. During the initialization of the object it tries to load saved models from the memory and if this task fails, it also generates new untrained models. User can explicitly forbid the loading process on initialization or set the expected model destination to testing directory. Each classifier implements multiple core functions described below.

1. **prepare_data**: Based on the arguments passed to this function it transforms received object representing loaded data into feature vector based on the design of dataset for either severity, malware type or family classification estimated by experimentation.
2. **fit**: Trains the models so that they can be able to generate predictions. This method receives the classification type, which describes the classifier that should be trained. If the classifier uses one hot encoding, those models are trained beforehand. This method returns the accuracy score calculated on training dataset.
3. **fit_all**: Trains severity, malware type and family models.
4. **predict**: Generates prediction for obtained JSON representing properties of the cluster. This prediction contains full classification tag (severity, malware type and family), even if the severity is not *clean*, and help description representing the order of lower probable decisions, because those information are necessary for secondary classification rules.
5. **save**: Saves the models into specified destination.

Classifiers require data pre-processing models like hashing functions and one hot encoders. They are implemented in the module `data_preprocessing`. Hasher is implemented manually because of the phenomenon described in Section 6.5.5. The implementation is based on Algorithm 1.

The older versions of classifiers are represented by `CERClassifier` (Classifier-encoder-reduction) and `DSCClassifier`. Those were implemented as prototypes of the classifiers and were based on the emphasis that all classifiers for severity, type and family would use the same dataset and varies only in the final machine learning methods. They were also used during the experimentation.

User can see stored classification tags on the default webpage of the web service. The classification results are visualized by table. The classified names are colourfully distinguished based on the severity tag using the same color scheme as Clusty. To easily access cluster information from Clusty is the cluster ID also a reference to Clusty web page showing information about the selected cluster. User can also search the results based on file type, classification tags, cluster ID, confidence or limit the number of shown results. Example of the shown results is represented by Figure 8.2.

Once the Flask receives a classification request and leaves control to routine responsible for handling the given endpoint, the data are checked for validity. The cluster JSON must contain properties, ID and other necessary attributes to be valid. If the validity checks fail, JSON representing error message is returned to client. Once the validity is confirmed, then

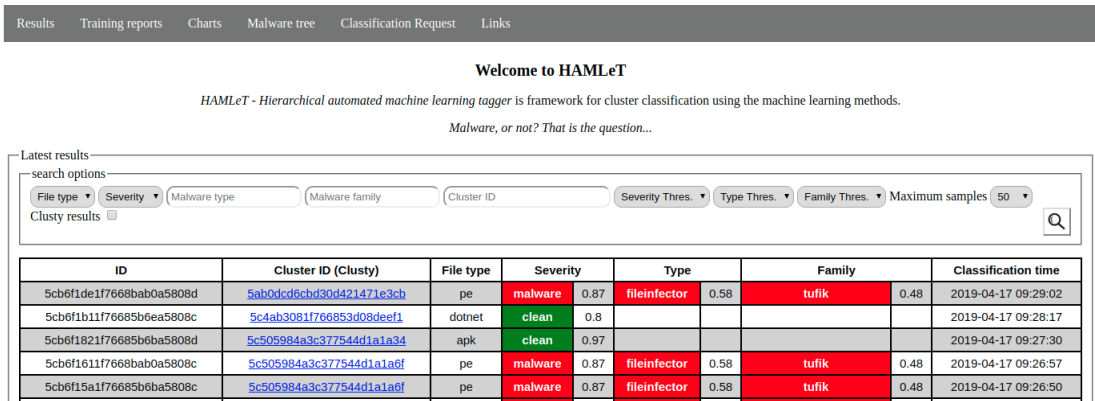


Figure 8.2: Hamlet – classification results

is the cluster classified by the classifier and the results are post-processed by the secondary rules. After that, the full classification tag is generated containing the classification time and saved to DB. User obtains JSON representing the classification.

Many internal services either generate or process the JSON data. When the classification request is made manually using the web interface and not API, user can easily copy the generated JSON and paste it into an input area. The input area already contains a cluster JSON template by default. Each time the data in the input area change, JSON is parsed and checked by Javascript functions. If the JSON data are valid, they are visualized on the right side of the page where supported keywords like file format properties are also highlighted. If the JSON is missing key information like properties or cluster ID, warning is shown in the bottom section and the form submit button is disabled until the data are valid. The interface for manual classification is shown in Figure 8.3.

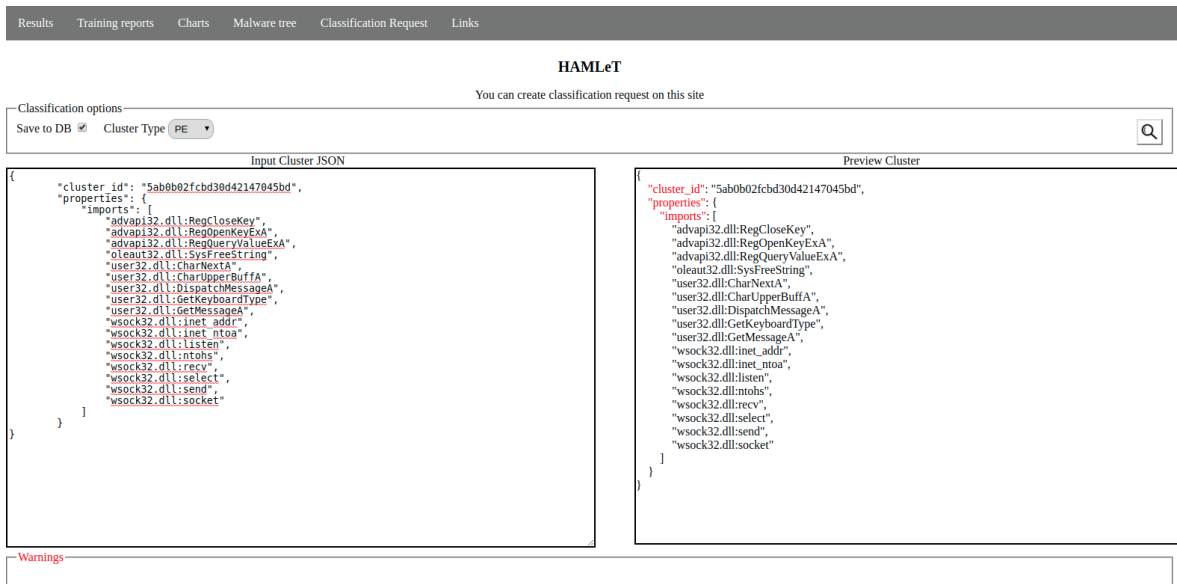


Figure 8.3: Hamlet – classification request interface

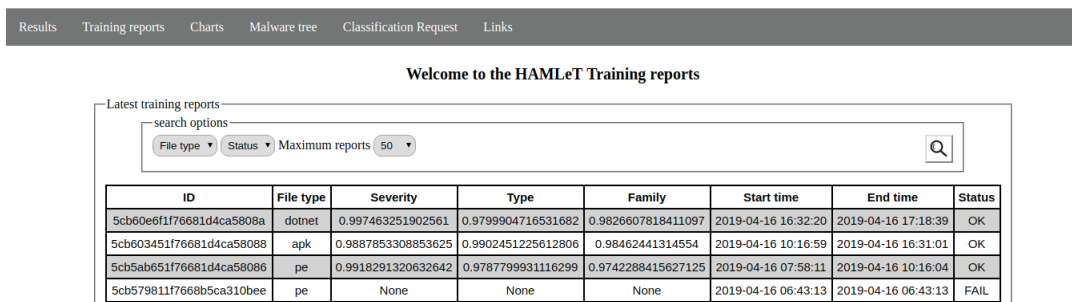
8.4 Training

To be able to make predictions, the classifiers need to be trained. The clusters are often changed or reclassified. Clusty is able to generate dump once a day into shared folder with Hamlet. The classifiers are trained each day at time specified in the configuration file. For this purpose a background scheduler is activated during the service initialization. This allows Hamlet to run a job similar to CRON jobs used on Unix operating systems. When the scheduler triggers an event, the main master worker is delegated to train the classifiers. The file format classifiers are trained sequentially. However, to train partial classifiers, from which they are composed, is used parallelism.

Supported properties for the given file formats are loaded during training of each classifier. The dump is validated by the function `validate_dump` provided by the `core.io` module. Name of the dump is generated by using a system date. If the file exists and dump is valid, function `generate_input_data` is used to load the information from dump and transform the data into format that is passed to classifiers. The function provides many parameters that might describe the loaded data, like specifying the classification type, minimal confidence, allowed severities, etc. After that process is the model trained and saved in the set destination.

When the training is completed, the main worker sends a signal to itself, which triggers update of all slave workers. Malware tree is also generated from the current dump and stored into database. If any error occurs during the training, the training information is stored into DB and training is scheduled on the next day because the problem would probably occur repeatedly on the same dump. Training information is saved separately for each file format and contains partial accuracy scores of severity, malware type and family classifiers achieved on the training dataset and the status whether the training was completed successfully. If the training process was not finished due to an error or problem, the accuracy of the partial classifiers shows value None. The reason can be easily found in the service logs. Time of the beginning and end of the training process is also saved.

User can also limit the number of shown reports or search by the status or file format. The training information interface is shown in Figure 8.4.



ID	File type	Severity	Type	Family	Start time	End time	Status
5cb60e6f1f76681d4ca5808a	dotnet	0.997463251902561	0.9799904716531682	0.9826607818411097	2019-04-16 16:32:20	2019-04-16 17:18:39	OK
5cb603451f76681d4ca58088	apk	0.9887853308853625	0.9902451225612806	0.98462441314554	2019-04-16 10:16:59	2019-04-16 16:31:01	OK
5cb5ab651f76681d4ca58086	pe	0.9918291320632642	0.9787799931116299	0.9742288415627125	2019-04-16 07:58:11	2019-04-16 10:16:04	OK
5cb579811f7668b5ca310bee	pe	None	None	None	2019-04-16 06:43:13	2019-04-16 06:43:13	FAIL

Figure 8.4: Hamlet – training reports

8.5 Malware tree

The malware tree is generated by function `core.io.construct_malware_tree` once the scheduled training was completed successfully and classifiers are updated. The malware

tree is a hierarchical tree structure, which is constructed from the dump, containing three layers: severity, malware types and families. Malware tree is generated using all names that are present in the dump. The set of all existing labels differs from the classified ones due to application of confidence threshold when the data are loaded into memory. When the family is represented by a few samples with very low confidence, they were not used for training and are not classified by Hamlet. User can access the supported and classified severities, malware types and families through the *Malware tree* option in the main menu. The handling routine obtains the classes classified by the currently used trained instances of classifiers in the memory, loads the last malware tree for the given file format from the database and subsequently highlights any names that occur in the tree. Example is shown in Figure 8.5. If the classifiers are not trained yet, the trained objects do not contain attribute `classes_` and the generated tree is blank. This functionality is not useful only for observation of Hamlet classification capabilities, but also for identification of the total number of families or lookup of all malware types that contain searched malware family.

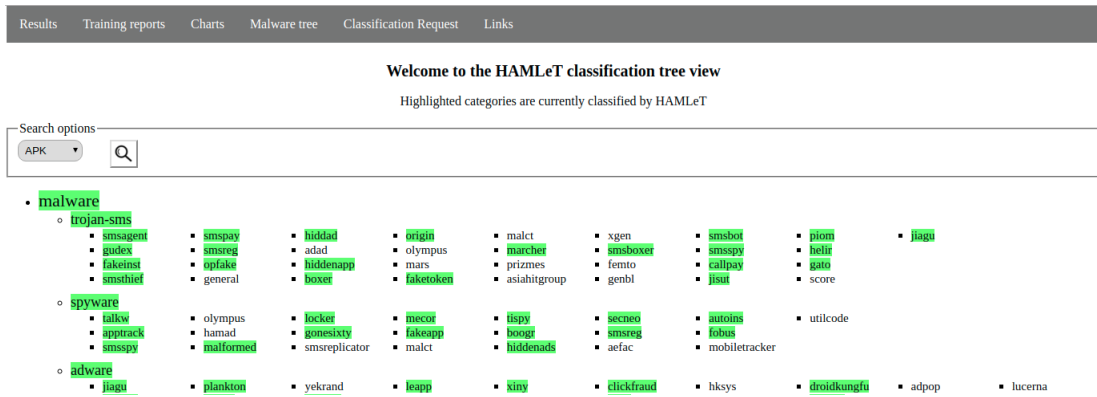


Figure 8.5: Hamlet – malware tree

8.6 Charts

Visualization of classification charts is an additional functionality, which allows the users to see classification statistics from the latest reports. Each chart represents the changing values of recorded properties in time. Three main properties were chosen to be visualized: total number of classifications, number of classifications per file format and malware types. When the classification result is saved to database, it is immediately counted into new generated charts. Users can access chart interface from the main menu. The charts show the latest results and user can specify time window and time step, from which are the charts generated. Generated charts are shown in Figure 8.6.

8.7 Logging

Logging interface is the only one not accessible from the main menu. To access the logging interface, user has to specifically request `server/log` endpoint. Both the user interface and API are protected by credentials. The credentials are specified in the configuration file and loaded during the service initialization. After accessing the log interface, user can see

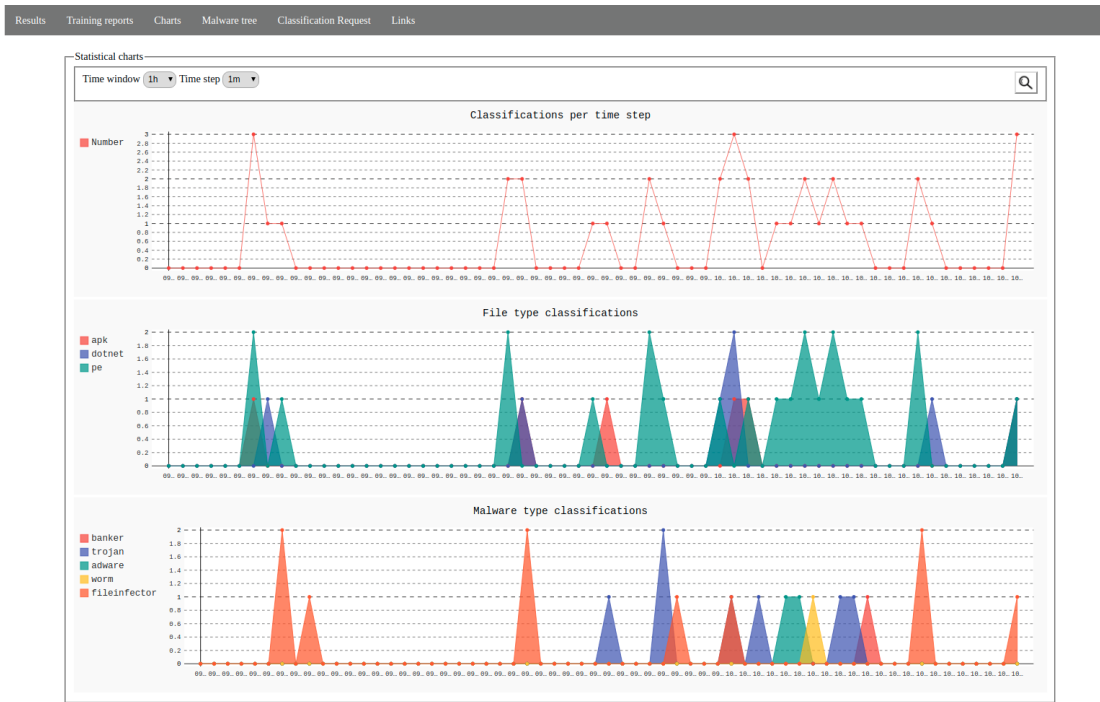


Figure 8.6: Hamlet – generated charts

the latest reports of the logs, select the active log and also select the type of reports to be shown (Figure 8.7). Python logging interface supports multiple priorities of the event logs from DEBUG with the lowest priority to CRITICAL with the highest. Error reports are distinguished from others using the red color, so they are easily identified. Logging reports are sorted by the logged date and time.

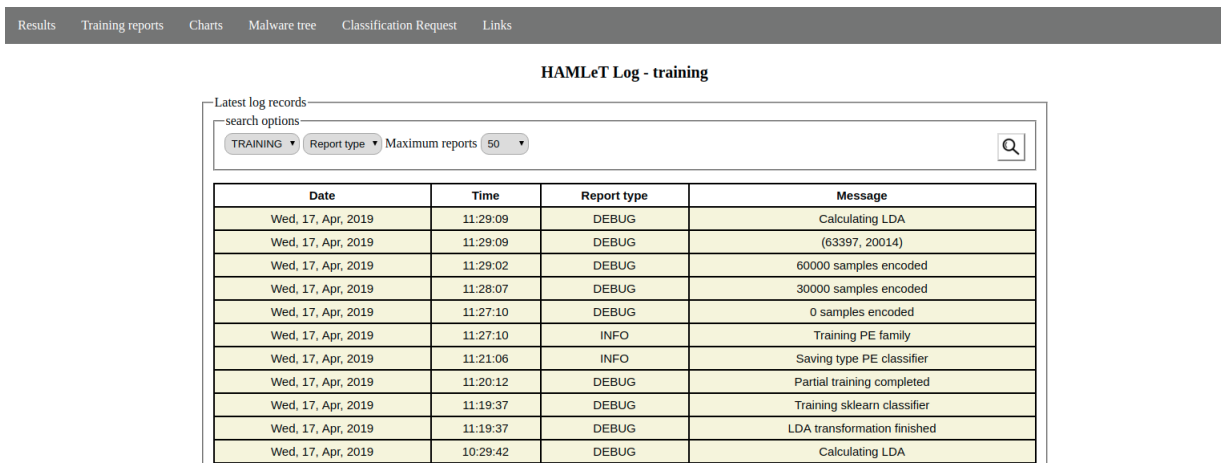


Figure 8.7: Hamlet – logging interface

Chapter 9

Testing

The implemented web service was properly tested with a set of unit and integration tests to verify and validate its functionality. Both types of tests focus on different areas of functionality and are a common combination for testing of production services. All tests were implemented using the Python 3 module *unittest* providing a high level interface for implementation of test cases. All tests are fully automated and do not need user's interaction. After the execution of tests is completed, a short statistic of passing and failing tests is printed to the standard output.

9.1 Unit tests

Unit tests focus on testing the functionality of specific modules, classes or functions. They do not test the overall functionality of application and cooperation between modules [40]. All tests are sorted into categories and can be run separately for each category or as a whole set. Basic description of each category is shown in Table 9.1. Unit tests do not need configuration file information, any access to external services or other servers. Manually chosen data and files that are stored in the testing directory are be used if any test needs a sample, cluster information or a small dump to work with. Total number of implemented unit tests is 101.

category	description
core	Core functionality & input, output
classification	Performing classification & predictions estimation
apiscout_qr	Apiscout image generators and converters
visual	Plotting, chart generation & animation
data_preprocessing	Preprocessing of dataset and properties
web	Web utils and handling functions

Table 9.1: Overview of unit test categories

9.2 Integration tests

Integration tests are in comparison with unit tests meant to verify the integration of modules and various parts of the service to function properly using the real database access and web

interface [40]. They load all credentials and necessary information from the configuration file. Many of them require to start the whole service as a sub-process that is terminated after the test run. A specific port different from the one used by production version is reserved for this purpose. Execution of integration tests takes longer time due to initialization of the whole service and connection to external services. Tests focused on the web interface are executed using Firefox¹ browser, which is downloaded and set by an automated script together with webdriver responsible for communication between the browser and Selenium², testing framework for the browser automation. A basic description of each category is shown in Table 9.2. Total number of implemented integration tests is 97.

category	description
actions	Web interface transactions
db	Database modules & connection
web_api	API requests and JSON validation
web_interface	Web interface, forms and visualization

Table 9.2: Overview of integration test categories

¹<https://www.mozilla.org/en-US/firefox/new/>

²<https://www.seleniumhq.org/>

Chapter 10

Conclusion

The goal of this thesis was to estimate whether malware classification is possible on the cluster-level using the machine learning methods. The results described in Chapter 6 show that the malware classification of PE, APK and .NET clusters is possible with lower accuracy than common file-level classification approaches. However, the machine learning methods provide one layer of detection behind other approaches that warns about suspicious clusters and the achieved results were satisfactory. Especially effective classification machine learning methods were random forests and neural networks. Comparison with meta-learning framework has shown that during experimentation proper classification models were designed, which were also validated using multiple different dumps. Classification of cluster severity was generally more accurate than classification of malware types and families.

The best classification results were achieved for classification of APK clusters. This might be caused by the higher number of missing features in clusters of other file formats. Extraction of properties that are not shared by all files in the cluster should increase the classification accuracy, for example, calculation of the mean file size value of cluster. Better validation of the current classifications made by Clusty would lower the irreducible error of the current datasets. The experimentation was also limited by the system resources. Linear discriminant analysis was not applicable on larger datasets or for severity classification. Because this method has significantly increased the accuracy for malware type and family classification, we can expect the increase for the severity classification as well. With more memory it would be also possible to increase the size of feature vectors containing the values of hashed cluster properties, which has resulted into increased accuracy as well.

To provide a classification interface for users and external services, a web classification service called Hamlet was implemented, which uses the proposed machine learning models for the classification of clusters. This service allows users to classify clusters, see the classification results, generate charts representing the latest classification trends or to access the service logs. Functionality of the web service was also properly verified by the set of automated unit and integration tests.

Hamlet is currently used as a weak classification system of Clusty when the classification using YARA rules fails. The confidence of classification result is recalculated based on confidence of severity, malware type and family classification to achieve lower false positive numbers and also to properly compare the confidence of classifications with other classifiers.

Bibliography

- [1] *Application Fundamentals*. Android developers. [online; cit. 14.11.2018]. Retrieved from: <https://developer.android.com/guide/components/fundamentals>
- [2] *Operating System Market Share*. NetMarketShare. NetApplications. [online; cit. 18.3.2019]. Retrieved from: <https://netmarketshare.com/>
- [3] *The devil's in the Rich header*. Global Research and Analysis Team. Kaspersky Lab. [online; cit. 18.3.2019]. Retrieved from: <https://securelist.com/the-devils-in-the-rich-header/84348/>
- [4] *Investigation: WannaCry cyber attack and the NHS*. National Audit Office. Apr 2018. [Online; cit. 17.10.2018]. Retrieved from: <https://www.nao.org.uk/wp-content/uploads/2017/10/Investigation-WannaCry-cyber-attack-and-the-NHS.pdf>
- [5] *PE Format*. Microsoft Corporation. 2018. [online; cit. 12.12.2018]. Retrieved from: <https://docs.microsoft.com/en-us/windows/desktop/Debug/pe-format>
- [6] Alvarez, V. M.: *YARA - The pattern matching swiss knife for malware researchers*. [Online; cit. 14.2.2019]. Retrieved from: <http://virustotal.github.io/yara/>
- [7] Attenberg, J.; Weinberger, K.; Dasgupta, A.; et al.: Collaborative Email-Spam Filtering with the Hashing Trick. In *Sixth Conference on Email and Anti-Spam (CEAS)*. Jan 2009.
- [8] Chawla V., N.; W. Bowyer, K.; Lawrence O. Hall, L.; et al.: *SMOTE: Synthetic Minority Over-sampling Technique*. 2002. pp. 321–357.
- [9] Elisan, C. C.: *Advanced Malware Analysis*. McGraw-Hill Education. 2015. ISBN 978-0-07-181975-6.
- [10] Fajmon, B.; Hlavičková, I.; Novák, M.; et al.: *Numerická matematika a pravděpodobnost*. Ústav matematiky FEKT VUT v Brně. 2014. [online; cit. 4.4.2019]. Retrieved from: http://matika.umat.feec.vutbr.cz/inovace/texty/INM/CZ/INM_plna_verze_CZ.pdf

- [11] Feurer, M.; Klein, A.; Eggenberger, K.; et al.: Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems 28*, edited by C. Cortes; N. D. Lawrence; D. D. Lee; M. Sugiyama; R. Garnett. Curran Associates, Inc.. 2015. pp. 2962–2970.
Retrieved from: <http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf>
- [12] Gavriluț, D.; Cimpoesu, M.; Anton, D.; et al.: *Malware detection using machine learning*. Nov 2009. pp. 735 – 741. doi:10.1109/IMCSIT.2009.5352759.
- [13] Goppit: *Portable Executable File Format – A Reverse Engineer View*. 2006.
- [14] Géron, A.: *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. O’Reilly. 2017. ISBN 978-1-491-96229-9.
- [15] Guo, G.; Wang, H.; Bell, D.; et al.: KNN Model-Based Approach in Classification. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*. Aug 2003. ISBN 978-3-540-39964-3.
- [16] Gurney, K.: *An introduction to neural networks*. Taylor & Francis e-Library. 2004. ISBN 0-203-45151-1.
- [17] Ho Park, S.; Goo, J. M.; Jo, C.-H.: Receiver operating characteristic (ROC) curve: practical review for radiologists. *Korean journal of radiology : official journal of the Korean Radiological Society*. vol. 5. Mar 2004: pp. 11–8. doi:10.3348/kjr.2004.5.1.11.
- [18] Hossin, M.; M.N, S.: A Review on Evaluation Metrics for Data Classification Evaluations. *International Journal of Data Mining & Knowledge Management Process*. vol. 5. Mar 2015: pp. 01–11. doi:10.5121/ijdkp.2015.5201.
- [19] Hrádek, I.: *Štruktúra APK súboru na OS Android*. Master’s Thesis. Masaryk University, Faculty of informatics. 2015. [online; cit. 15.01.2019].
Retrieved from: <https://is.muni.cz/th/uiuub/thesis.pdf>
- [20] Ivanović, M.; Radovanović, M.: *Modern machine learning techniques and their applications*. Jun 2015. ISBN 978-1-138-02830-2. pp. 833–846.
doi:10.1201/b18592-153.
- [21] Jolliffe, I.: *Principal Component Analysis*. Springer. 2002. ISBN 0-187-95442-2.
- [22] Kalash, M.; Ročan, M.; Mohammed, N.; et al.: Malware Classification with Deep Convolutional Neural Networks. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. Feb 2018. ISSN 2157-4960. pp. 1–5.
doi:10.1109/NTMS.2018.8328749.
- [23] Kennedy, D.; O’Gorman, J.; Kearns, D.; et al.: *Metasploit: The penetration tester’s guide*. William Pollock. 2011. ISBN 978-1-59327-288-3.
- [24] Koret, J.; Bachaalany, E.: *The antivirus hacker’s handbook*. John Wiley and Sons, Inc. 2015. ISBN 978-1-119-02875-8.
- [25] Kriesel, D.: *A Brief Introduction to Neural Networks*. 2007.
Retrieved from: <http://www.dkriesel.com>

- [26] Kruegel, C.: Full System Emulation: Achieving Successful Automated Dynamic Analysis of Evasive Malware. In *Proc. BlackHat USA Security Conference*. Lastline, Inc. 2014. pp. 1–7.
- [27] Li, J.; Cheng, K.; Wang, S.; et al.: Feature Selection: A Data Perspective. *ACM Computing Surveys*. vol. 50. Jan 2016. doi:10.1145/3136625.
- [28] Müller, A. C.; Guido, S.: *Introduction to Machine Learning with Python*. O’Reilly. 2017. ISBN 97814449369415.
- [29] Nakov, S.; et al.: *Fundamentals of computer programming in C#*. Faber Publishing. Bulgaria. 2013. ISBN 987-954-400-773-7.
- [30] Namanya, A. P.; Cullen, A.; Awan, I. U.; et al.: The World of Malware: An Overview. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*. Aug 2018. pp. 420–427. doi:10.1109/FiCloud.2018.00067.
- [31] Nwankpa, C. E.; Ijomah, W.; Gachagan, A.; et al.: *Activation functions: comparison of trends in practice and research for deep learning*. 2018. [online; cit. 14.1.2019]. Retrieved from: <https://arxiv.org/pdf/1811.03378.pdf>
- [32] Perlich, C.: *Learning Curves in Machine Learning*. Jan 2011. doi:10.1007/978-0-387-30164-8_452.
- [33] Raschka, S.: *Python Machine Learning*. Packt Publishing. 2015. ISBN 978-1783555130.
- [34] Rish, I.: An Empirical Study of the Naive Bayes Classifier. *IJCAI 2001 Work Empir Methods Artif Intell*. vol. 3. Jan 2001.
- [35] Rokach, L.; Maimon, O.: Decision Trees. *The Data Mining and Knowledge Discovery Handbook*. vol. 6. Jan 2005: pp. 165–192. doi:10.1007/0-387-25465-X_9.
- [36] Sami, A.; Yadegari, B.; Peiravian, N.; et al.: Malware detection based on mining API calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC ’10. USA: ACM. 2010. ISBN 978-1-60558-639-7.
- [37] Shahzad, R. M. K.: *Classification of potentially unwanted programs using supervised learning*. Blekinge Institute of Technology. Sweden. 2013. ISBN 978-91-7295-247-8.
- [38] Tharwat, A.; Gaber, T.; Ibrahim, A.; et al.: Linear discriminant analysis: A detailed tutorial. *Ai Communications*. vol. 30. May 2017: pp. 169–190,. doi:10.3233/AIC-170729.
- [39] Walck, C.: *Hand-book on statistical distributions for experimentalists*. Particle Physics Group Fysikum. University of Stockholm. Stockholm. 2007. [online; cit. 10.2.2019]. Retrieved from: <http://www.stat.rice.edu/~dobelman/textfiles/DistributionsHandbook.pdf>
- [40] Zemek, P.: *What is Clusty*. Internal Avast documentation. [Online; cit. 30.04.2019].

Appendix A

Contents of the DVD

- **src/** - Source code of the application
- **tests/** - Unit and integration tests
- **data/** - Samples of JSON dumps
- **docs/** - Folder into which documentation is generated
- **bt_documentation/** - Source and PDF file of technical report
- **requirements.txt** - List of all required Python packages
- **config.ini** - Configuration file
- **Makefile**
- **README.md**