POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Machine Learning for malware characterization and identification

Supervisor
prof. Antonio Lioy
prof. Andrea Atzeni

Candidate
Francesco Romano

October 2023

Ai miei genitori che mi supportano e spingono sempre ad impegnarmi e a dare il meglio.

Summary

During the last few years, we have seen malware's spread have an exponential growth, according to AV Atlas Dashboard [1] in 2021 there was a growth in the total number of malwares of 116,946,859 ones and 4,710,843 PUA (Potential Unwanted Application) under Windows. This year, instead the total number of new malwares and PUA is 97,050,954 with a daily growth of 269,157. Cybercriminals are always working on finding new ways to get around controls in an attempt to make their attacks successful. The success of an attack entails direct and indirect damage to enterprises. The major impacts of malware on enterprises are the interruption and disabling of services, breakdowns of the entire enterprise network infrastructure, loss of control of all applications in execution, disclosure of sensitive information, and reputational damage.

Malware detection through classic methods like signature ones, detection and prevention system, or antivirus software is becoming ever-more difficult, this is due to the evolution of recent malware based on multiple polymorphic layers that elude detection or automatic mechanisms updating themself in short periods eluding the detection by any antivirus software. The struggle between security analysts and malware developers is a never-ending battle with the complexity of malware changing as quickly as innovation grows. Current state-of-the-art research focuses on the development and application of machine learning techniques for malware detection due to its ability to keep pace with malware evolution. Several methodologies to automatically detect malwares through machine learning models have been proposed to cope with the malware during the last decade like Random Forest, Support Vector Machine (SVM), Decision Tree, AdaBoost, Gaussian Naive Bayes (GNB), and Gradient Boosting. The work of this thesis focuses on the classification of malware families using only network traffic as information.

The first step was the search for a dataset that was suitable for the task and whose goodness was sufficient for the purpose of best training the model. Subsequently, a profound study of the art was conducted with respect to the topic to be addressed and unfortunately, even here not many ideas were found but enough to be able to move on to the next phase of data pre-processing and data mining. This was the longest first phase of the whole thesis process as it needed to compare each single network packet with the others of the same and other files. After this phase, three different machine learning models were created which are Random Forest, Gradient Boosting, and K-nearest neighbors. During the experimentation phase on the models, various situations of difficulty arose in being able to correctly manage all the families of malware and above all their evolution over the years. With the use of evidence from the data mining phase, it was demonstrated how important that phase is since the classification results underwent a significant increase, which however was not enough, which is why re-elaborations of the dataset were applied and the use of the Repeated Stratified K-Fold cross validator managing to obtain the best results in the classification of malware families and the management of their evolution with the model having the Random Forest algorithm. The results therefore proved to be fully sufficient for the purpose of classifying malware in a large period, just under ten years, promptly managing their evolutions, thus expanding research on the state of the art and constituting a valid solution for problems in this era.

Ringraziamenti

Vorrei porrei i miei più sinceri ringraziamenti al Professore Lioy e al Professore Atzeni per avermi dato l'opportunità di lavorare a questa tesi, per l'aiuto che mi hanno dato durante tutto il percorso e per la grande disponibilità.

Vorrei poter ringraziare i miei genitori, se sono riuscito a raggiungere quest'importante obiettivo è grazie a tutti i sacrifici che hanno fatto per me. Vi ringrazio per aver creduto in me e per avermi esortato a non arrendermi davanti alle difficoltà. Spero che un giorno riuscirò a ripagare tutti i vostri sforzi rendendovi fieri di aver cresciuto un figlio come me.

Colgo l'occasione per ringraziare mia sorella, anche se siamo da sempre come il giorno e la notte, ti ringrazio di essermi stato vicino ed aver sostenuto il mio percorso.

Vorrei ringraziare i miei parenti che nei momenti di difficoltà sono sempre stati un porto sicuro. Vorrei ringraziarvi inoltre per la fiducia che avete sempre avuto in me che mi motiva nel perseguire gli obiettivi che mi prefisso.

Ringrazio il mio amico Biagio, per te che è difficile esternare i sentimenti, ti ringrazio per avermi supportato. Abbiamo scoperto che ciò che è rotto può essere riforgiato, così come la nostra amicizia che ha acquisito maggior valore condividendo dolori e frustrazioni ma anche momenti di grande gioia e passioni.

Vorrei porre uno speciale ringraziamento alla mia amata Clarissa, con te ho scoperto cosa voglia dire amare e cosa essere amati. Ti ringrazio per essermi stata sempre vicina, anche nei momenti più grigi, e per aver migliorato i miei lati peggiori contribuendo nella formazione della persona che sono ora. Spero di poter esserti d'aiuto quanto tu lo sia stata per me.

Ringrazio il mio amico Vincenzo che mi ha seguito come un fratello minore sin dall'adolescenza ed esser stato la mia valvola di sfogo in palestra e non. Grazie a te ho appreso la via della cedevolezza e che si arriva in alto superando sé stessi.

Infine, ringrazio tutti i miei amici e colleghi per l'apporto che hanno dato nell'alleggerire lo stress del lavoro e degli studi nonché ad essere fonte d'ispirazione.

Contents

Chapter 1

Introduction

In the ever-evolving digital landscape, cybersecurity plays a pivotal role in safeguarding our information, digital assets, and computer systems. Although there are many proactive actions, the possibility that a cyber-attack can occur is never zero. For these reasons, cybersecurity is in an endless battle with the attackers, it is based on the study of new attacks, which have bypassed the current defenses, reinforcing them in order to avoid similar attacks again.
Nowadays malwares represents one of the most insidious cyber threats of the digital society. There are many kinds of malware, each of them designed with its own purpose; compromise, damage, or illicitly access other people's computer systems, sensitive data, and digital resources. Malware developers, thanks to the evolution of technology and global interconnectivity, in recent years have obtained a wide range of opportunities to propagate their attacks. Malware can be spread through email attachments, compromised websites, infected USB devices, and even through computer networks. Their ability to change quickly and hide between the normal operations of the system makes them even more insidious and difficult to detect.
In the last decade the growth of malware has been remarkable, an increase of 1044% with an average growth of about 60,000,000 malware per year (Figure 1.1). If the annual data can be

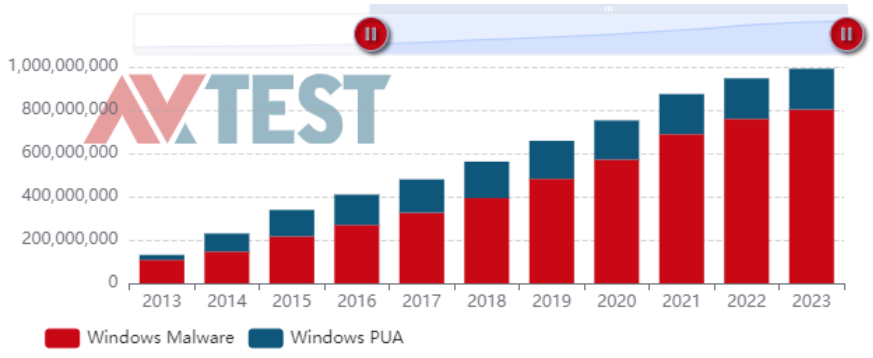

Figure 1.1. Total amount of malware and PUA under Windows.

difficult to imagine in reality just think that every second they spread about 3.1 malware, 11,078 per hour and 265,893 per day (Figure 1.2). Data cited are shared by the web portal of the AV-TEST Institute, which provides numerous cybersecurity data [1]. Another important thing to focus on is their economic impact, according to IBM researches [2], the global average cost of a data breach in 2022 was the highest ever since the dawn of conducting these reports. The cost of a data breach in 2022 was $4.35M - a 12.7% increase compared to 2020 when the cost was $3.86M (Figure 1.3). Should be specified that already in 2023, thanks to the awareness, the profits related to ransomware have fallen compared to the previous year according to Malwarebytes [3] (Figure 1.4). In addition to the increase in malware, we must also consider their evolution relative to complexity. Malware developers have also developed evasion and obfuscation techniques in response

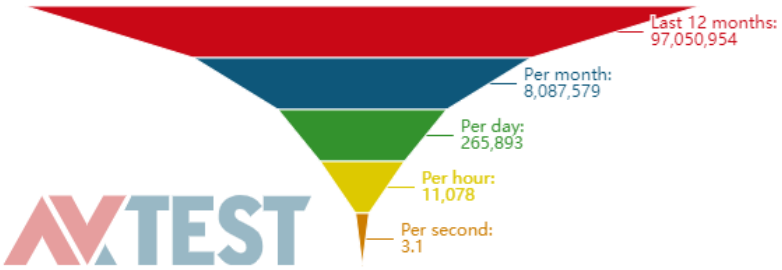

Figure 1.2. New malware and PUA per second.

Figure 1.3. Average cost of a data breach in recent years.

to detection techniques, the most famous are polymorphism, metamorphism, and obfuscation. In the final part of Chapter 2, all the evasion techniques already known are deeply explained.
Malware detection is a primary countermeasure and there are several techniques used to identify a malware in a system or in the network, among which: signature-based detection, heuristic analysis, and behavioral analysis. In addition to detecting malware, it is increasingly appropriate to also carry out a classification of it. The classification is necessary to warn the analyst of the possible presence of that family of malware for which mitigation and prevention actions can be implemented instead of others. The malware detection and classification techniques are widely discussed in Chapter 3, highlighting the positive and negative sides of each of them with the analysis of related articles.
In recent years, machine learning has become increasingly important in a wide variety of sectors. Significant improvements in the ability to collect, process, and analyze large amounts of data, as well as improvements in computer computing power, have been the driving forces behind this development.
The goal of using machine learning in cybersecurity is to have a greater proactive force in order to reduce the workload of security analysts, thus making the battle between them and the malware developers more feasible. Due to the large amount of malware data that can be collected and processed, the use of machine learning has recently gained popularity in cybersecurity.
In the last years, many papers have been published with the aim to detect and classify malwares proposing new and performing algorithms. The aim of this thesis is to obtain a good tool that can manage the evolution of the malware families over the years in order to have a proactive defense implemented on end systems or network probes.
The development process of the thesis is based on the following stages: research of the dataset,

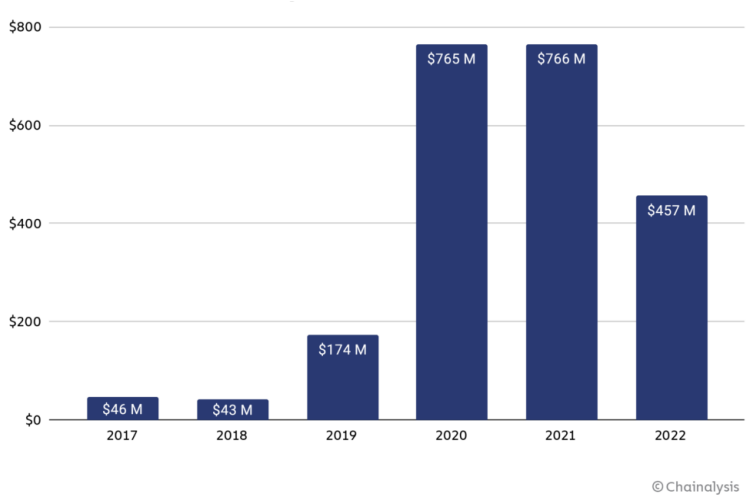

Figure 1.4. Total Value received by ransomware attackers, 2017 - 2022.

data pre-processing and data mining, implementation of machine learning algorithms, iterative cycle of model refinement based on previous results, rework of the dataset, and analysis two at once of the families, another one iterative cycle of model refinement for each pair, comparison with cross-validation, evaluation of machine learning models.

The first phase was dedicated to the search for datasets that were composed solely of network packets and whose contents were sufficient for training a machine learning model. From the outset, there were difficulties in finding public datasets that contained only this type of information and that, above all, were reliable and of acceptable size. A first round of research produced the CTU-13 dataset, a dataset based on 13 samples of malware (which malware families are: Neris, Rbot, Virut, Menti, Sogou, Murlo, NSIS.ay) made by Stratosphere IPS, as results, but which was later discarded and a custom dataset took over, created by merging individual network traffic captures from systems infected by Trickbot, Ramnit, and Dridex always having Stratosphere IPS as sources but also the malware-traffic-analysis blog.

Subsequently, the pre-processing and data mining phases were carried out which, respectively, first processed the packets by extrapolating the most important features and reassembled the flows, and then calculated the correlation of the packets of each capture in order to highlight which were the data that is most important for training machine learning models.

Finally, an experimental iterative cycle was carried out in which phases of division and/or remodulation of the dataset, configuration, and tuning of the machine learning models, training and classification, analysis of the results, and optimization for the next iteration followed. The objective of this iterative cycle was to obtain the best results of the metrics produced by the implemented models, learning from the successes and errors received at each stage. The machine learning algorithms that have been used in this phase are Random Forest, Gradient Boosting, and K-Nearest Neighbors, all the models have obtained good metric results during the course of the experimentation but in particular, the Random Forest algorithm was the best of the three for this type of task to classify a malicious flow through network packets due to a particular family of malware, which could have developed significant evolutions over time. The last two chapters deal with, respectively in order, the results of the most significant iterations recorded during the experimentation phase and finally the conclusions I reached once the study was finished.

Chapter 2

Background

Is not that easy to describe malware, this is due to the fact that malware types continue to evolve, and that makes them vary, a simple definition could be: Malware, or "malicious software", is an umbrella term that describes any malicious program or code that is harmful to systems. Hostile, intrusive, and intentionally nasty, malware seeks to invade, damage, or disable computers, computer systems, networks, tablets, and mobile devices, often by taking partial control over a device's operations [4].
Also, the NIST (National Institute of Standards and Technologies) needs to revise the malware definition keeping traces of all new types. One of the most recent revisions is this: Software or firmware intended to perform an unauthorized process that will have adverse impacts on the confidentiality, integrity, or availability of a system. A virus, worm, Trojan horse, or other code-based entity that infects a host. Spyware and some forms of adware are also examples of malicious code [5].

2.1 Types of malwares

Today, malwares are not more based on a single type, in most of the cases they are a combination of different types of malicious software. In this paragraph are exposed the main type of malware with some real examples.

2.1.1 Virus

A virus is a piece of code that inserts itself into an application and executes when the app is run. Once inside a network, a virus may be used to steal sensitive data, launch DDoS attacks, or conduct ransomware attacks. Usually spread via infected websites, file sharing, or email attachment downloads, a virus will lie dormant until the infected host file or program is activated. Once that happens, the virus can replicate itself and spread through your systems.
Virus example:

- Stuxnet appeared in 2010 and was widely believed to have been developed by the US and Israeli governments to disrupt Iran's nuclear program. Spread via a USB thumb drive, it targeted Siemens industrial control systems, causing centrifuges to fail and self-destroy at a record rate. It is believed that Stuxnet infected over 20,000 computers and ruined one-fifth of Iran's nuclear centrifuges - setting its program back years.

2.1.2 Worms

One of the most common types of malwares, worms, spread over computer networks by exploiting operating system vulnerabilities. A worm is a standalone program that replicates itself to infect

other computers without requiring action from anyone. Since they can spread fast, worms are often used to execute a payload piece of code created to damage a system. Payloads can delete files on a host system, encrypt data for a ransomware attack, steal information, delete files, and create botnets.
Worm example:

- SQL Slammer was a well-known computer worm that did not use traditional distribution methods. Instead, it generated random IP addresses and sent itself out to them, looking for those not protected by antivirus software. Soon after it hit in 2003, the result was more than 75,000 infected computers unknowingly involved in DDoS attacks on several major websites. Though the relevant security patch has been available for many years now, SQL Slammer nevertheless experienced a resurgence in 2016 and 2017.

2.1.3 Trojans

A Trojan (or Trojan Horse) disguises itself as legitimate software to trick you into executing malicious software on your computer. Because it looks trustworthy, users download it, inadvertently allowing malware onto their devices. Trojans themselves are a doorway. Unlike a worm, they need a host to work. Once a Trojan is installed on a device, hackers can use it to delete, modify, or capture data, harvest your device as part of a botnet, spy on your device, or gain access to your network.
Trojan examples:

- Qbot malware, also known as "Qakbot" or "Pinkslipbot", is a banking Trojan active since 2007 and focused on stealing user data and banking credentials. The malware has evolved to include new delivery mechanisms, command and control techniques, and anti-analysis features.
- TrickBot malware -first identified in 2016- is a Trojan developed and operated by sophisticated cybercrime actors. Originally designed as a banking Trojan to steal financial data, TrickBot has evolved into modular, multi-stage malware that provides its operators with a full suite of tools to carry out numerous illegal cyber activities.

2.1.4 Spyware

Spyware is a form of malware that hides on your device, monitors activity, and steals sensitive information like financial data, account information, logins, and more. Spyware can spread by exploiting software vulnerabilities or else be bundled with legitimate software or in Trojans.
Spyware examples:

- CoolWebSearch - This program took advantage of the security vulnerabilities in Internet Explorer to hijack the browser, change the settings, and send browsing data to its author.
- Gator - Usually bundled with file-sharing software like Kazaa, this program monitors the victim's web surfing habits and uses the information to serve them with specific ads.

2.1.5 Adware

Adware, a contraction of "advertising-supported software", displays unwanted and sometimes malicious advertising on a computer screen or mobile device, redirects search results to advertising websites, and captures user data that can be sold to advertisers without the user's consent. Not all adware is malware, some are legitimate and safe to use.
Users can often affect the frequency of adware or what kinds of downloads they allow by managing the pop-up controls and preferences within their internet browsers or using an ad blocker.
Adware examples:

- Fireball - Fireball hit the headlines in 2017 when an Israeli software company discovered that 250 million computers and one-fifth of the corporate networks worldwide were infected with it. When Fireball affects your computer, it takes over your browser. It changes your homepage to a fake search engine - Trotus - and inserts obtrusive ads into any webpage you visit. It also prevents you from modifying your browser settings.
- Appearch - Appearch is another common adware program that acts as a browser hijacker. Usually bundled with other free software, it inserts so many ads into the browser that web browsing becomes very difficult. When you attempt to visit a website, you are taken to Appearch.info instead. If you manage to open a web page, Appearch converts random blocks of text into links, so when you select the text, a pop-up invites you to download software updates.

2.1.6 Ransomware and crypto-malware

Ransomware is malware designed to lock users out of their system or deny access to data until a ransom is paid. Crypto-malware is a type of ransomware that encrypts user files and requires payment by a specific deadline and often through a digital currency such as Bitcoin. Ransomware has been a persistent threat to organizations across industries for many years now. As more businesses embrace digital transformation, the likelihood of being targeted in a ransomware attack has grown considerably.
Ransomware examples:

- CryptoLocker is a form of malware prevalent in 2013 and 2014 that cybercriminals used to gain access to and encrypt files on a system. Cybercriminals used social engineering tactics to trick employees into downloading the ransomware onto their computers, infecting the network. Once downloaded, CryptoLocker would display a ransom message offering to decrypt the data if a cash or Bitcoin payment was made by the stated deadline. While the CryptoLocker ransomware has since been taken down, it is believed that its operators extorted around three million dollars from unsuspecting organizations.
- Phobos malware - a form of ransomware that appeared in 2019. This strain of ransomware is based on the previously known Dharma (aka CrySis) family of ransomware.

2.1.7 Fileless malware

Fileless malware is a type of malicious software that uses legitimate programs to infect a computer. It does not rely on files and leaves no footprint, making it challenging to detect and remove. Fileless malware emerged in 2017 as a mainstream type of attack, but many of these attack methods have been around for a while.
Without being stored in a file or installed directly on a machine, fileless infections go straight into memory, and the malicious content never touches the hard drive. Cybercriminals have increasingly turned to fileless malware as an effective alternative form of attack, making it more difficult for traditional antivirus to detect because of the low footprint and the absence of files to scan.
Fileless malware examples:

- Frodo, Number of the Beast, and The Dark Avenger were all early examples of this type of malware.

2.1.8 Keyloggers

A keylogger is a type of spyware that monitors user activity. Keyloggers can be used for legitimate purposes - for example, families who use them to keep track of their children's online activity or organizations that use them to monitor employee activity. However, when installed for malicious purposes, keyloggers can be used to steal password data, banking information, and other sensitive information. Keyloggers can be inserted into a system through phishing, social engineering, or

malicious downloads.
Keylogger example:

- In 2017, a University of Iowa student was arrested after installing keyloggers on staff computers to steal login credentials to modify and change grades. The student was found guilty and sentenced to four months in prison.

2.1.9 Bots and botnets

A bot is a computer that has been infected with malware so it can be controlled remotely by a hacker. The bot - sometimes called a zombie computer - can then be used to launch more attacks or become part of a collection of bots called a botnet. Botnets can include millions of devices as they spread undetected. Botnets help hackers with numerous malicious activities, including DDoS attacks, sending spam and phishing messages, and spreading other types of malwares.
Botnet examples:

- Andromeda malware - The Andromeda botnet was associated with 80 different malware families. It grew so large that it was at one point infecting a million new machines a month, distributing itself via social media, instant messaging, spam emails, exploit kits, and more. The operation was taken down by the FBI, Europol's European Cybercrime centre, and others in 2017 - but many PCs continued to be infected.
- Mirai - In 2016, a massive DDoS attack left much of the US East Coast without internet access. The attack, which authorities initially feared was the work of a hostile nation-state, was caused by the Mirai botnet. Mirai is a type of malware that automatically finds Internet of Things (IoT) devices to infect and conscripts them into a botnet. From there, this IoT army can be used to mount DDoS attacks in which a firehose of junk traffic floods a target's servers with malicious traffic. Mirai continues to cause trouble today.

2.1.10 PUP malware

PUPs - which stands for "potentially unwanted programs" - are programs that may include advertising, toolbars, and pop-ups that are unrelated to the software you downloaded. Strictly speaking, PUPs are not always malware - PUP developers point out that their programs are downloaded with their users' consent, unlike malware. However, it is widely recognized that people mainly download PUPs because they have failed to realize that they have agreed to do so. PUPs are often bundled with other more legitimate pieces of software. Most people end up with a PUP because they have downloaded a new program and did not read the small print when installing it - and therefore did not realize they were opting in for additional programs that serve no real purpose.
PUP malware example:

- Mindspark malware - this was an easily installable PUP that ended up on users' machines without them noticing the download. Mindspark can change settings and trigger behavior on the device without the users' knowledge. It is notoriously difficult to eliminate.

2.1.11 Logic bombs

Logic bombs are a type of malware that will only activate when triggered, such as on a specific date and time or on the 20th log-on to an account. Viruses and worms often contain logic bombs to deliver their payload (i.e., malicious code) at a pre-defined time or when another condition is met. The damage caused by logic bombs varies from changing bytes of data to making hard drives unreadable.
Logic bomb example:

- In 2016, a programmer caused spreadsheets to malfunction at a branch of the Siemens corporation every few years, so they had to keep hiring him back to fix the problem. In this case, nobody suspected anything until a coincidence forced the malicious code out into the open.

2.1.12 RAM Scraper

A RAM scraper is a type of malware that harvests the data temporarily stored in memory or RAM. This type of malware often targets point-of-sale (POS) systems like cash registers because they can store unencrypted credit card numbers for a brief period of time before encrypting them and then passing them to the back-end.
RAM Scraper example:

- PoSeidon Malware - it spread in 2015 and targeted point-of-sale through phishing campaigns exploiting the vulnerabilities of that time to gain access to the system. Once installed, the aim of the malware is to steal credit card information and other payment-related data so it scans the memory of the system and captures the sensitive data.
- FastPOS - is a RAM Scraper that spread in 2016, its targets were POS systems of hospitality businesses and retailers.

2.1.13 Crimeware

Crimeware is a class of malware designed to automate cybercrime. It is designed to perpetrate identity theft through social engineering or stealth to access the victim's financial and retail accounts to steal funds or make unauthorized transactions. Alternatively, it may steal confidential or sensitive information as part of corporate espionage.

2.1.14 Rootkits

A rootkit is a collection of malware designed to give unauthorized access to a computer or area of its software and often masks its existence or the existence of other software. Rootkit installation can be automated, or the attacker can install it with administrator access.
Access can be obtained as a result of a direct attack on the system, such as exploiting vulnerabilities, cracking passwords, or phishing.
Rootkit detection is difficult because it can subvert the antivirus program intended to find it. Detection methods include using trusted operating systems, behavioral methods, signature scanning, difference scanning, and memory dump analysis.
Rootkit removal can be complicated or practically impossible, especially when rootkits reside in the kernel. Firmware rootkits may require hardware replacement or specialized equipment.
Rootkits example:

- TDL Rootkit - The TDL rootkit, or Alureon, is a family of rootkits that started to spread in 2008 and its activeness is still present. Developed by an anonymous group of cybercriminals, the nature of the compromised data by this malware was broad and was used for different purposes such as click fraud and distribution of other types of malware.

2.1.15 Backdoor

A backdoor is a covert method of bypassing normal authentication or encryption in a computer, product, embedded device (e.g., router), or other part of a computer.
Backdoors are commonly used to secure remote access to a computer or gain access to encrypted files. From there, it can be used to gain access to, corrupt, delete, or transfer sensitive data. Backdoors can take the form of a hidden part of a program (a trojan horse), a separate program, or code in firmware and operating systems. Further, backdoors can be created or widely known. Many backdoors have legitimate use cases such as the manufacturer needing a way to reset user passwords.

2.2 Malware spread

The exponential growth of malware attacks is also caused by social engineering techniques used by cybercriminals with events of phishing or pretexting or baiting, a campaign to raise awareness of social engineering could be a little countermeasure to the big spread of malwares. The most common ways in which malware threats can spread include:

- Email: If your email has been hacked, malware can force your computer to send emails with infected attachments or links to malicious websites. When a recipient opens the attachment or clicks the link, the malware is installed on their computer, and the cycle repeats.
- Physical media: Hackers can load malware onto USB flash drives and wait for unsuspecting victims to plug them into their computers. This technique is often used in corporate espionage.
- Pop-up alerts: This includes fake security alerts that trick you into downloading bogus security software, which in some cases can be additional malware.
- Vulnerabilities: A security defect in software can allow malware to gain unauthorized access to the computer, hardware, or network.
- Backdoors: An intended or unintended opening in software, hardware, networks, or system security.
- Drive-by downloads: Unintended download of software with or without knowledge of the end-user.
- Privilege escalation: A situation where an attacker obtains escalated access to a computer or network and then uses it to launch an attack.
- Homogeneity: If all systems are running the same operating system and connected to the same network, the risk of a successful worm spreading to other computers is increased.
- Blended threats: Malware packages that combine characteristics from multiple types of malwares, making them harder to detect and stop because they can exploit different vulnerabilities.

2.3 Malware attacks

The number of malware attacks during the last years is quickly rising, in fact in 2021 there was an increase of about 10%. In parallel, during the same year, there were estimated damages amounted to around 6 thousand billion dollars, which corresponds to 4 times the Italian GDP.
The main victims affected by these kinds of attacks are the government and the military sector, the healthcare, the computer science sector, and the instruction sector, all of them are fundamental and represent the foundations of each country. The leaderboard for the most affected continents by the malware attacks is led by America, but the attackers are slowly losing interest in it, followed by Europe, with an increment of 5% in the last years, and in the third place there is Asia.
Taking a closer look at our country, in 2021 in Italy registered at least 42 million security events, which increased by 16% compared to the previous year. The main kind of attacks are the ones deploying malwares and creating botnets, causing a huge rise of 58% for the number of servers and devices compromised. It is clear this fact: the malware threat is real and concrete, creating the need for solutions that mitigate their risks and damages.

2.4 Detection evasion

As there are ways to detect malware, there are also ways to be undetected, just like in a game of thieves and cops. There are several techniques used by attackers that are leaving behind anti-malware vendors [6].

2.4.1 Encryption

Encrypted malware consists of an encryption algorithm, encryption keys, encrypted malicious code, and a decryption algorithm. The key and decryption algorithm are used to decrypt the malicious component in the malware [7]. It is composed of two main sections: a decryption loop and a main body. The decryption loop is capable of encrypting and decrypting the main body. The main body contains the code of the malware itself which is encrypted with simple algorithms like XOR or using complex and robust ones such as AES. Anti-malware solutions must decrypt the main body to get a valid signature and detect the malicious piece of software.

2.4.2 Packing

Packing is a technique that is used to encrypt or compress the executable file. Usually, a phase of unpacking is necessary to reveal the overall semantics of the packed malicious program.

2.4.3 Oligomophism

The purpose of this technique is to produce a different decryptor for every new infection. An additional improvement is that there are several decryptors that are randomly chosen making a new type of code on every instance. This technique can be detected but requires more time.

2.4.4 Polymorphism

A polymorphic malware is programmed to look different each time it is replicated while keeping the original code intact. Different from simple encryption, polymorphic malware can use an unlimited number of encryption algorithms, and in each execution, a part of the decryption code will change. Depending on the malware type, different malicious actions performed by the malware can be placed under the encryption operations. Usually, a transformation engine is embedded in the encrypted malware. Note that at any change the engine generates a random encryption algorithm. Then, the engine and malware are encrypted using the produced algorithm, and a new decryption key is connected to them [7].
Polymorphic malware is harder to detect as there is an unlimited number of new decryptors. The main feature of this technique is that the code constantly changes with every new variant. Code obfuscation is used to mutate the decryptor to produce a new version for another victim.

2.4.5 Metamorphism

It does not contain an encrypted part. However, it uses mutation engines to change the body on every compilation, rather than using cryptography for protecting the code. A metamorphic engine should consist of the following components:

- Disassembler
- Code analyzer
- Code transformer
- Assembler

To detect this type of technique a complex and robust engine including heuristics and behavior analysis has to be in place. However, as stated there is no solid approach in place that helps detect this type of malware.

2.4.6 Obfuscation

Hiding information to avoid being caught is a common practice among attackers, in order to defeat certain security devices like IDS or any other based on signature detection. Using encoding and manipulating strings is a practice that can easily bypass Snort signatures. For example, either replacing a / with a \or using Hex, Unicode, or UTF-8 can avoid detections. Moreover, using encryption to encode a whole session is a classic way to avoid being detected. The problem lies in the lack of the appropriate key to decrypt information.
Polymorphic code aims to mutate its code while maintaining its original algorithm. It is usually combined with a cipher/decipher module that is embedded in the code. Encrypting malware is the first stage to bypass signature-based solutions as they do not have readable information to compare. Likewise, metamorphic malware presents a novel approach as it improves obfuscation, it has to recognize, parse, and mutate its own body every time it wants to propagate. Obfuscation has helped polymorphic and metamorphic malware to bypass anti-malware solutions. Indeed, they have used several coding techniques, to achieve their goals, some of which are.

- Dead-Code insertion: It simply adds not effective instructions to a program to change its appearance, but its behavior remains intact.
- Register Reassignment: It consists of switching registers from one version to another.
- Subroutine Reordering: A subroutine is obfuscated and reordered in a random way giving n chances of variants.
- Instruction Substitution: Its objective is to replace the original code with others that are equivalent to the original.
- Code Transposition: It reorders the sequence of a set of instructions from the original code, either by using unconditional branches or based on independent instructions.
- Code Integration: A malware joins its code with a valid program by decompiling the original one and rebuilding it with infected instructions.

2.4.7 Fragmentation and Session Splicing

These are network attack evasion techniques that take advantage of a feature of IP protocol called packet fragmentation, which allows to handle packets of different sizes. This evasion technique affects security devices as they have to wait for the whole package to arrive and then analyze it.
In reality, evasion attacks are not as easy to exploit as theory leads to believe, in fact, it requires knowledge of the network in which to attack. Not only it is necessary to generate packets considered invalid by the IDS, but there could also be the need for evading pattern recognition mechanisms.
In this paragraph, we are going to discuss in particular two evasion techniques: session splicing and fragmentation attacks.
The first technique splits the attack payload into multiple small packets so that the IDS must reassemble the packet stream to detect the attack. A simple way of splitting packets is by fragmenting them, but an adversary can also simply craft packets with small payloads [8]. The "whisker" evasion tool calls crafting packets with small payloads "session splicing". By delivering data a few bytes at a time, string matching can be evaded: it is effective because it can be spread out over a long period of time, but it is also restricted to TCP protocol. In order to detect this type of attack, the IDS needs to monitor connections and context, reassembling packet streams to identify it, but for the IDS it is difficult to keep track of them.
The second technique is based on reassembly issued that exists within the IP layer, recognized in the RFC791. A field in the IP header allows systems to break individual packets into smaller ones, while the offset field acts as a marker, indicating where a fragment belongs in the context of the original packet. IDS systems do not reconstruct packets until all fragments have arrived, so they are susceptible to resource exhaustion. There are 3 types of IP fragmentation: Overlapping,

Timeout, and TTL.
In the first case, the attacker sends fragments of varying sizes out of order and in overlapping positions, causing a fragment conflict in the IDS, and the reassembled packets at the end-system can be different than what IDS sees (Figure 2.1).
In the second case, the victim has a timeout greater than the IDS, so it will wait until the new fragment arrives. The IDS will drop the right fragment and let pass the infected one covered inside a "good" fragment (Figure 2.2).
In the last one, the attacker sets the TTL to 1 only for some fragments, so he can send an infected fragment with this TTL with a normal one in order to make it stay in the IDS, then he will send another one and the infected one will be compressed between these two. The victim will only see the good fragments, ignoring that inside there is a false fragment (Figure 2.3).
For example, for a certain original flow representation ori = [150, 300, 350, 280, 500, 350, 150, 250, 500, 400], the noise vector is [10, -20, 5, 10, -30, 30, 20, -30, -5, 50], and the target representation is [160, 280, 355, 270, 470, 380, 170, 220, 495, 450]. To add noise to the original traffic, firstly insert a packet with a length of 280 between the 1st and 2nd packets. Then insert a packet with a length of 270 between the 2nd and 3rd packets. Lastly insert four data packets with lengths of 170, 220, 495, and 450 after the 4th packet. Now the flow representation becomes [150, 280, 300, 270, 350, 280, 170, 220, 495, 450]. Then append extra bytes with the lengths of 10, 55, 120, and 100 to the 1st, 3rd, 5th, and 6th packets to get the noise representation. There is a total amount of data with a size of 2170 added into the original flow.

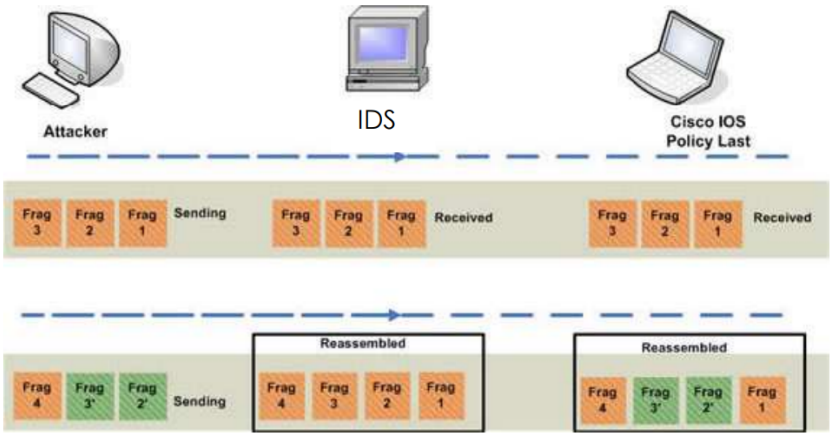

Figure 2.1. IP Fragmentation Overlap attack: first of all, the attacker sends some "noise" fragments to avoid matching form IDS, then he will send duplicated fragments that IDS could not discard.

2.4.8 Code reuse attacks

Data Execution Prevention is a mitigation used to prevent the execution of code from the non-executable segments, however, the attackers may not need to inject a shellcode, instead they can borrow pieces from the target program. Definitely, this attack is much more complex to perform than using shellcode.
There are two ways to perform code reuse attacks, one crafting the gadgets and one using lib-c. The attacker can craft its own sequences of meaningful instruction followed by a return. The gadget can be crafted using part of the program or using ad hoc tools like ropgadget and gdb extensions. It is possible to chaining the gadgets to write the shellcode.
in case the libc is available in the program, an alternative to build a chain of gadgets is to use return-to-lib-c. This leads to the research of the addresses of the functions that the attacker wants to call. Overall, this attack is a lot easier than building a ROP chain.
These attacks are orchestrated through sets of instructions and do not necessitate code injection or function calls, thereby evading existing anti-malware solutions.

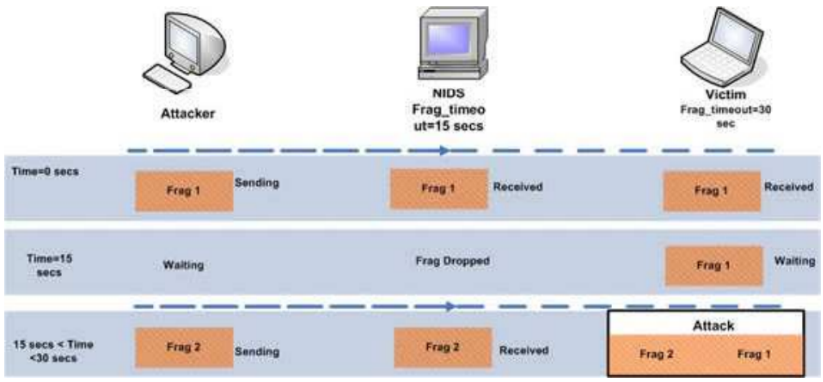

Figure 2.2. IP Fragmentation Timeout attack: the victim has a Timeout greater than the IDS. While the victim is waiting for the fragment, the IDS drops a fragment and the next one will pass with the infected fragment attached.

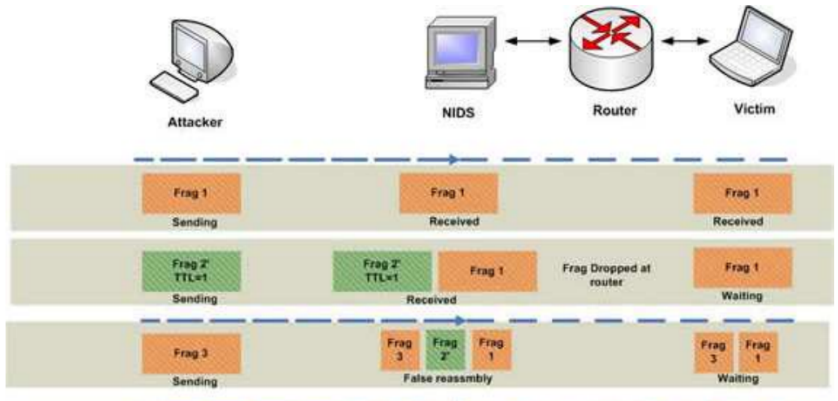

Figure 2.3. IP Fragmentation TTL attack: the attacker sets TTL to 1 for some fragments. In this way, the fragment stays in the IDS and when the next one arrives, it will be delivered to the victim.

2.4.9 GPU-assisted malware

Graphics processing units (GPUs) were initially responsible for rendering 2D and 3D graphics. However, recent malware has exploited these resources. A prototype utilizing this concept was created, employing techniques like self-unpacking, brute-force unpacking, and runtime polymorphism to distribute the malware. Notably, this malware does not function on virtual machines due to its use of specific non-virtualized GPU libraries.
Additionally, GPUs possess the ability to directly access host memory, enabling resource sharing between the CPU and GPU. The author leverages the GPU to perform tasks that would typically raise a flag on the CPU, with the aim of remaining undetected. Although this emerging malware employs familiar packing methods, it capitalizes on the computational prowess of GPUs to evade current anti-malware measures. This technique may not be universally applicable, but it introduces the potential for utilizing alternative computing devices beyond the conventional CPU and RAM setup.

2.4.10 File-less malware

Kaspersky Labs uncovered a novel evasion method in February 2017 involving malware leaving digital traces within compromised server RAM. This technique, combined with tools like Metasploit, offers a fresh way to elude anti-malware systems. Airbus Security's 2016 report further supported this by showing code stored in registry keys with non-ASCII characters, challenging retrieval. These keys also employed obfuscation for complexity.
This tactic of avoiding digital traces in permanent storage effectively evades anti-malware solutions and could be incorporated into targeted attack strategies.

2.4.11 Virtual machine-based malware

Virtual machines (VMs) are used for cloud environments, malware analysis, and penetration testing. They rely on a Virtual Machine Monitor (VMM) to manage resources and abstract hardware for guest machines using emulation.
Virtual Machine Introspection (VMI) techniques enable external services to interact with guest VMs, allowing processes to be injected discreetly.
A framework was devised to satisfy security requirements for this purpose, ensuring that implanted processes remain concealed, resilient, and non-disruptive. This technique holds the potential for malware to fulfill these criteria, allowing malicious processes to be injected without being detected by anti-malware software. Additionally, it could be exploited to compromise virtualized infrastructures, granting malicious users control over multiple VMs.
For VM security, anti-malware solutions must be installed within VMs, as host-level solutions lack access to guest VM memory and files. Harnessing VMI opens avenues for malware development and introduces a novel evasion technique targeting guest VMs, potentially impacting individual VMs or entire infrastructures.

2.4.12 Silent SFX

A Self-Extracting Archive is a form of executable file containing multiple compressed files. This type of file incorporates security measures like password protection to evade antivirus detection. This approach protects malware by managing the process directly, eliminating user input. Malware is deployed using a script and decryptor, it is estimated that the success rate in identifying this type of malware is around 5%. This innovative technique offers a means of camouflaging malware using compressed files.

Chapter 3

Malware detection and classification approaches

The malware detection process is the mechanization that must be implemented to discover and identify the malicious activities of the files under investigation. As a result, several approaches to detecting malware have been improved year after year, with no single approach providing 100% success with all malware types and families in every situation [9]. Therefore, there are three major approaches for malware detection: signature-based, behavioral-based, and heuristic-based. Each one of them has its own advantages and disadvantages, in the following sections there are in-depth analyses of the aforementioned malware detection approaches.
Ömer Aslan Aslan et Refik Samet analyzed a huge amount of the most important papers related to malware detection through machine learning and I will use their consideration for the related works of each detection type [10], in Table 2 are summarized all analyzed works.
Malware classification, on the other hand, involves categorizing and organizing malware into different classes or types based on specific characteristics and behaviors. Instead of determining if a single piece of software is malicious, malware classification aims to understand the broader categories of malware. For example, malware can be classified into categories such as viruses, worms, Trojans, ransomware, adware, and more. The classification helps in studying the different types of malwares and developing targeted strategies for their prevention and removal.
In summary, "malware detection" is the process of identifying individual instances of malicious software, while "malware classification" involves organizing and categorizing malware into different types or classes based on their attributes and behaviors.

3.1 Malware Detection Techniques

3.1.1 Signature-Based

This method relies on a process of signature generation, to uniquely identify each malware a signature is created encapsulating the program structure. The signature generated is a short sequence of bytes uniquely linked to the malware.
The process of generating a signature can be automated but it is preferred to be done manually by malware analysts and reverse engineers especially when is detected a new family of malware. A signature is generated by extracting the content of executables of malwares and when a program must be marked as positive or negative, a signature of the program is extracted and directly compared to all possible malwares signatures, this is often done through a database of malware signatures. There are different methods of signature creation and some of them are: integrity checking, string scanning, top and tail scanning, and entry point scanning.

- Integrity checking: generates a cryptographic checksum with different algorithms like SHA and MD5 for each file in the system in order to identify possible changes that could be caused by malware.

- String scanning: compares the byte sequence in the program file to analyze with the byte of sequences of files in the database.
- Top and tail scanning: it performs a comparison like string scanning but instead of the whole file it analyzes only the top and end points.
- Entry point scanning: the entry point indicates where the first run starts when the file starts to run. This scanning is done because malwares usually changes the entry point of a program in order to execute, before the actual code, malicious code.

Related works for signature-based detection

Ömer Aslan Aslan et Refik Samet analyzed a huge amount of related works for this type of approach, and I would like to put the most emphasis on two of them.
Tang et al. proposed the first system in the category of sequence alignment introducing the contiguous matching encouraging a Needleman-Wunsch (CMENW) algorithm for pair-wise alignment and hierarchical multi-sequence alignment (HMSA) for multiple sequence alignment (MSA), a bioinformatics technique to generate accurate exploit-based signatures for polymorphic worms [11]. Tang et al. availed themselves of the T-coffee approach in order to obtain more accurate Simplified Regular Expression (SRE) signatures. The T-coffee is a consistency-based progressive alignment that allows a combination of MSA methods. The technique involves three steps: multiple sequence alignment to reward consecutive substring extractions, noise elimination to remove noise effects, and signature transformation to make the simplified regular expression signature compatible with current IDSs.
In summary, the sequence alignment approach is costly for long sequences and can be fooled by deliberate noise injection.
The authors claim that the suggested schema is noise-tolerant, and more accurate and precise than those generated by some other exploit-based signature generation schemas. This is because it extracts more polymorphic worm characters like one-byte invariants and distance restrictions between invariant bytes. However, the proposed schema is limited to polymorphic worms and cannot be generalized to other malware types.
Borojerdi and Abadi proposed a MalHunter detection system which is a new method based on sequence clustering and alignment [12]. It generates signatures automatically based on malware behaviors for polymorphic malware. The novel method works as follows: First, from different malware samples, behavior sequences are generated. Then, based on similar behavioral sequences, different groups are generated and stored in the database. To detect malware samples, behavior sequences are gathered and compared with sequences that have been generated earlier and stored in the database. Based on the comparison, the sample is marked as malware or benign. The test results showed that by choosing the cluster radius 0.4 and similarity threshold 0.05, they achieved a detection rate of 90.83% with an FPR of 0.80%.
The authors claim that the proposed schema is resistant to obfuscation techniques, and it can be used for the generic detection of all types of polymorphic malware rather than being limited to a specific malware type. The authors also claim that the suggested system outperformed state-of-the-art signature generation methods including Tang et al. [11], Newsome et al. [13], and Perdisci et al. [14] previously reported in the literature. The proposed method is limited to polymorphic malware and it has been tested on only hundreds of malware which is not enough to determine the performance of the proposed method.

Evaluation of signature-based detection

Signature-based detection is largely used by commercial antivirus thanks to the fast and efficient process, however, it fails to detect new generation malware which uses evasion techniques like polymorphism and obfuscation. There are many techniques and features to extract more powerful and general signatures but each one is limited to the specific cause of use, there is not a generic solution for all detection evasion techniques. Furthermore, static and behavioral signature-based malware detection models suffer from low detection rates when classifying unknown signatures that may be linked to unknown malware or different variants of known malware [9].

3.1.2 Behavioral-Based

Instead of analyzing the executables of malware the behavioral-based approach observes the program behaviors and determines if the program is malware or not. If the program codes could be changed, the behavior of the program would be the same or at least similar. In this way, almost all new malwares can be detected, but, then again, some malwares does not run under a protected or virtualized environment.
There are different ways to extract behaviors and the main procedures are the following:

- Automatic analysis by using sandbox;
- Process monitoring;
- Monitoring of system calls;
- Comparison of registry snapshots;
- Monitoring of file changes;
- Monitoring network activities.

Once obtained the behaviors, with one of the previously cited procedures, there will be a selection of them using data mining. To mark a program as malicious or benign it will be a classification, based on the previously selected feature, through Machine Learning algorithms.

Related works for behavioral-based detection

The behavior-based detection approach is proposed by Fukushima et al. in [15]. The proposed method can detect both unknown and encrypted malware on Windows OS. The proposed framework checks not only specific behaviors that malware performs but also normal behaviors that malware usually does not perform. According to the authors, DR was approximately 60% to 67% without any FP. The DR is very low, to increase the DR, more malicious behaviors could be identified, and to prove the effectiveness of the new method, the test set will be extended.
Semantics-aware malware detection is proposed in [16]. The authors determined that certain malicious behaviors such as a decryption loop in a polymorphic virus appear in all variants of a certain malware. According to the authors, experimental evaluation demonstrated that the algorithm can detect all variants of certain malware with no FPs, and is resilient to obfuscation transformations. However, the algorithm has some limitations for obfuscation transformations. For instance, it cannot handle instruction replacement very well and fails to detect malware that uses this technique. Handling instruction replacement problems and different ordering of memory updates can improve performance.
The authors of [17] were interested in the Windows platform and used the Cuckoo sandbox to extract machine activity data (CPU, memory, received, and sent packets). After that, the observations were transformed into vectors, which were used to train and assess classification algorithms. Common behaviors graph-based malware detection and classification models have been proposed by [18] in their work through observing the most frequent behavior graphs in each malware family. Additionally, [18] presented binary and multi-classification models using (LSTM) long-short term models based on the common API call sequences offered by each malware family.

Evaluation of behavioral-based detection

This approach stems from the need to overcome the limitations of signature-based detection systems, gathering information on how the malware interacts with the system.
The most difficulties of this kind of detection rely on the identification of the common traits among extracted features and handling a huge amount of them after the data mining phase.
Furthermore, some new-generation malwares tries to identify if it is located in a virtual environment or sandbox blocking all its malicious activities. Thankfully, more updated solutions of

antivirus and malware analysis tools provide some mitigations from the above malware countermeasures.
Behavioral-based detection has a huge request of professional studies to improve on the current deficiencies of methods, despite the difficulties in selecting the right features in the large number of features extracted, and the difficulties in identifying the similarities and differences within the features extracted.

3.1.3 Heuristic-Based

The heuristic-based approach depopulated in recent years, relies on generating rules that investigate the extracted data, which are obtained through static or dynamic analysis to guide the inspection of the extracted data to support the proposed malware detection model. Such rules can either be manually generated on the knowledge of expert malware analysts or can be developed automatically using machine learning, tools such as YARA, and other tools.
This malware detection approach has a high accuracy rate in detecting zero-day malware, except the complicated ones.
The heuristic-based detection can use both a signature and some behaviors to generate rules and on top of that generate the signature.

Related works for heuristic-based detection

Authors of [19] introduced a heuristic virus detection based on an automatic generation of multiple neural network classifiers for the detection of unknown Win32 viruses. In order to reduce the false positive (FP) they combine the individual classifier outputs using a voting procedure.
On the basis of the previous paper, [20] presented some improvements.
The number of rules automatically generated was reduced by using rule pruning, ranking, and selection. The reduction of rules impacted higher accuracy and faster detection time. According to the paper, the proposed paper system outperformed popular antivirus software tools, data mining-based detection systems such as support vector machine (SVM), decision tree techniques, and naive Bayes.
Static [21] and dynamic [22] approach are widely used for heuristic-based detection.
Bilar et al. used a statistical analysis of opcode frequency distributions to detect and differentiate new generation malware like polymorphic ones. The paper highlights a statistically remarkable difference in opcode distribution between malware and benign. Unfortunately, there is a need to extend the number of samples to get more reliable results.
On the other hand, Naval et al.[22] suggested a dynamic malware detection system, which collects system calls and constructs a graph that finds the semantically relevant paths among them. By measuring the most relevant paths the authors obtained a reduction of time complexity. Furthermore, the proposed approach, according to what is stated in the paper, should outperform the competitors due to the high resilience to system call injections. Sadly, the proposed approach has some problems like the inefficient selection of relevant paths, vulnerability to call injection, and low performance during path computation. An improvement in the performance could be expected from previous problem resolution.

Evaluation of heuristic-based detection

Heuristic-based malware detection combines the use of strings and behaviors in order to generate the rules, which are in turn used to generate the signature.
Despite the signature-based once, this approach can detect various forms of unknown malwares but this is not extended to all new-generation malware which is still suffering. Furthermore, this method is prone to a high false positive rate.

3.2 Malware Classification Approaches

AlAhmadi and Martinovic introduce MalClassifier [23], a privacy-preserving system for automatic malware analysis and classification based on network flow sequence mining. It identifies malware families without needing access to the infected host, reducing response time. The system abstracts malware behavior into n-flows, generating profiles used to build supervised machine learning classifiers (K-Nearest Neighbour and Random Forest). It achieves 96% F-measure for family classification using ransomware and botnet datasets. MalClassifier remains resilient to malware evasion through flow manipulation and proves effective in identifying reoccurring malware flow patterns. The increasing number of malware variants poses a challenge for anti-malware vendors, requiring accurate family classification. Existing tools often depend on sandbox environments, but MalClassifier performs on-the-wire classification without direct host access. The research compares related work and improves the system's privacy awareness by using non-payload features and addressing malware behavior changes. MalClassifier applies fuzzy and order sequence similarity measures for flow matching, enabling IP-agnostic analysis and making it resistant to encryption. It mines distinctive n-flows for each family, achieving over 95% F-measure for classification.

Piskozub, Spolaor, and Martinovic addressed the challenge of detecting malware in network traffic using flow-level data. They propose MalAlert [24], a machine learning-based system that distinguishes between different types of malware (adware, ransomware, viruses, etc.) using flowsets and statistical fingerprints. The system aggregates flows based on communicating IP addresses, extracts 441 statistical features, and selects representative features using relative mutual information. The fingerprinting approach is IP address- and port-agnostic, preserving user privacy and resisting port spoofing. It can aggregate thousands of flows into a single flow set while maintaining information about their maliciousness. MalAlert is tested on datasets containing over 65,000 malware samples and 23 billion flows from the University of Oxford, identifying 0.11% suspicious flowsets classified as malicious. The system's effectiveness lies in its ability to extract informative fingerprints while maintaining privacy and robustness against malware evasion techniques.

Piskozub, De Gaspari, Barr-Smith, Mancini, and Martinovic introduce MalPhase [25], a system designed to address the limitations of aggregated flow-based network traffic analysis for malware detection. MalPhase utilizes a multi-tier architecture and an extended set of network flow features to improve deep learning models' performance in detecting malicious flows (<98% F1]) and classifying them into respective malware types(<93% F1) and families (<91% F1). The increase in new, unique malware samples and their widespread success are discussed, along with various techniques proposed to counter daily threats, including network-based approaches. The paper distinguishes packet-level and flow-level network analysis methods, highlighting the advantages and challenges of each. MalPhase's core contributions include its ability to detect and classify various malware families, its multi-tier design for offline and online analysis, and its evaluation of a large malware dataset. The system demonstrates robustness to real-life conditions with noise and detects unseen malware samples with comparable performance to known samples. Overall, MalPhase provides an effective and scalable solution for detecting and classifying malware based on aggregated flow data.

Joonseo Ha and Heejun Roh discussed the challenge of detecting malware in TLS-encrypted traffic and proposed a systematic framework [26] to evaluate malware family classification methods in a controlled environment. Researchers have focused on statistical, machine learning, and neural network-based methods for TLS-encrypted malware detection but evaluating sequential information usage in malware family classification is lacking. The proposed framework extracts common flow-level features from TLS-encrypted traffic and evaluates state-of-the-art methods. Experimental results show that graph-based representations achieve better performance. The article addresses flaws in existing labeled datasets and provides insights into future work. The growth of TLS adoption has facilitated encrypted malware traffic, necessitating better detection methods. The study presents a comprehensive background on encrypted traffic classification and the challenges of malware family classification for TLS-encrypted traffic. The proposed framework contributes to advancing TLS-encrypted malware detection and classification research. In the

evaluation, the TIG-based classifier achieves the best accuracy of 97.83%, but noise in the training set can lead to mispredictions. The study suggests potential research directions to improve classifiers, such as enhancing the graph neural network architecture with noisy labels.

3.3 Consideration of Malware Detection and Classification

In general, research and papers on malware classification based on network traffic are less developed due to the scarce presence of datasets but furthermore, the classification is even less developed. There are generally fewer papers on malware classification compared to malware detection in the context of network traffic analysis for several reasons:

- Complexity: Malware classification is a more challenging task than malware detection. Malware detection aims to identify whether network traffic contains malicious activity, while malware classification seeks to categorize the specific malware family or type. Classification requires a more detailed analysis of the network traffic and demands more sophisticated algorithms and features.
- Availability of Labeled Data: Building a reliable labeled dataset for malware classification is more difficult than for malware detection. Collecting and labeling network traffic samples with specific malware families or types is time-consuming and resource-intensive. As a result, researchers may face limited access to large and diverse labeled datasets for classification tasks.
- Focus on Detection: The primary concern in cybersecurity research often revolves around early detection and prevention of malware attacks. Researchers tend to prioritize developing efficient detection methods that can identify malicious traffic quickly. Consequently, there might be more emphasis on detection papers due to their immediate applicability in real-world scenarios.
- Complexity of Feature Engineering: Effective malware classification often requires more sophisticated feature engineering techniques, such as considering the sequential information in the network traffic, as seen in the paper mentioned. This adds complexity to the research process, and some researchers might opt for simpler detection approaches instead.
- Computational Resources: Malware classification tasks, especially those using deep learning methods, may require more computational resources than detection tasks. This could be a limiting factor for some researchers, leading them to focus on detection, which can be computationally more efficient.
- Specialized Skillset: Developing robust and accurate malware classification models may require a more specialized skillset in machine learning, deep learning, and data analysis. Researchers might be more inclined to pursue detection tasks that may not require the same level of expertise.

Overall, while both malware detection and classification are essential in cybersecurity, the focus on detection may be more prominent due to its immediate practicality and relatively easier implementation. However, as the field of network security advances and more labeled datasets become available, we can expect to see a growing interest in malware classification research. Despite the previous considerations in this work, I will analyze the strengths and weaknesses of malware classification by network traffic.

3.4 Machine Learning Classification Algorithms

3.4.1 Decision Tree

Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where internal nodes represent the features of a dataset, branches represent

the decision rules, and each leaf node represents the outcome [27].
Its name comes from the graphic representation that resembles a tree, starting from a root node that grows on additional branches and creates a structure resembling a tree.
In the figure (Figure 3.1) there is an example of how a tree branches.

- Advantages:
  - Interpretability: Since the process is similar to human decision it is easy to understand, it is possible to say that is a white box model because of its transparency;
  - Features Interpretability: This algorithm helps in understanding the importance of different features;
  - Categorical Data: Decision trees do not need encoded data because they can handle both categorical and numerical features.
- Disadvantages:
  - Instability: A solution can be completely different even if there are only a few changes in data;
  - Dominant Class: This algorithm, in case of imbalanced data, is prone to biased predictions towards the dominant class.
  - Computational complexity: In case of a dataset with a large amount of data and features, the trees can grow deep, becoming difficult to interpret and large.
  - Overfitting: Decision trees may have overfitting issues that can be resolved by using the Random Forest algorithm.

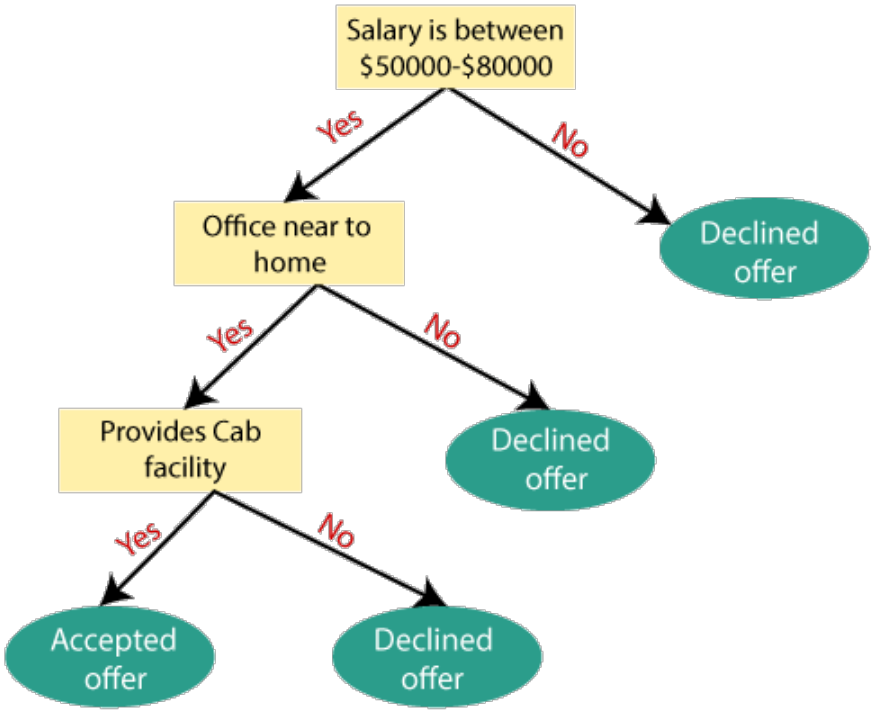

Figure 3.1. Random forest algorithm with samples taken from the fruit basket.

3.4.2 Random Forest

Random Forest is an ensemble learning method that builds multiple decision trees and combines their predictions to make a final decision. Each tree is trained on a random subset of the training data and features. For example the figure (Figure 3.2) explains how random forest works [28].

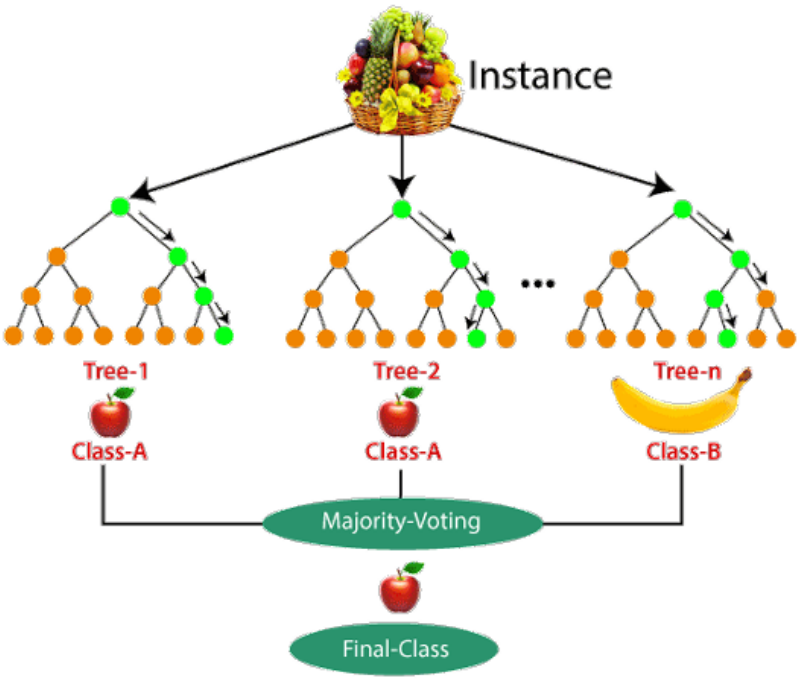

Figure 3.2. Random forest algorithm with samples taken from the fruit basket.

- Advantages:
  - Robustness: Random Forest is less prone to overfitting due to the averaging of multiple trees;
  - High accuracy: It typically yields accurate results, making it suitable for various tasks;
  - Feature importance: Random Forest can provide insights into feature importance, aiding in feature selection;
  - Parallelization: The trees in the forest can be trained in parallel, making it computationally efficient.
- Disadvantages:
  - Complexity: Random Forest models can become complex and challenging to interpret, especially with a large number of trees;
  - Memory consumption: The storage requirements can be high for large forests with many trees;
  - Training time: Training multiple trees can be time-consuming, especially on large datasets.

3.4.3 Support Vector Machine(SVM)

Support Vector Machine is a supervisored learning and its objective of the support vector machine algorithm is to find a hyperplane in N-dimensional space(N - the number of features) that distinctly

classifies the data points [29].
As an example, figure (Figure 3.3) illustrates how a hyperplane is selected in 2D and 3D, while in figure (Figure 3.4) how it is selected among the candidates in 2D.

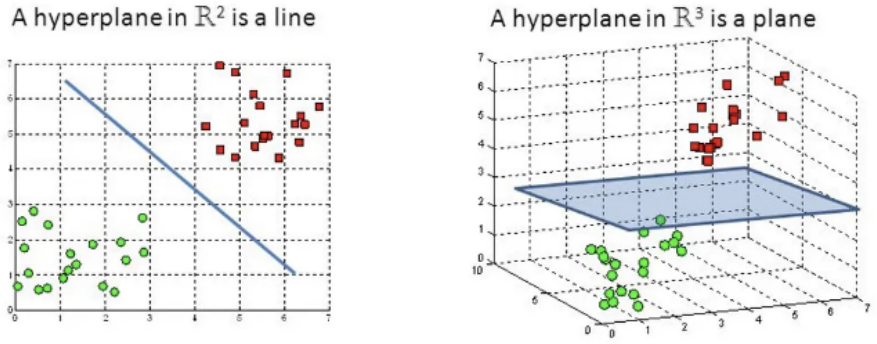

Figure 3.3. Hyperplanes in 2d and 3D feature space

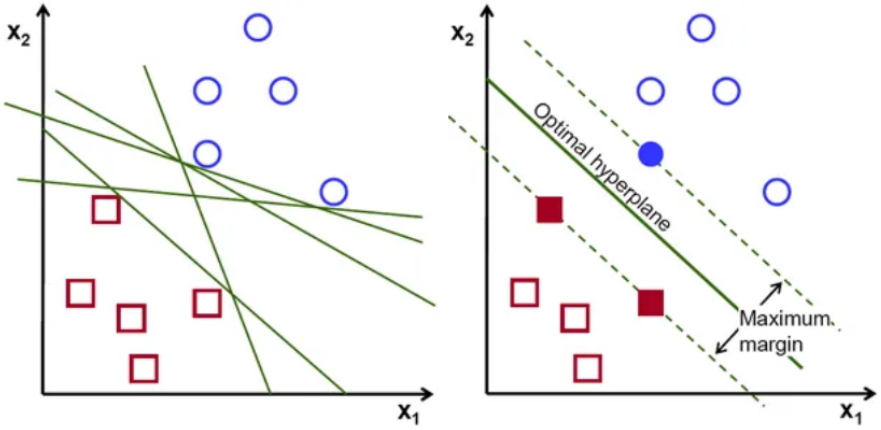

Figure 3.4. Possible hyperplanes

- Advantages:
  - Features dependencies: The SVM performs well in case the classes have a clear margin of separation and in case the number of features is greater than the number of samples;
  - Effective in High-Dimensional Spaces: In case of a high number of features the SVM works well;
  - Robust to Overfitting: Because of its algorithm, the plane search that best separates the two classes, makes it less subject to overfitting;
- Disadvantages:
  - Computationally Intensive: In case of large dataset or complex kernel functions, the training of an SVM can be computationally expensive;
  - Sensitive to noise: In case of datasets with noise like target classes that overlap, the SVM will underperform;
  - Interpretability: As illustrated in figure (Figure 3.3) the interpretability of the boundaries and representation of the points becomes more complex as the number of features grows.

3.4.4 K-Nearest Neighbors (KNN)

KNN is a simple and intuitive algorithm that classifies data points based on the majority class of their K nearest neighbors in the feature space. For example the figure (Figure 3.5) describes the KNN algorithm [30].

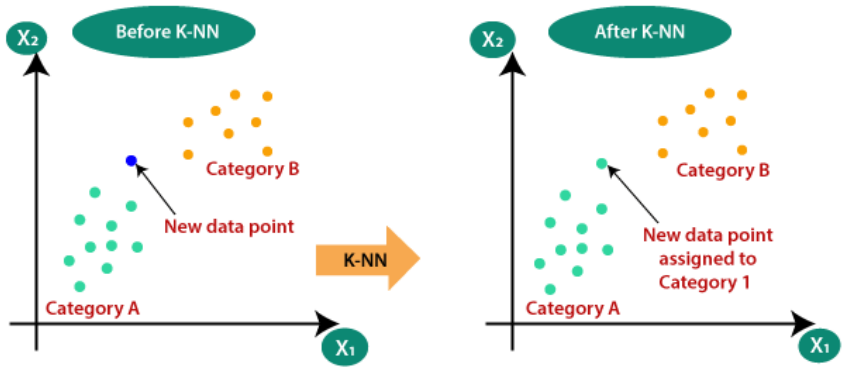

Figure 3.5. K-Nearest Neighbours algorithm example

- Advantages:
  - Simple implementation: KNN is easy to understand and implement, making it a good starting point for classification tasks;
  - No training phase: KNN does not require a training phase, as it memorizes the training data;
  - Non-parametric: It does not assume any underlying data distribution, making it versatile.
- Disadvantages:
  - Computational cost: The classification time grows with the size of the training data, as it needs to calculate distances to all data points;
  - Sensitivity to feature scaling: KNN is sensitive to the scale of features and may require normalization;
  - Optimal K value: Selecting the appropriate K value is crucial for KNN's performance and can be challenging.

3.4.5 Artificial Neural Network

Artificial Neural Networks are one of the greatest and most appreciated algorithms of machine learning, in particular, they are part of deep learning ones.
Artificial neural networks (ANNs) are comprised of node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network [31].
In figure (Figure 3.6) you can see a graphic representation of a neural network of one layer. Starting from the previously cited image, if the number of hidden layers is more than three we can identify it in deep learning, and this kind of neural network with one input layer, one output layer, and a series of hidden layers is also known as feed-forward neural network.

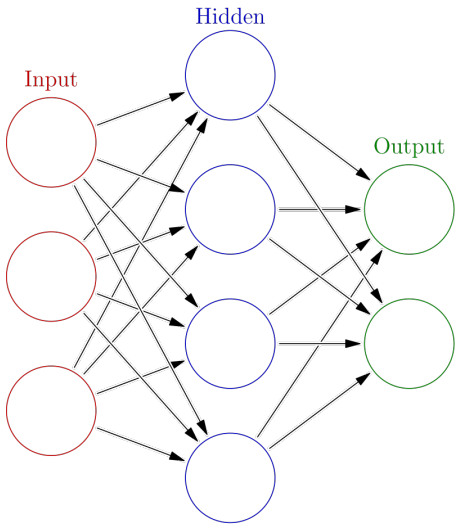

Figure 3.6. Neural network of 1 layer.

Instead, the most famous artificial neural networks are Convolutional Neural Networks (CNNs) and Recurrent Neural Networks(RNNs).

- CNN: similar to feed-forward network but the layers used are normally from linear algebra, matrix manipulations, and identification of patterns in images. For this reason, this kind of network is widely used in image recognition, pattern recognition, and/or computer vision.
- RNN: the peculiarity of these networks relies on their feedback loops implementing the concept of memory, this feature is well suited to the recurring data and time-series data to make predictions of future outcomes, like market prediction but also for speech-to-text, automatic translations, and audio tasks. (Figure 3.7)

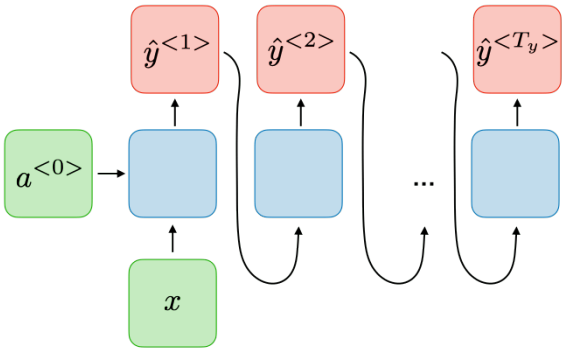

Figure 3.7. Recurrent Neural Network One-to-Many

- Advantages:
  - Versatility: the vastness of the artificial neural network allows you to range over any field of application;
  - Transfer Learning: previously trained neural network models can be fine-tuned for other tasks, thus reducing costs in terms of time and resources;
  - Feature Learning: the artificial neural network can automatically learn relevant features from raw data, saving time for data mining;

- Disadvantages:
  - Complexity: Artificial Neural Network architectures can be complex, making them challenging to design and tune, especially for a beginner;
  - Computationally intensive: The training phase of an Artificial Neural Network is a computationally expensive process that needs specific resources, especially in case of deep learning and large datasets;
  - Data Dependency: In order to achieve good performance and avoid overfitting from an ANN a large amount of training data is required;
  - Hyperparameter sensitivity: This model is very sensitive to the hyperparameter choice, small changes in the learning rate, number of neurons per layer, and number of layers can lead to a totally different performance;
  - Complexity: Is very difficult to understand the model predictions, this is caused by their black-box nature.

Chapter 4

Datasets

This chapter describes the evolution of the dataset used during the different stages of the thesis. Before moving on to the description of the chapter you need a premise; unfortunately, specific datasets that deal with malware via network traffic are not easily accessible publicly and most papers and theses use custom datasets combining some well-known with self-produced datasets of malicious and non-malicious traffic. In recent research efforts to classify malware families in TLS-encrypted traffic, only a handful of research groups have been actively involved. Among these, the work by Anderson and McGrew at Cisco Systems [32] is widely regarded as providing some of the most comprehensive and reliable results in the field. Cisco's successful commercialization of their research as "encrypted traffic analytics" (ETA) [33]but the datasets they used are not publicly available. Due to the lack of malware family datasets with network traffic, there have been collaborative efforts within the research community to collect and distribute datasets of both encrypted and unencrypted malware traffic. These datasets, as discussed by Thakkar and Lohiya [34], offer pre-processed features tailored for intrusion detection. However, they typically lack crucial sequential information regarding TLS flow, making it challenging to assess recent research endeavors in this area. As a starting point, I have selected the paper [23] referenced by most of the recent papers and one of the most efficient algorithms for malware family classification which is cited by almost all the newest papers. I have selected this particular paper also because of its well-described workflow and procedures, the public availability of the dataset, and the possibility of applying improvements. Following the previous example of the StratosphereIPS dataset, a recent malware traffic classification challenge, NetML 2020, incorporated a feature dataset selectively derived from this raw dataset and was conducted as part of the NETAML workshop, coinciding with IJCAI-PRICAI 2020. The NetML feature dataset contains most of the enhanced features used by Anderson and McGrew. Nonetheless, it is crucial to exercise caution when utilizing both types of datasets. But the NetML 2020 [35] dataset has some problems related to the correctness of the connection established by the malwares, for example, the TrickBot samples with a simple Markov chain fingerprint due to a restricted communication pattern: ClientHello from Trickbot and HandshakeFailure from the server due to an unsupported SSL/TLS version. For this reason, also this dataset has been discarded, and I decided to build my own dataset using the public repository of the blog malware-traffic-analysis [36], with malware family pcap collected by a security expert from a sandbox.

4.1 CTU-13 Datset

Following the example of [23] I have tried to use CTU-13 Dataset [37], The CTU-13 is a dataset of botnet traffic that was captured in the CTU University, Czech Republic, in 2011. The goal of the dataset was to have a large capture of real botnet traffic mixed with normal traffic and background traffic. The CTU-13 dataset consists of thirteen captures (called scenarios) of different botnet samples. In each scenario, we executed a specific malware, which used several protocols and performed different actions. The relationship between the duration of the scenario, the number of packets, the number of NetFlows, and the size of the pcap file is shown in table 4.1. This

Table also shows the malware used to create the capture, and the number of infected computers on each scenario.

ID	Duration(hrs)	# Packets	#Netflows	Size	Bot	#Bots
1	6,15	71.971.482	2.824.637	52 GB	Neris	1
2	4,21	71.851.300	1.808.123	60 GB	Neris	1
3	66,85	167.730.395	4.710.639	121 GB	Rbot	1
4	4,21	62.089.135	1.121.077	53 GB	Rbot	1
5	11,63	4.481.167	129.833	37,6 GB	Virut	1
6	2,18	38.764.357	558.920	30 GB	Menti	1
7	0,38	7.467.139	114.078	5,8 GB	Sogou	1
8	19,5	155.207.799	2.954.231	123 GB	Murlo	1
9	5,18	115.415.321	2.753.885	94 GB	Neris	1
10	4,75	90.389.782	1.309.792	73 GB	Rbot	10
11	0,26	6.337.202	107.252	5,2 GB	Rbot	10
12	1,21	13.212.268	325.472	8,3 GB	NSIS.ay	3
13	16,36	50.888.256	1.925.150	34 GB	Virut	1

Table 4.1. Item specifications of CTU-13 Datset

4.2 Custom Dataset

After the data mining stage, I had to go back and select a new one because of the high time consumption; I have also tried to use a hash-map for already calculated distances but the results still did not make the task feasible. To overcome this problem I decided to create a new dataset based on different sources for the same family of malware. In particular, I merged the datasets of stratosphereips.org and the malware-traffic-analysis blog. Stratosphere IPS [37] provides, in addition to CTU-13, a set of single captures of different malware families and normal traffic. On the other hand, Malware-traffic-analysis [36] is a blog that focuses on network traffic related to malware infections. Is a source for packet capture (pcap) files and malware samples. Since the summer of 2013, this site has published over 2,200 blog entries about malicious network traffic. From the merge of the two previously discussed datasets, there were obtained three malware families: trickbot, ramnit, and dridex. In the following table 4.2 is described the dataset created.

Trickbot	2017-06-12-Trickbot-malspam-traffic
	2017-06-14-Trickbot-malspam-traffic
	2017-07-24-Trickbot-malspam-traffic
	2017-08-11-Trickbot-infection-from-carriereiter.com
	2017-08-12-Trickbot-infection-from-carriereiserphotography.com
	2017-08-12-Trickbot-infection-from-carriereiter.com.exe
	2017-08-12-Trickbot-infection-from-usdata.estoreseller.com
	2017-08-21-Trickbot-malspam-traffic
	2018-04-30-Trickbot-goes-from-client-to-domain-controller
	2018-05-25-Trickbot-malspam-infection-traffic
	2018-06-29-Trickbot-infects-client-then-moves-to-DC
	2018-07-05-Trickbot-infection-traffic
	2018-07-21-Trickbot-malspam-infection-traffic
	2018-08-17-Emotet-plus-Trickbot-infection-traffic
	2018-09-03-Trickbot-malspam-infection-traffic
	2018-10-15-Trickbot-gtag-jim332-infection-traffic
	2018-11-12-Trickbot-infection-traffic-gtag-sat100
	2018-12-07-Trickbot-infection-traffic-ser1207
	2019-01-11-Trickbot-malspam-infection-traffic
	2019-04-27-Trickbot-infection-traffic
	2019-07-02-Trickbot-infection-with-CookiesDll-module
	2019-09-25-Trickbot-gtag-ono19-infection-traffic
	2020-02-19-Trickbot-gtag-wecan23-infection
	2020-02-25-Trickbot-gtag-red4-infection-traffic
	2020-02-26-Trickbot-spreads-from-infected-client-to-DC
	2020-03-04-Trickbot-spreads-from-client-to-DC
	2020-04-13-Trickbot-gtag-man6-infection-traffic
	2020-04-13-Trickbot-gtag-yas27-infection-traffic
	2020-04-20-Trickbot-gtag-ono38-infection-traffic
	2020-06-16-Trickbot-gtag-ono47-infection-traffic
	2020-06-25-Trickbot-gtag-gi6-infection-traffic
	2020-07-10-Trickbot-gtag-chil65-infection-traffic
	2020-09-08-Trickbot-gtag-ono72-infection-traffic
	2020-11-09-Trickbot-gtag-rob2-infection-traffic
	2020-11-09-Trickbot-gtag-tar2-infection-traffic
	2021-02-17-Trickbot-gtag-rob13-infection-in-AD-environment
	2021-05-26-Trickbot-infection-with-Cobalt-Strike
	2021-07-12-Trickbot-gtag-rob106-infection-traffic
	Traffic-for-2017-08-15-ISC-diary-on-Trickbot-malspam
	Trickbot-strato-2017-03-07
	Trickbot-strato-2017-06-13
Dridex	dridex-starto
	dridex-blog-2019-11
	dridex-blog-2019-07
	dridex-blog-2020-05
	dridex-blog-2020-07
	dridex-blog-2020-12
	dridex-blog-2021
Ramnit	ramnit-blog-2016-05-16-first
	ramnit-blog-2016-05-16-second
	ramnit-blog-2017-06-02-first
	ramnit-blog-2017-06-02-second
	ramnit-blog-2017-08-25
	ramnit-blog-2017-12-28
	ramnit-blog-2018-02-12-first
	ramnit-blog-2018-02-12-second
	ramnit-strato-2018-04-03

Table 4.2: Items of the dataset crafted

Starting from a difficulty in processing the large amount of data from the first dataset, the opportunity arose to be able to verify how the classifiers produced by the previous papers react to the evolution of malware over the years. The objective of this thesis is the analysis of the MalClassifier algorithm with the evolution of malware families and the proposal of a model that improves performance.

Chapter 5

Proposed Model

The intent of this thesis is to identify and classify malware families despite their evolution over time using machine learning algorithms. In order to do so, it is necessary to compare different solutions using different algorithms and assigning different weights to the available features. It is important to specify that there is not an optimal algorithm that always performs better than others, machine learning algorithms depend on the inputs this is the cause why we have used different paths to achieve the same goal.

Given the complexity of the task, it was necessary to follow up a specific workflow. In this chapter, although it may fit into this section, I will not focus on the research of the dataset and malware families evolution because they were widely discussed in the previous chapter. The first step was one of the longest steps and it aimed to pre-process the data with the Zeek framework in order to assemble the various packets in flows, followed by the data mining which aims to extrapolate the most important packets for family classification through the correlation of each packet with the whole family dataset.

After achieving the results of the data mining I divided the dataset in train and test following the results of the previous step and tested it with the selected machine learning algorithms. The results produced with the test were followed by a deep analysis and based on the analysis the process was iterated re-splitting the dataset in train and test.
In order to clarify the results of the previous step and inspect better the behavior of the classification a series of tests were conducted with a dataset based on one pair of the malware families at a time.
In the last steps I have performed some tests with the Repeated Stratified K Fold cross-validator, these tests aim to compare the efficiency of cross-validators as a replacement for the data mining phase.
In the end was done a detailed analysis of all the results understanding the problems during the work and the evidence. The figure (Figure 5.1) summarizes the workflow.

5.1 Work environment and tools used

The following section lists the working environments, frameworks, and libraries used during the thesis process. Only the data mining has been done from scratch, following the directions listed by MalClassifier. On the contrary, machine learning algorithms have been reused from versions that have already been developed, and tested and for which their goodness has been recognized.

5.1.1 Legion

Politecnico di Torino HPC project is an Academic Computing center that provides computational resources and technical support for research activities for academic and didactical purposes. The

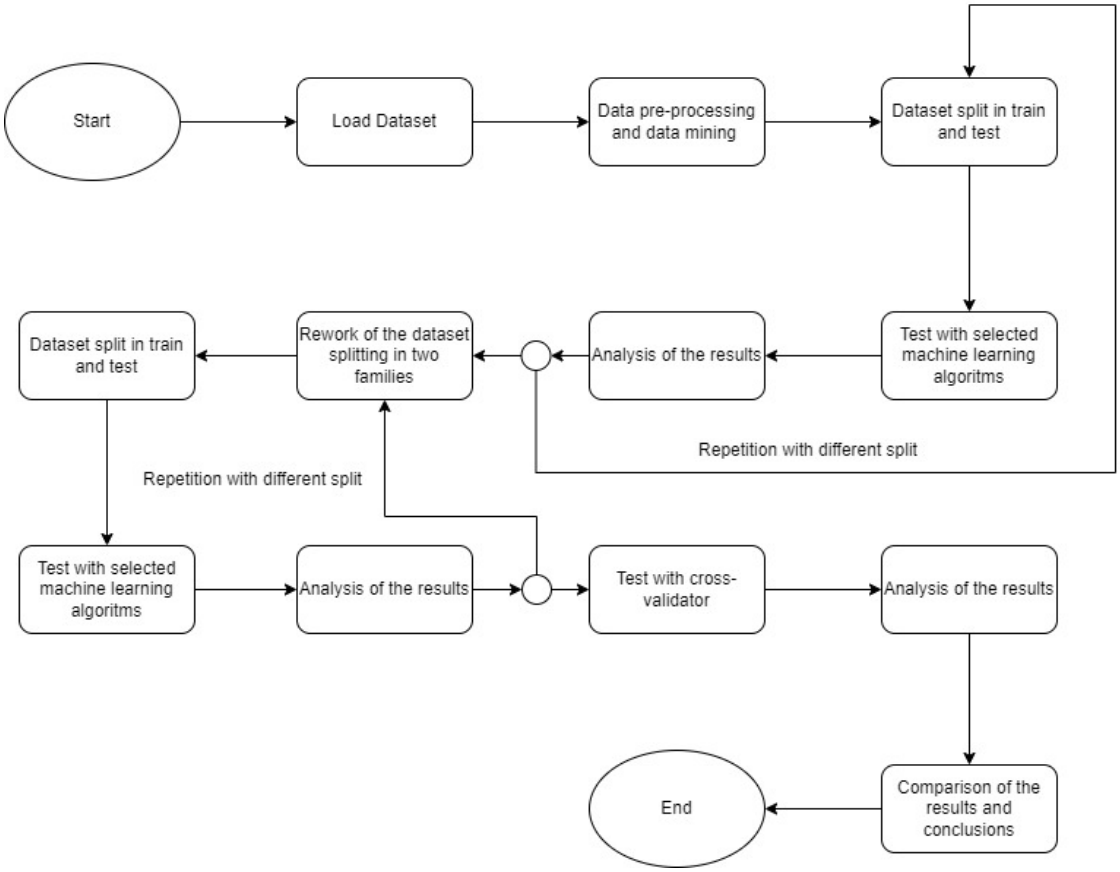

Figure 5.1. Overviw of the workflow.

HPC project is officially managed by LABINF (Laboratorio Didattico di Informatica Avanzata) under the supervision of DAUIN (Department of Control and Computer Engineering) which granted by Board of Directors [38]. The HPC provides three different clusters, I have used them for this purpose the Legion and its specifications are detailed in the table 5.1.

Architecture	Cluster Linux Infiniband-EDR MIMD Distributed Shared-Memory
Node Inteconnect	Infiniband EDR 100 Gb/s
Service Network	Gigabit Ethernet 1 Gb/s
CPU Model	2x Intel Xeon Scalable Processors Gold 6130 2.10 GHz 16 cores
GPU Node	24x nVidia Tesla V100 SXM2 - 32 GB - 5120 cuda cores
Performance	90 TFLOPS (July 2020)
Computing Cores	1824
Number of Nodes	57
Total RAM Memory	22 TB DDR4 REGISTERED ECC
OS	CentOS 7.6 - OpenHPC 1.3.8.1
Scheduler	SLURM 18.08

Table 5.1. Legion specifiications

5.1.2 Google colaboratory

Colaboratory, or “Colab” for short, is a product from Google Research. Colab [39] allows anybody to write and execute arbitrary Python code through the browser, and is especially well suited to

machine learning, data analysis, and education. More technically, Colab is a hosted Jupyter notebook service that requires no setup to use, while providing access free of charge to computing resources including GPUs.

5.1.3 Zeek

Zeek is a passive, open-source network traffic analyzer. Many operators use Zeek as a network security monitor (NSM) to support investigations of suspicious or malicious activity. Zeek also supports a wide range of traffic analysis tasks beyond the security domain, including performance measurement and troubleshooting [40]. This tool was used during the pre-processing phase and it reduced a lot the working time estimated to statically and behaviourally analyze the network interactions, application-level protocols, and exchanged content for each network stream.
This solution is widely used by enterprises also because of the difficulties in preserving complete network traces (PCAPs) for extended durations, Zeek provides several logs for each PCAP analyzed and it helps in cost-effective storage management.

5.1.4 Scikit-learn

Scikit-learn is a Python module integrating a wide range of state-of-the-art machine-learning algorithms for medium-scale supervised and unsupervised problems. This package focuses on bringing machine learning to non-specialists using a general-purpose high-level language [41].
Thanks to previous experience, obtained during other courses at the Politecnico di Torino, I was able to familiarize myself with the previously mentioned library, which immediately proved to be simpler than its counterparts for a person less accustomed to machine learning.
The goal of this thesis has never been the creation of a machine learning algorithm as much as the use of already existing and consolidated libraries for solving the task, and this has been reflected in the use of the scikit-learn library.

The library was crafted to seamlessly integrate with the collection of numeric and scientific packages that revolve around the NumPy and SciPy libraries. NumPy enhances Python by introducing a continuous numeric array data type and efficient array computing functions. In addition, SciPy expands Python's capabilities by offering common numerical operations. These can either be implemented within Python/NumPy or incorporated through existing C/C++/Fortran implementations. [41]

5.2 Pre-processing and Sub-Sequence Extraction

The first layer of the model is the Pre-processing and Sub-Sequence Extraction, starting from the captures of network traffic packets we reassemble the flow discarding useless packets or corrupted ones by the Zeek framework. In input we give Pcaps files and the output obtained from them are numerous logs, in particular, we use a conn.log file in which there are all the connection flows reassembled with mapped parameters as shown in the following list.

- Ts - timestamp;
- Uid - unique identifier of the connection;
- Id - connection's 4-tuple of endpoint addresses/ports;
- Proto - transport layer protocol;
- Service - an identification of an application protocol;
- Duration;
- Orig_bytes - number of payload bytes sent by origination;

- Resp_bytes - number of payload bytes sent by responder;
- Conn_state - (Description in table 5.2);
- Local_orig - If the connection is originated locally, this value will be T;
- Local_resp - If the connection is responded to locally, this value will be T;
- Missed_bytes - Indicates the number of bytes missed in content gaps, which is representative of packet loss;
- Orig_pkts - Number of packets that the originator sent;
- Orig_ip_bytes - Number of IP level bytes that the originator sent;
- Resp_pkts - Number of packets that the responder sent;
- Resp_ip_bytes - Number of IP level bytes that the responder sent;
- Tunnel_parent - If this connection was over a tunnel, indicate the uid values for any encapsulating parent connections used over the lifetime of this inner connection;
- History - (Description in table 5.3);

Connection state parameters	
S0	Connection attempt seen, no reply
S1	Connection established, not terminated
SF	Normal establishment and termination
REJ	Connection attempt rejected
S2	Connection established and close attempt by originator seen (but no reply from responder)
S3	Connection established and close attempt by responder seen (but no reply from originator)
RSTO	Connection established, originator aborted (sent a RST)
RSTR	Responder sent a RST
RSTOS0	Originator sent a SYN followed by a RST, we never saw a SYN-ACK from the responder
RSTRH	Responder sent a SYN-ACK followed by a RST, we never saw a SYN from the (purported) originator
SH	Originator sent a SYN followed by a FIN, we never saw a SYN-ACK from the responder (hence the connection was “half” open)
SHR	Responder sent a SYN-ACK followed by a FIN, we never saw a SYN from the originator
OTH	No SYN seen, just midstream traffic (one example of this is a “partial connection” that was not later closed)

Table 5.2. Zeek connection state description

From the previous list, only the following parameters were analyzed:

- Ts
- Uid
- Id
- Proto
- Service

History parameters	
s	a SYN w/o the ACK bit set
h	a SYN+ACK (“handshake”)
a	a pure ACK
d	packet with payload (“data”)
f	packet with FIN bit set
r	packet with RST bit set
c	packet with a bad checksum (applies to UDP too)
g	a content gap
t	packet with retransmitted payload
w	packet with a zero window advertisement
i	inconsistent packet (e.g. FIN+RST bits set)
q	multi-flag packet (SYN+FIN or SYN+RST bits set)
^	connection direction was flipped by Zeek’s heuristic

Table 5.3. Zeek history parameter description

- Duration
- Orig_bytes
- Resp_bytes
- Conn_state
- Orig_pkts
- Orig_ip_bytes
- Resp_pkts
- Resp_ip_bytes
- History

As the output of the Zeek application, we obtain .txt files with the previous information that must be converted into .csv format for the next steps. After having obtained the csv files we have all the inputs needed for the data mining section.

5.3 Data mining

As previously introduced, the initial dataset based on CTU-13 was discarded at the first round of data mining, because the time needed to analyze and populate the matrices was estimated to be more than 6 months. To be more specific with 21 hours of work only 253 rows of 31734 were produced for an example of a matrix of dimension 31734 x 137839.

During this stage is done the correlation between the different files, filling the correlation matrices with a similarity distance based on the following formula: *Flow similarity* = 3 *Binary similarity*+2 *Levenshtein Distance*+6 *Cosine Similarity*+1 *Inter-flow Distance*.

The resulting similarity distance is in the range 0 to 12 and is the result of a weighted average of four similarity distances that are described in the following lines.

- **Binary similarity** (resp_port, protocol, service): The similarity is 1 if the attribute values are the same, otherwise 0.

- **Levenshtein Distance** (history, conn_state): Levenshtein Distance is a fuzzy string similarity measure that measures the minimum number of modifications required (insertions, deletions, and substitutions) to change one string into the other, divided by the maximum length of the same two strings. It also takes into consideration the order of the characters in the string
- **Cosine Similarity**: The numeric attributes of the two flows are represented as two vectors. Cosine Similarity measures the similarity of two non-zero vectors by calculating the cosine of the angle between them. Given the vectors x and y of length n = 6 (number of numeric attributes), the cosine similarity is represented as:

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{xy}}{\|\mathbf{x}\|\|\mathbf{y}\|} = \frac{\sum_{i=1}^{n} \mathbf{x}_i \mathbf{y}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{x}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{y}_i)^2}}$$

- **Inter-flow Distance** (resp_port): distance calculates the distance between the resp port in every two consecutive flows of a sub-sequence. This helps identify malware network behavioral attributes such as performing a port scan (e.g. when the difference of resp port of two consecutive flows is 1). To calculate the inter-flow similarity, we first calculate the distance of the resp port in each consecutive flow in a sub-sequence.

In [23], in IV. Malclassifier Design section B, Malware Family Profile Extraction, there are some examples of the different distances that can be resumed as follows:

Thus, the Binary Similarity in our example of ($f_{0_A} \in Seq_1 : 80|tcp|http, f_{0_B} \in Seq_2 : 80|tcp|http$) and ($f_{1_A} \in Seq_1 : 80|tcp|http, f_{1_B} \in Seq_2 : 25|tcp|ssl$) is (1,0.33) respectively, resulting in an average Binary Similarity of 0.665.

Assuming the cost of insertion, deletion, and modification is the same (= 1), then the Levenshtein Distance of making $ShADdFa$ into ShADadR is 3.

the Levenshtein Distance of ($f_{0_A} \in S_{eq_1} : SF\,|ShADdFa, f_{0_B} \in Seq_2 : SF|$ ShADadfF) and ($f_{1_A} \in Seq_1 : SF \mid ShADdFa, f_{1_B} \in Seq_2 : S0 \mid ShAdDafF$) is (2,4) respectively, resulting in an average Levenshtein Distance of 3, scaled to 0.03.

The Cosine Similarity of x = [367,3547,6,615, 7,3835] $\in f_{0A}, y$ = [1801,15606,14,2369,18,16334] $\in f_{0_B}$ is 0.00025 . Similarly, the Cosine Similarity of x = [322,464,5,530,4,632] $\in f_{1_A}, y$ = [5274,1370,18, 5975,28,456] $\in f_{1B}$ is 0.284 . Thus, the average Cosine Similarity for Seq_1 and Seq_2 is 0.142 .

For the Inter-Flow distance the resp_port distance between $f_{0_A}, f_{1_A} \in Seq_1$ is 0 , as the resp_port in both flows is the same. However, the resp_port distance between $f_{0_B}, f_{1_B} \in Seq_1$ is 55, which is the difference between port 80 and port 25. The inter-flow similarity of Seq_1 and Seq_2 is the distance of (0,55), thus is 55 . Normalized, resulting in a dissimilarity of 0.5.

sim = 3(0.665)+2(1 - 0.03)+6(1 - 0.142)+1(1- 0.5) = 9.5

An analysis based solely on the similarity of individual packets is not very accurate for Cybersecurity Attacks, as example a single ICMP could be interpreted as harmless, but a set of ICMPs can be a factor of DOS attack identification.
On the basis of the above during this stage I have analyzed the data as a group of packets, in other words, said as an N-flow.
In particular, I have based my analysis on the remarks of the paper [23] and used it to group the N-flow starting from 3 packets to 6 packets.
For the next step we need to select the most significant N-flow based on the data mined, starting from the matrices produced I have calculated the average of all the N-flow of each family.
During this analysis, I have found some interesting statistics presented in the following table 5.4.

In particular, for the N-flow of dimension 5, we have a good number of packets, and also, a more important thing, the presence of the highest similarity for 3 flows found in the PCAP 2020-02-26-Trickbot-spreads-from-infected-client-to-DC almost attributable to equality.

In order to take into account the presence of this equality I have re-arranged the scores of N-flow similarity with a factor of occurrence, see the below formula:

family + flow dimension	dictionary dimension	pcap	occurrences
dridex-nflow3	44	132	0
dridex-nflow4	24	96	0
dridex-nflow5	16	80	0
dridex-nflow6	10	60	0
ramnit-nflow3	30	90	0
ramnit-nflow4	20	80	0
ramnit-nflow5	18	90	0
ramnit-nflow6	13	78	0
trickbot-nflow3	238	714	0
trickbot-nflow4	117	468	0
trickbot-nflow5	70	350	3
trickbot-nflow6	51	306	0

Table 5.4. Data-mining statistics

N-flow score = average similarity * occurrence

As a result, the following table shows, in order of relevance for each family, the most relevant pcaps from which the best performance of classification is expected, Table 5.5.

Dridex	dridex-blog-07-2019
Dridex	dridex-blog-12-2020
Dridex	dridex-blog-05-2020
Dridex	strato-dridex
Dridex	dridex-blog-2021
Ramnit	ramnit-blog-2017-06-02-first
Ramnit	ramnit-blog-2017-12-28
Ramnit	ramnit-blog-2017-08-25
Ramnit	ramnit-blog-2016-05-16-first
Ramnit	ramnit-blog-2017-06-02
Ramnit	ramnit-strato-2018-04-03
Trickbot	2017-03-07-trickbot-strato
Trickbot	2019-09-25-trickbot-blog
Trickbot	2018-07-21-trickbot-blog
Trickbot	2019-07-02-trickbot-blog
Trickbot	2020-09-08-trickbot-blog
Trickbot	2017-08-15-trickbot-blog
Trickbot	2020-02-26-trickbot-blog
Trickbot	2018-11-12-trickbot-blog
Trickbot	2018-09-03-trickbot-blog
Trickbot	2017-08-21-trickbot-blog
Trickbot	2020-06-16-trickbot-blog
Trickbot	2020-11-09-trickbot-blog-tar2
Trickbot	2018-12-07-trickbot-blog
Trickbot	2021-02-17-trickbot-blog
Trickbot	2019-01-11-trickbot-blog

Table 5.5. Most relevant pcaps for training.

5.4 Machine learning tools and algorithm for Malware Identification and Classification

This paragraph will describe the tools and algorithms used to perform the identification and classification of malware. As already discussed in section 5.1.4, I used the scikit-learn library to accomplish the classification and identification tasks. It is important to note that each algorithm has somewhat different implementations among the many libraries, in case of replication and different results, the reason could be the previous one.

The scikit-learn library provides several classifiers like support vector machine, random forest, gradient boosting and k-nearest neighbors, etc.

5.4.1 Dataset splitting

As the first step, the dataset has to be split into at least train and test or if it is possible in train, validation, and test. This can be done in different ways, the most common is by using the train_test_split() function in which it is also possible to specify the percentage destined for the train and the test.
Whenever we change the random_state parameter present in train_test_split(), we get different accuracy for different random_state and hence we cannot exactly point out the accuracy for our model [42]. In the cited article there are also some examples explaining the random sampling, the stratified sampling, and the Stratified K-Fold Cross validation that can be summarized as follows.
Random sampling can result in skewed class distribution, impacting model performance. Stratified sampling, a more accurate approach, ensures proportional class representation. In stratified sampling, the population's characteristics are preserved in both the training and test sets.
Moreover, standard K-Fold Cross-Validation helps with the random_state issue, but it still relies on random sampling. Stratified K-Fold Cross-Validation applies stratified sampling during each fold, overcoming both the accuracy inconsistency and class distribution concerns.
As an example in figure (Figure 5.2), there is a flowchart of five-fold cross-validation.
Furthermore, there is also RepeatedStratifiedKFold that allows improving the estimated perfor-

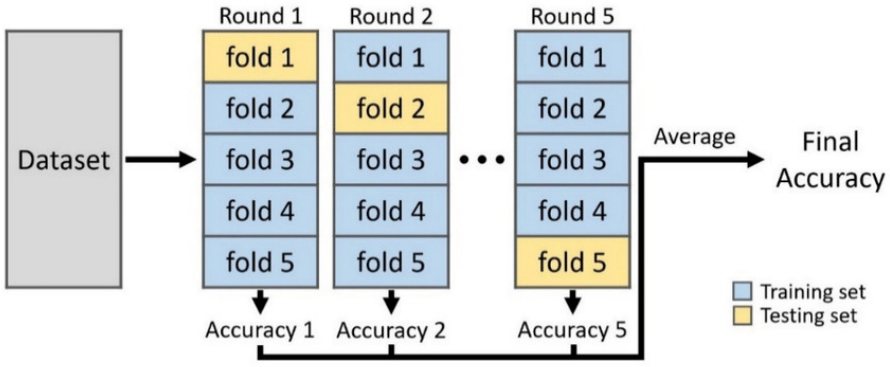

Figure 5.2. Flowchart of five-fold cross-validation.

mance of a machine learning model, by simply repeating the cross-validation procedure multiple times (according to the n_repeats value) and reporting the mean result across all folds from all runs. This mean result is expected to be a more accurate estimate of the model's performance [43].
Finally, there is also the possibility to build your own splits based on your own experiences or data mining results which is the most time-consuming method but also the most effective one.
During the development of this thesis only the last two ways, in particular, I used to build by myself the dataset splits based on the results of the data mining and, in order to have a method

of comparison, I have chosen to use the RepeatedStratifiedKFold, which turned out to be the best method given the class imbalance at the expense of a greater time required.

5.4.2 Machine learning Algorithms

In this section I will illustrate the machine learning algorithms that I have used, they have already been discussed in chapter 3 paragraph 4 but in a generic way, below I will specify how they have been configured and used.

Random Forest

This algorithm is one of the most used in the whole literature of machine learning, simplicity is its strong point, thus offering very low times for train and test. These were the reasons for choosing this algorithm and the overall results rewarded the choice.

As regards the configurable hyperparameters for the Random Forest algorithm, the only two are the number of estimators and the weights attributed to the features. For the second, if not specified, they are attributed during training, thus leaving the possibility for the algorithm to attribute them once the data has been processed. Initially, I thought about changing the weight attributed to the features only once I learned better what the behavior of the model was, but subsequently, I re-evaluated this option because, from the research conducted, changing the weights attributed to the features would force the model to have a certain behavior going against the basic principle of machine learning and unless you are particularly expert, it is strongly discouraged to carry out this type of tuning. Instead for the number of estimators it was carried out, at the beginning they were used with values 100 and 1000 to then move on to a search at each run varying from one hundred to one hundred specifying the range of interest (1000, 2500 and 3000).

Gradient Boosting

Thanks to the advice of Professor Atzeni and my colleague Sarracino Marco, I reevaluated the use of the Gradient Boosting algorithm. Sarracino showed me how the algorithm performed well during his thesis work, so I decided to give it a chance to this algorithm which is mainly used in image and video classification activities.

Gradient Boosting is a machine learning algorithm that combines the predictions of multiple base models to create a stronger overall predictor, typically based on decision trees.
The whole process can be summarized in the sequential construction of the model which is based on the improvement of the weaknesses of the previous ones. This is granted by analyzing the errors made in the predictions of the predecessor models, thus adapting the subsequent model.
The term “gradient” in Gradient Boosting refers to the process of improving the hyperparameters that generate prediction errors in order to minimize them as much as possible.
What differs most from the random forest is the inability to parallelize the creation of the next decision trees as these are based on the improvement of the previous ones thus leading to greater accuracy at the expense of longer training times.
It is widely used in regression and classification tasks, what sets Gradient Boosting apart is its proficiency in handling complex relationships within data and usually outperforms random forest.

As regards the configuration of the model, I searched for the best hyperparameters through consecutive iterative cycles, varying one hyperparameter at a time and keeping the others fixed. This approach turned out to be quite time-consuming but to obtain precisely which are the hyperparameters that best fit there are no alternatives.
The configurable hyperparameters are the number of estimators, the learning rate, and the maximum depth. The first iterative cycles were focussed on the research of the number of estimators and the learning rate. Once obtained, the focus shifts to finding the best maximum depth.

K-Nearest Neighbours

The K-Nearest Neighbours algorithm is another widely used algorithm, the predictions it makes are based on the concept of proximity considering the "k" closest data points from the dataset. This algorithm is as simple to configure as it is complex to interpret, unfortunately it is not possible to consult or vary the weights to be attributed to the features, and in many cases, it is difficult to understand why data is attributed to one class rather than another.

For the previously discussed reasons, only iterative cycles have been carried out in search of the best value of "k".

Chapter 6

Results

In this chapter, we will discuss the results obtained from the experimentation phase and how the tests have gradually evolved on the basis of the intermediate results. It should be noted that, sometimes, some subsequent tests have lower results than the previous ones, these should not be understood as negative results as they also provide useful information for the progression of the final results but also for a better understanding of the model and the data used.

Before jumping into the tests and results it is good to get an introduction to the metrics used during this phase. The libraries used to build the diagrams and statistical data visualization are matplotlib [44] and seaborn [45].

As the first metric analysed we have the accuracy. This metric is one of the four must-have metrics for machine learning purposes, it indicates how many predictions are correct out of the total number of predictions. The formula is the following:

$$Accuracy = \frac{Numberofcorrectpredictions}{Totalnumberofpredictions} = \frac{TP + TN}{TP + TN + FP + FN}$$

The major issue for accuracy is due to the inability to track an unbalanced dataset. As well as in the examples of [42], in order to best deal with unbalanced datasets, the necessary precautions are needed, as regards the metrics, the use of precision, and recall are very important in the case of class imbalance. Precision expresses the amount of true positives out of the total number of positives, and is defined as follows:

$$Precision = \frac{TP}{TP + FP}$$

Note that, TP stands for true positives and FP stands for false positives.
Recall instead, expresses the number of true positives compared to the number of true positives and false negatives and therefore takes the form of:

$$Recall = \frac{TP}{TP + FN}$$

By analyzing the precision and recall values in pairs, it is possible to understand how the model behaves with a certain type of class. Through the analysis of the precision, we can understand if the model can handle cases of overlapping of classes well but also of generic false positives, which can be caused by classes with very different data from each other as in my case with the Dridex class. Instead, with the recall analysis we can understand how the model manages to manage all positive cases belonging to that class.
In the context of malware detection/classification these two metrics can also be analyzed to determine what percentage of benign data is blocked (precision) and instead with what percentage of malicious data is not blocked (recall).

Precision and recall offer a trade-off, i.e., one metric comes at the cost of another. More precision involves a harsher critic (classifier) that doubts even the actual positive samples from the dataset, thus reducing the recall score. On the other hand, more recall entails a lax critique that allows any sample that resembles a positive class to pass, which makes border-case negative samples classified as "positive," thus reducing the precision. Ideally, we want to maximize both precision and recall metrics to obtain the perfect classifier [46].
For these reasons, the F1 score metric was introduced which seeks to simultaneously maximize both precision and recall, and its formula is the following:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN}$$

Furthermore, in the case of multi-classification, the metrics are divided into three sub-categories: micro, macro, and weighted.

- **Micro**, calculate metrics globally by counting the total true positives, false negatives, and false positives.
- **Macro**, calculate metrics for each label and find their unweighted mean. This does not take label imbalance into account.
- **Weighted**, calculate metrics for each label and find their average weighted by support (the number of true instances for each label). This alters "macro" to account for label imbalance; it can result in an F-score that is not between precision and recall.

In our case, the micro category is not used because of the class imbalance.

6.1 Results with full dataset without manual splitting into train and test

In this paragraph, I will analyze the first test that was carried out, without taking into account the data mining results in order to confirm or refute them. Based on the previous statements, the test was conducted on the three machine learning algorithms discussed in section 5.4.2, namely, random forest, gradient boosting, and k-nearest neighbors. In addition to the metrics analysis, an analysis is also conducted on the most relevant features for each type of model.

6.1.1 Random Forest

The test was conducted on the Random Forest algorithm with two values of n_estimators as suggested by the library, 10 and 100, but both led to the same result of accuracy as 0.6677.
But more interesting are the features that are considered interesting by the model, they are expressed in numerical value in the table 6.1 and graphically in the image 6.1 And finally, the table 6.2 lists all the metrics obtained from the model.

6.1.2 Gradient Boosting

The test was conducted on Gradient Boosting researching the best parameters for n_estimators, learning_rate, and max_depth hyperparameters. The research led to the following results:

- learning_rate= 0.1
- n_estimators = 400
- max_depth = 9

With the previous values, the model reported an accuracy of 0.67 with the values of all the metrics shown in the table 6.3. As far as the most important features of the model are concerned, we can observe them precisely in the table 6.4 and in a graphical representation in the image 6.2.

resp_port	0.229786
orig_ip_bytes	0.127955
orig_bytes	0.099031
service	0.096726
resp_ip_bytes	0.093841
protocol	0.071644
orig_pkts	0.056362
resp_pkts	0.052757
resp_bytes	0.037571
history_5	0.035878
conn_state	0.029607
history_6	0.020348
history_3	0.015575
history_4	0.014839
history_7	0.005377
history_8	0.005223
history_0	0.004857
history_2	0.001592
history_1	0.001031

Table 6.1. Random Forest: feature scores

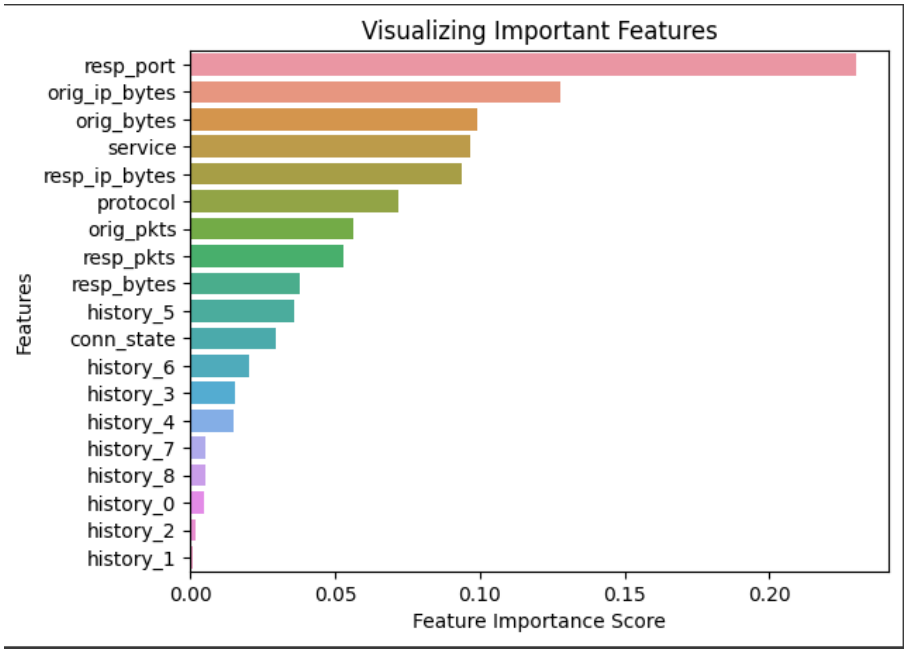

Figure 6.1. Random forest: feature importance

6.1.3 K-Nearest Neighbours

Instead, the KNN algorithm, as already anticipated in the previous chapters, does not make many hyperparameters available for modification, the only hyperparameter that can be varied is the value of K. For these reasons, the test focused on making as many K values as possible in a reasonable time to be able to compare the trend of the model to its growth.

The figure 6.3 shows the accuracies of the model as k increases for both the train and the test in the interval of k from 1 to 8. In order to have a more detailed view in light of the previous results obtained, I preferred to carry out a further test with the values of k from 6 to 20 which are shown

	precision	recall	f1-score	support
dridex	0.74	0.28	0.40	40912
ramnit	0.99	0.14	0.24	5282
trickbot	0.65	0.95	0.77	66131
macro avg	0.79	0.46	0.47	112325
weighted avg	0.70	0.67	0.61	112325
accuracy			0.67	112325

Table 6.2. Random Forest: metrics scores

	precision	recall	f1-score	support
dridex	0.74	0.28	0.40	40912
ramnit	0.99	0.13	0.23	5282
trickbot	0.65	0.95	0.77	66131
macro avg	0.79	0.45	0.47	112325
weighted avg	0.70	0.67	0.61	112325
accuracy			0.67	112325

Table 6.3. Gradient Boosting: metrics scores

resp_port	0.356633
orig_bytes	0.114690
protocol	0.103021
resp_ip_bytes	0.098471
orig_ip_bytes	0.084555
resp_bytes	0.081427
service	0.072104
orig_pkts	0.041040
resp_pkts	0.033041
conn_state	0.015017

Table 6.4. Gradient Boosting: feature scores

in the picture 6.4 and 6.5.

6.1.4 Comments

In this first paragraph, we have seen the first results of a model without major optimizations and with a division into train and test based on the simplest of methods.

Despite the low optimizations, the model has already reported some good metrics. It was noted that the difference in the amount of data made the difference between the various classes, in fact, the class relating to the trickbot malware was found to be the one with the highest metrics, see precision/recall and F1 score, while for the other classes of malware, the metrics turn out to be quite low.
Instead, the feature scores attributed to the random forest and gradient boosting models are much more interesting, they appear to be profoundly different from each other and in the case of gradient boosting they rely too much on the response port which turns out to be non-optimal in terms of resistance during the years.

Finally, as far as the KNN algorithm is concerned, its results are slightly lower than the two other contending algorithms but what is the main demerit is the times, as far as the random

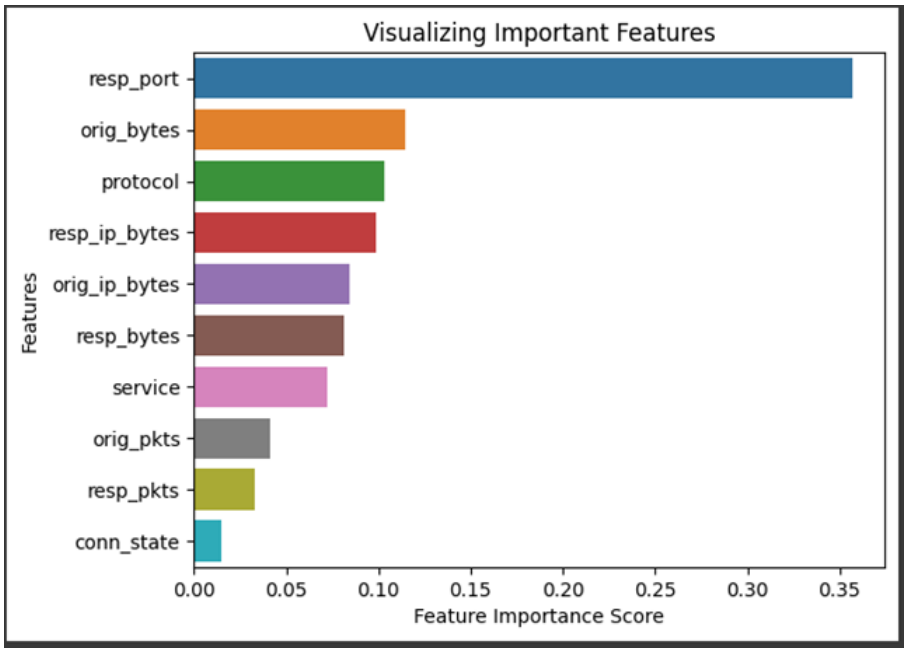

Figure 6.2. Gradient boosting: feature importance

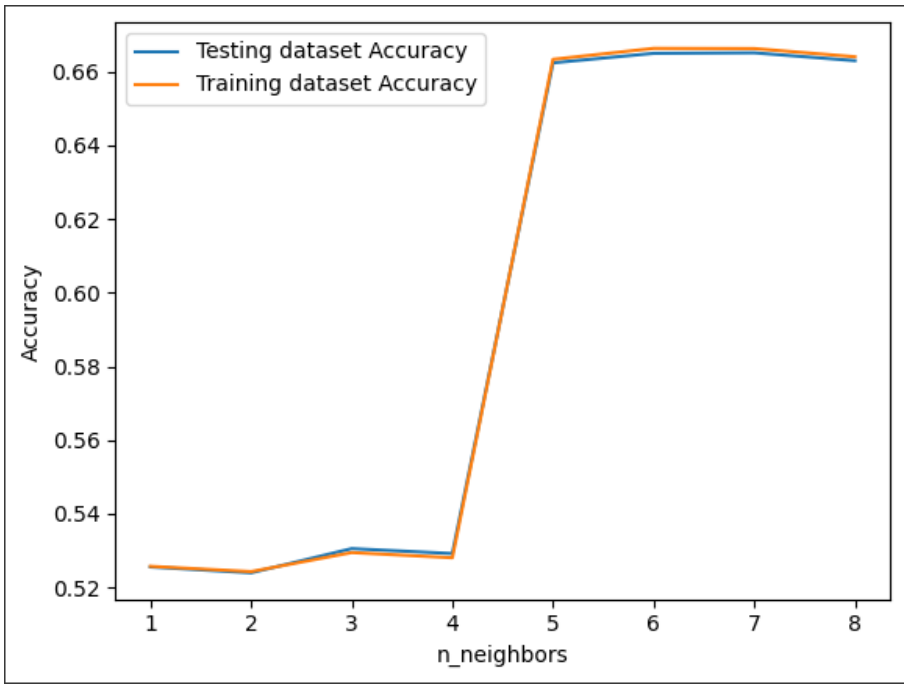

Figure 6.3. KNN: Accuracy per k-value 1 to 8 (train and test)

forest is concerned, it turns out to be the fastest, with the architecture used and described above, there are maximum times of thirty minutes while for gradient boosting they are just over an hour and finally, for KNN they are about three hours.

In summary, the test objective was achieved in a good state, the random forest algorithm, based on the feature scores, was the one that had a very similar trend to what was sought during data mining and together with its extreme train speed would seem to be the optimal model for this task.

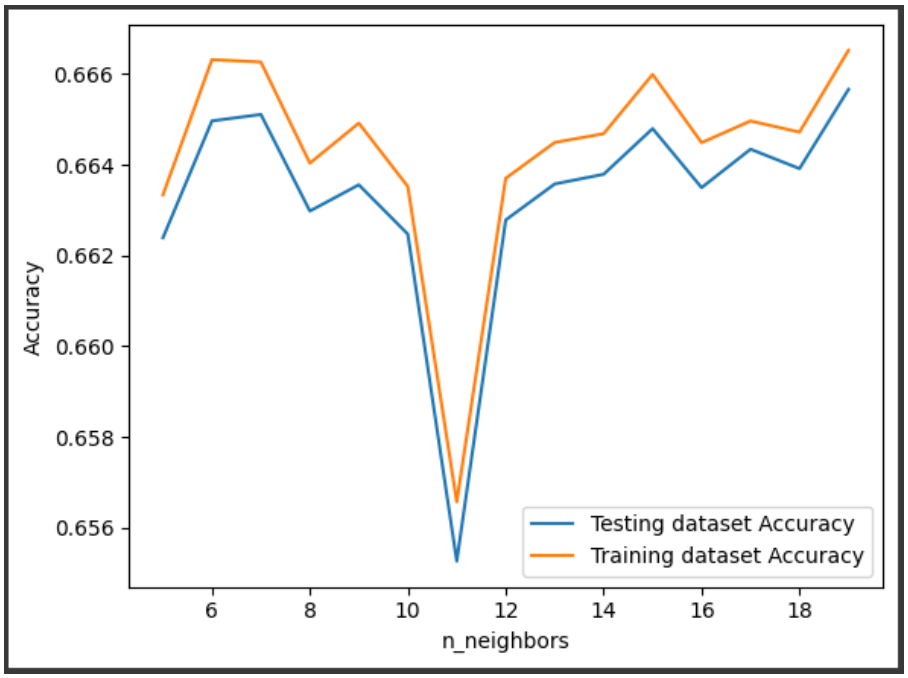

Figure 6.4. KNN: Accuracy per k-value 6 to 20 (train and test)

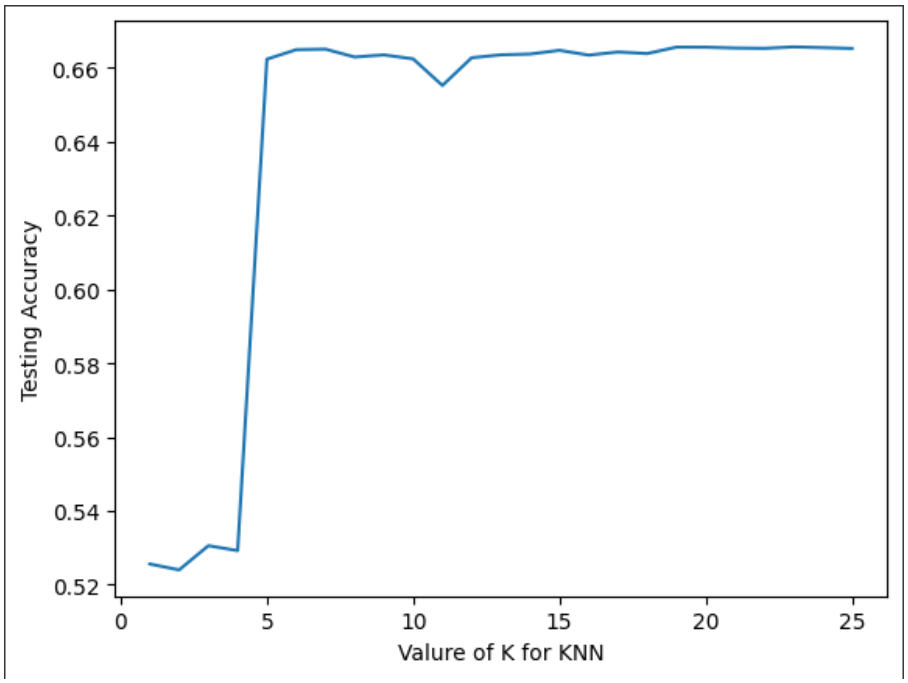

Figure 6.5. KNN: Accuracy per k-value (test)

The gradient boosting model had similar statistics to the random forest except for the importance given to the features that seem to be a little resistant to the evolution of malware. Finally, KNN's model turned out to be the worst performer both in terms of time and in terms of statistics.

On the basis of the previous observations, the second test was carried out, carrying out a manual division between train and test on the basis of the results of the data mining.

6.2 Results with dataset split with data mining evidence

Following the observations of the previous paragraph, in this section, I will analyze the new test results against the manual division of the dataset into train and test.

During the test, the same guidelines of the previous test were followed, therefore always using the same variation of the hyperparameters for random forest, gradient boosting, and KNN.

Also, from this test, I will start to take into consideration the distribution of the classes with respect to the set of trains and tests.

6.2.1 Random Forest

The test conducted with the random forest algorithm produced the same accuracy results for both the values of n_estimators, 10 and 100, which are 0.9799.
Instead, the importance weights attributed to the features by the model are slightly more homogeneous than in the previous configuration. They are expressed in numerical value in the table 6.5 and graphically in the image 6.6 At the end, table 6.6 shows the values of the metrics

resp_port	0.210744
orig_bytes	0.146240
resp_ip_bytes	0.145165
orig_ip_bytes	0.107908
resp_pkts	0.102206
conn_state	0.064387
resp_bytes	0.061919
service	0.039638
orig_pkts	0.034938
history_7	0.030770
protocol	0.021575
history_5	0.010510
history_4	0.007994
history_6	0.006857
history_8	0.005720
history_2	0.001816
history_3	0.001519
history_1	0.000088
history_0	0.000007

Table 6.5. Random Forest: feature scores

obtained from the random forest model.

	precision	recall	f1-score	support
dridex	0.05	0.04	0.04	138
ramnit	0.00	0.02	0.00	164
trickbot	1.00	0.98	0.99	175439
macro avg	0.35	0.35	0.35	175741
weighted avg	1.00	0.98	0.99	175741

accuracy			0.98	175741

Table 6.6. Random Forest: metrics scores

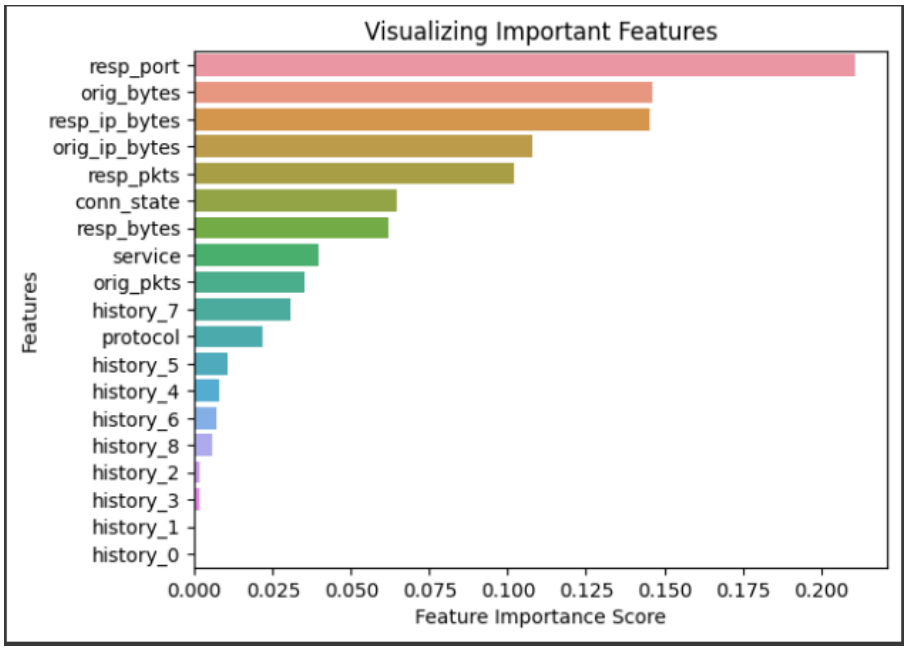

Figure 6.6. Random forest: feature importance

6.2.2 Gradient Boosting

After the tests with the random forest algorithm, the tests with the gradient boosting algorithm followed. From the iterative sequential search, the following optimal values of the hyperparameters were deduced.

- learning_rate= 0.01
- n_estimators = 300
- max_depth = 10

With the previous values, I obtained accuracy values equal to 0.8945491872980393 for the train and 0.15797679539777285 for the test. Instead, the final values of the metrics are reported in table 6.7. As regards the importance values attributed to the features by the model, in the case

	precision	recall	f1-score	support
dridex	0.00	0.04	0.01	138
ramnit	0.00	0.08	0.00	164
trickbot	0.99	0.19	0.32	175439
macro avg	0.33	0.10	0.11	175741
weighted avg	0.99	0.19	0.32	175741

accuracy			0.19	175741

Table 6.7. Gradient Boosting: metrics scores

of gradient boosting the evolution of the weights is much more interesting than in the random forest counterpart. They are listed in the table 6.8 and in a graphical representation in the image 6.7.

resp_port	0.298973
orig_bytes	0.214656
resp_pkts	0.200905
resp_ip_bytes	0.141377
history_7	0.058701
orig_ip_bytes	0.025192
service	0.020103
resp_bytes	0.012543
protocol	0.010159
orig_pkts	0.006414
conn_state	0.004293
history_4	0.002363
history_5	0.001933
history_8	0.001776
history_3	0.000258
history_2	0.000170
history_6	0.000161
history_0	0.000017
history_1	0.000007

Table 6.8. Gradient Boosting: feature scores

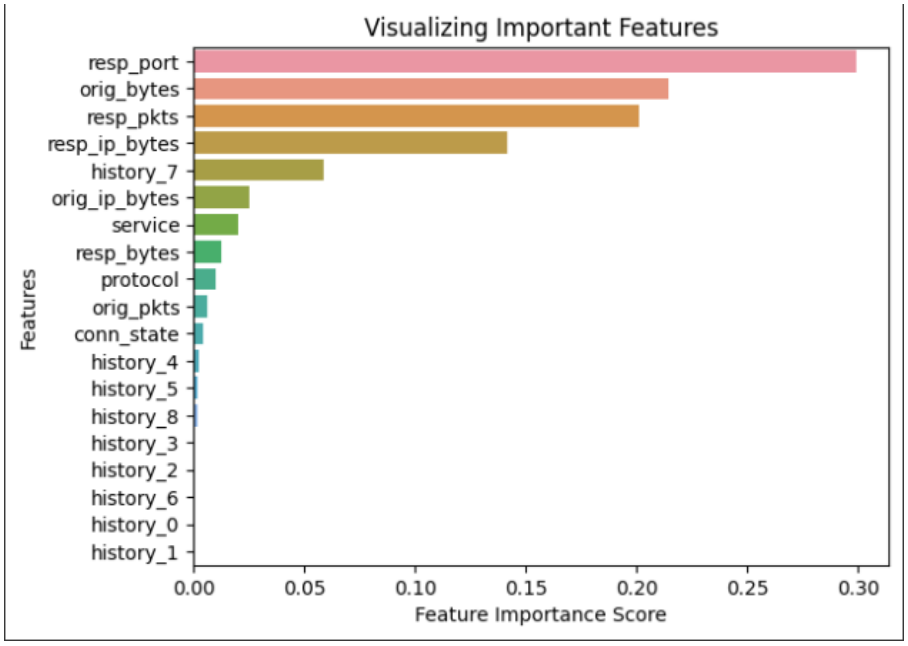

Figure 6.7. Gradient boosting: feature importance

6.2.3 K-Nearest Neighbours

Based on previous experience, the test conducted on the KNN algorithm was carried out with the value of k ranging from 1 to 20. The figure 6.8 describes the trend of the accuracy with the variation of k, both for train and for test.

The highest accuracy recorded by the KNN model is with the value of n_Neighbours equal to 3 which turns out to be 0.13029401221115164 on test. Table 6.9 instead lists all the metrics obtained with n_Neighbours equal to 3.

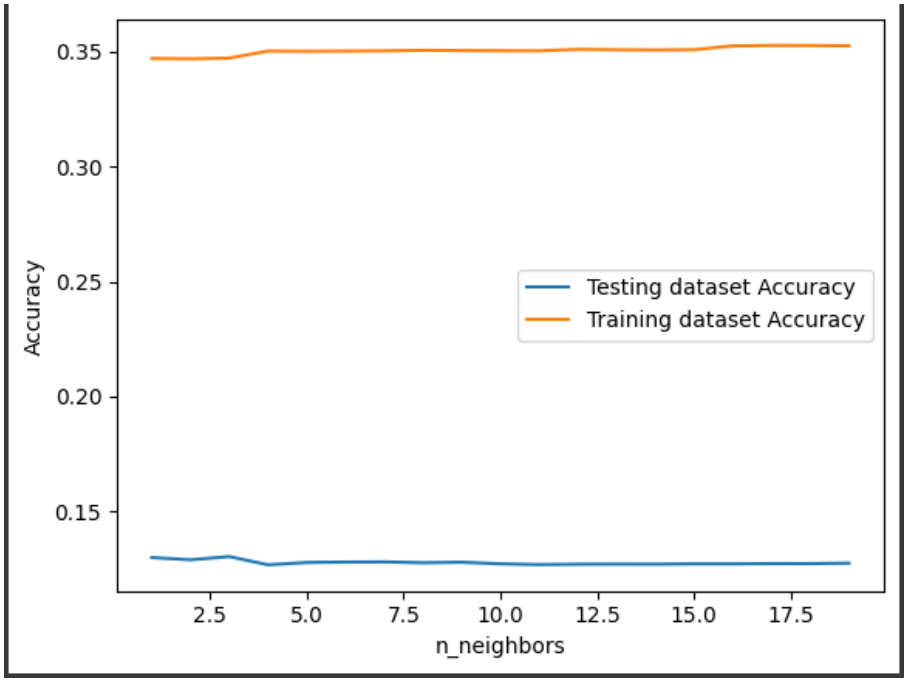

Figure 6.8. KNN: Accuracy per k-value 1 to 20 (train and test)

	precision	recall	f1-score	support
dridex	0.01	0.28	0.01	138
ramnit	0.00	0.07	0.00	164
trickbot	0.99	0.13	0.23	175439
macro avg	0.33	0.16	0.08	175741
weighted avg	0.99	0.13	0.23	175741
accuracy			0.13	175741

Table 6.9. K-Nearest Neighbours: metrics scores

6.2.4 Comments

During this paragraph, the various metrics for each model were analyzed with the respective weights attributed to the features. Before going on with the considerations it is important to observe the graphs of the distributions of the classes with respect to the train and test sets. Figure 6.9 shows the ratio of the classes to the whole train split which is of:

- Trickbot = 123300 (74.892%),
- Dridex = 25321 (15.380%),
- Ramnit = 16015 (9.728%).

Instead figure 6.10 shows the ratio of the classes to the whole test split which is of:

- Trickbot = 175439 (99.828%),
- Dridex = 164 (0.093%),
- Ramnit = 138 (0.079%).

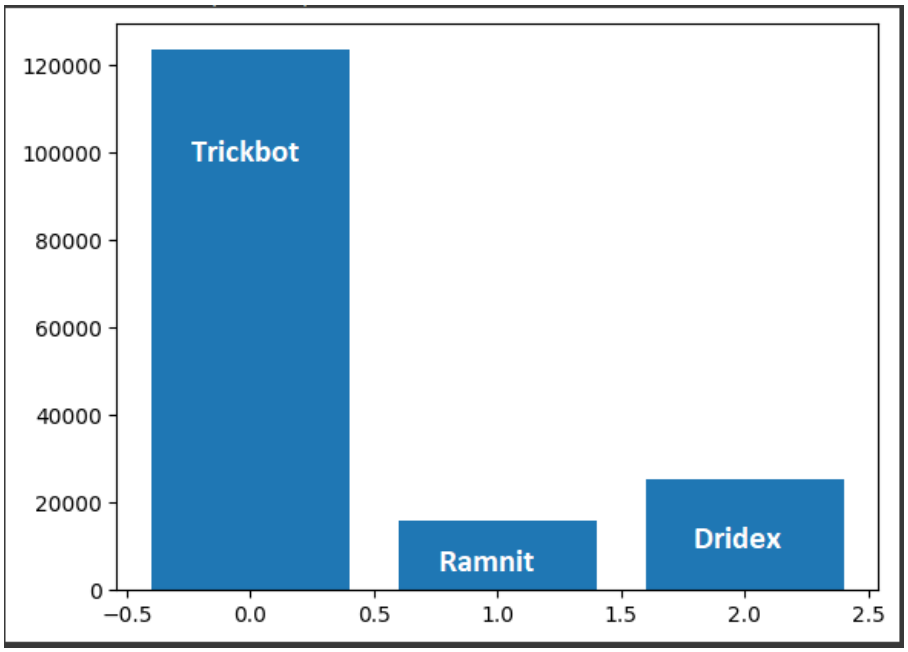

Figure 6.9. Distribution of classes in the train split

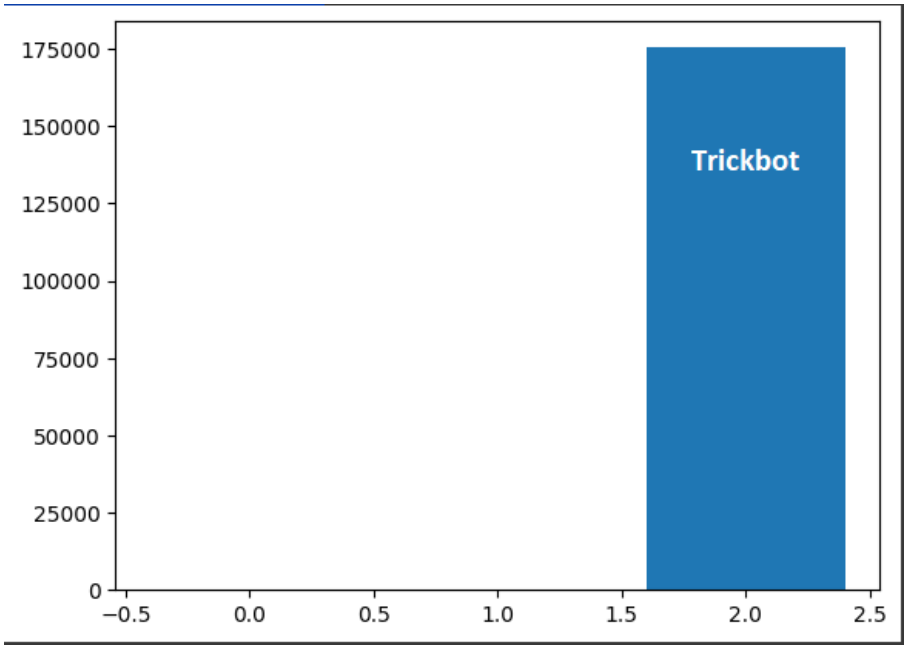

Figure 6.10. Distribution of classes in the test split

After the quick preamble on the distribution of the classes, what is highlighted in the metric values for random forest and gradient boosting is an excellent management of the trickbot class while all three models reported very low statistics in the management of the other two malware families.

By inspecting the curves of the accuracy graph for the KNN model and the accuracy values in test and train for the gradient boosting model, it is easy to see that the model has gone heavily overfitted. This means that the model has learned to classify the train data too accurately, resulting in it not being able to classify as well on the test portion.

As far as the weights attributed to the features are concerned, there appears to be a slight balance in the case of random forest while a considerable improvement was noted in the case of gradient boosting. The features relating to the history are still of little importance, the percentages of which are very low.

In summary, in this test phase, based solely on the results obtained during data mining, I obtained a too precise model on the train data which scales badly on the predictions that do not fit into it (overfitting).
Instead, the weights of the features have normalized concerning the initial test, resulting in more compliance with what was obtained from the data mining phase.
The algorithm that has best managed the task is the random forest, as the metrics for the recognition of the class of trickbot are the highest. The Dridex and Ramnit malware families are under-recognized, and during this phase, it seems to be mainly due to class imbalance.

Therefore, the next tests will focus on class imbalance and the search for greater accuracy and recall for the Dridex and Ramnit malware families.

6.3 First results with expanded dataset manually split with only Trickbot and Dridex malware families

After several tests spent trying to optimize the metrics considering only the trickbot and dridex malware families, I concluded that there was a problem in recognizing the dridex family, especially in terms of recall. So what happens is that the system hardly makes a mistake when it classifies a datum as belonging to the dridex class (precision) but on the other hand very often when it does not consider it is actually belonging to it (recall), with the result of a very precise but insensitive to the dridex class.

As an example, table 6.10 lists the metrics obtained from the random forest model with n_estimators value equal to 300 whose recorded accuracy is 0.9923557141037075. Only the most relevant statistics among the tests carried out are shown below, table 6.10 lists the metrics obtained from the random forest model with n_estimators value equal to 300 whose recorded accuracy is 0.9923557141037075, while the most relevant features according to the model are those shown in figure 6.11. Based on the previous considerations, further searches were carried out in order

	precision	recall	f1-score	support
dridex	0.90	0.14	0.24	138
trickbot	0.99	1.00	1.00	15560
macro avg	0.95	0.57	0.62	15698
weighted avg	0.99	0.99	0.99	15698
accuracy			0.99	15698

Table 6.10. Random Forest: metrics scores

to find other examples of pcap related to dridex traffic. They were processed in the same way as the previous ones through the Zeek firmware, converted into .csv format, and finally encoded before being integrated into the dataset. Table 6.11 lists the new files added to the dataset. The motivation to search for new samples for the dridex family is to make the system more sensitive to it, providing more examples the system will rebalance the weights between the classes trying to be more balanced.

After expanding the dataset, the distribution of classes obtained for train and test is shown in figures 6.9 and 6.10. Figure 6.9 illustrates how much as a percentage of the train is:

- Trickbot = 185200 (59.824%),

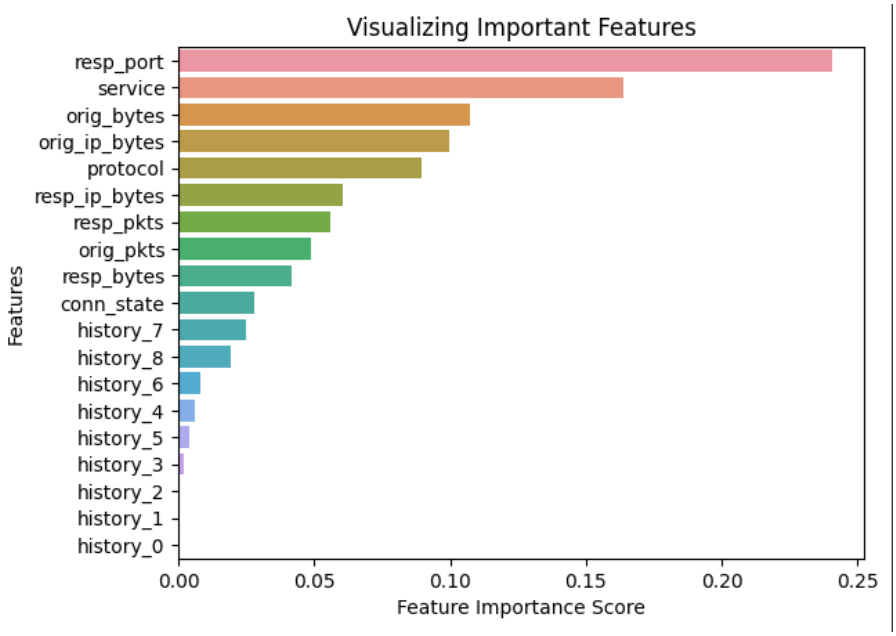

Figure 6.11. Random forest: feature importance

Pcap name	Number of flow
2016-02-01-Dridex-infection	95
2017-03-30-Dridex-booking-malspam	6
2017-03-30-Dridex-confirmation-letter	213
2017-06-05-Dridex	4
2017-12-04-Dridex	26
2019-06-17-passw-protected-word-dridex	25
2019-10-28-Ursnif-Dridex	491
2019-10-30-Ursnif-Dridex	558
2019-12-11-Ursnif-Dridex	427
2020-04-29-Dridex	99
2020-06-03-Dridex	14
2020-09-24-Dridex	113
2020-11-12-Dridex	37
2018-01-25-Dridex	9
2018-01-25-Dridex-2	61
2019-07-09-password-protec-Drid	27
2019-07-12-Dridex	268
2020-05-14-Dridex	279
2018-12-10-Dridex-tempry	236
2018-12-10-Dridex	73
2020-05-11-Dridex	32

Table 6.11. New pcaps added to the previous dataset

- Dridex = 124373 (40.176%).

Figure 6.10 illustrates how much as a percentage of the test is:

- Trickbot = 15560 (92.990%),
- Dridex = 1173 (7.010%).

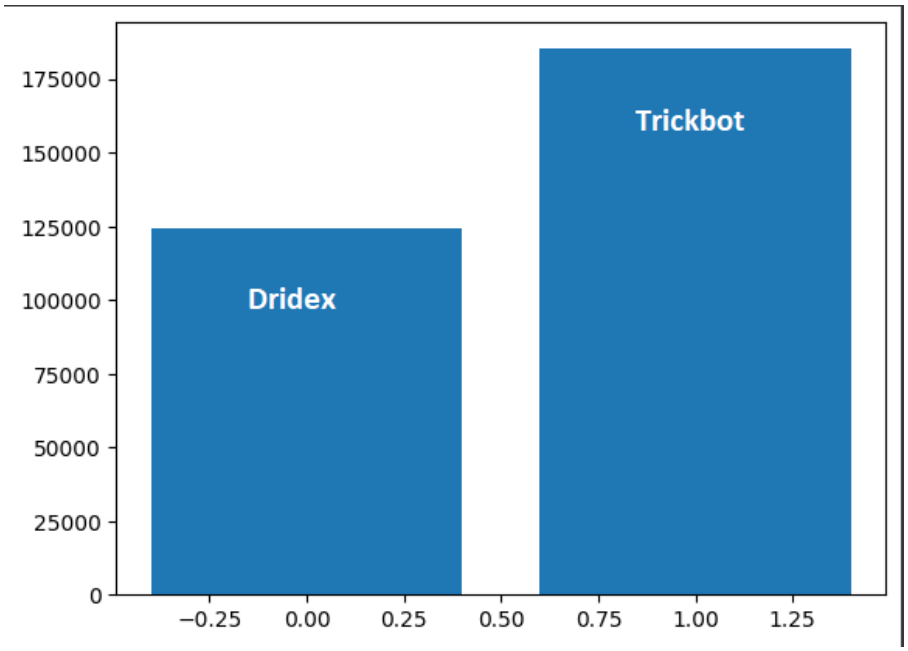

Figure 6.12. Distribution of classes in the train split

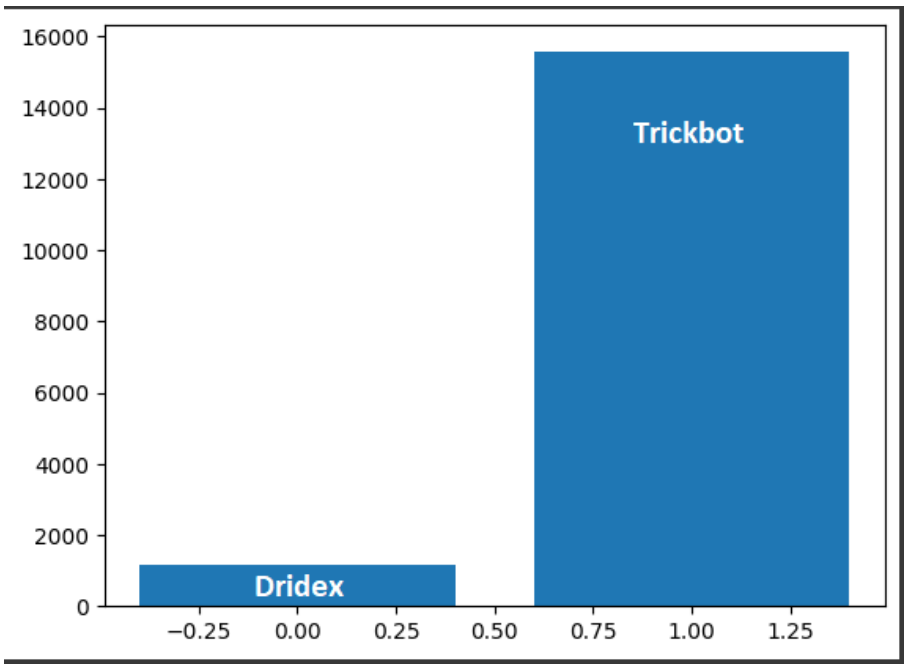

Figure 6.13. Distribution of classes in the test split

6.3.1 Random Forest

The test conducted on the random forest was based on the iteration of the n_estimators hyperparameter with values from 100 to 1400 but these changes were not reflected in a real improvement as can be seen from the graph in figure 6.14.

The greatest accuracy recorded is 0.9300185262654634 and table 6.12 lists the obtained values of the metrics.

Instead, the weights that the model attributes to the features are graphically illustrated in the image 6.15 and listed in the table 6.13.

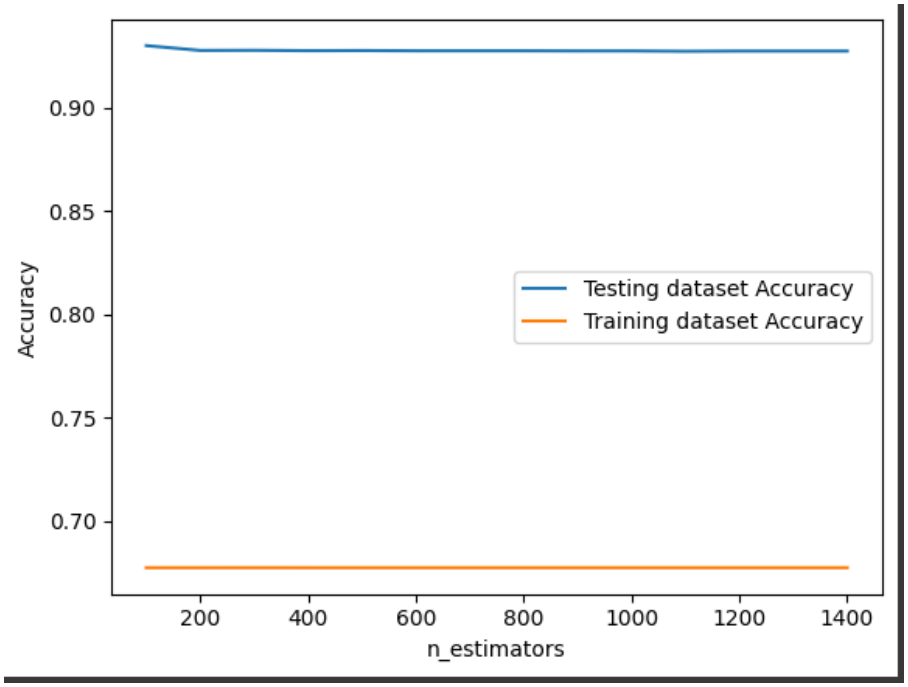

Figure 6.14. Random Forest: Accuracy per n_estimators 1 to 1500 (train and test)

	precision	recall	f1-score	support
dridex	0.45	0.15	0.22	1173
trickbot	0.94	0.99	0.96	15560
macro avg	0.69	0.57	0.59	16733
weighted avg	0.90	0.93	0.91	16733
accuracy			0.93	16733

Table 6.12. Random Forest: metrics scores

6.3.2 Gradient Boosting

The test carried out with the gradient boosting model did not bring great results and a first red flag could be the hyperparameters identified as the best they are:

- learning_rate= 0.1
- n_estimators = 300
- max_depth = 7

With the previous values, we obtained accuracy values equal to 0.6723131707872279 for the train and 0.8943405247116476 for the test. Instead, the final values of the metrics are reported in table 6.7. The doubts resulting from the analysis of the hyperparameters are confirmed with the analysis of the weights associated with the features from the model which are graphically represented in the image 6.16 and listed promptly in the table 6.15.

6.3.3 K-Nearest Neighbours

Finally, the analysis with the KNN model is followed with the same considerations of the previous tests and therefore with the values of k with a range from 1 to 20. The trend of the accuracy concerning the number of Neighbours is shown in the graph in the image. Instead, the values of the metrics obtained are shown in the table 6.16 with the value of k equal to 4.

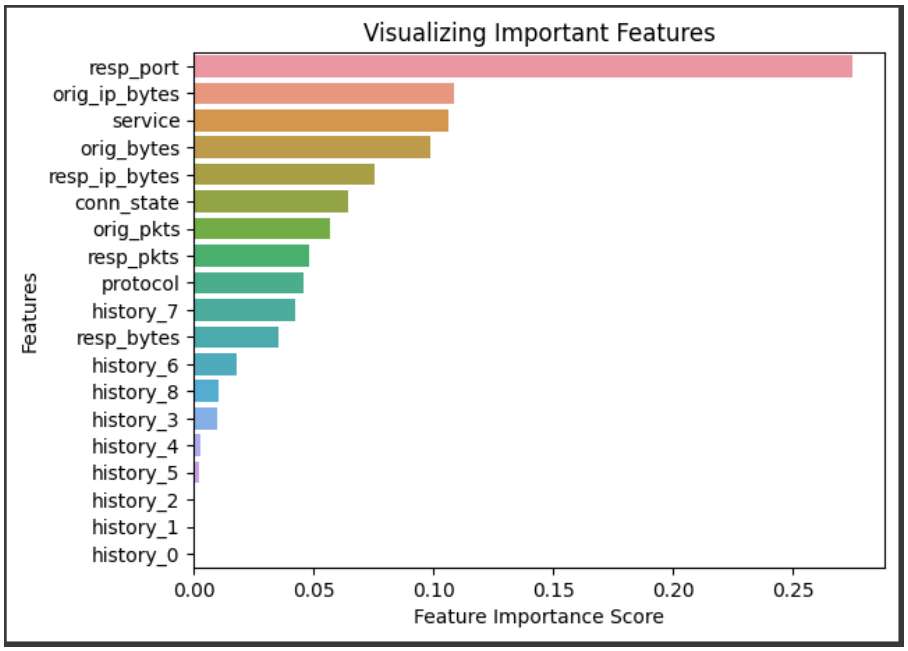

Figure 6.15. Random forest: feature importance

resp_port	0.274631
orig_ip_bytes	0.108314
service	0.106330
orig_bytes	0.098694
resp_ip_bytes	0.075495
conn_state	0.064646
orig_pkts	0.056937
resp_pkts	0.047898
protocol	0.045559
history_7	0.042097
resp_bytes	0.035414
protocol	0.021575
history_6	0.018067
history_8	0.010314
history_3	0.009934
history_4	0.002480
history_5	0.002322
history_2	0.000523
history_1	0.000322
history_0	0.000023

Table 6.13. Random Forest: feature scores

6.3.4 Comments

In this section, we have analyzed how the random forest, gradient boosting and KNN models have evolved based on an expansion of the dataset.

What immediately catches the eye is how the gradient-boosting model reacted completely negatively to this change. As previously anticipated, both the hyperparameters and the weights associated with the features show a drastic worsening since both the maximum depth of the trees and the learning speed combined with the prevalent use of the response port for classification

	precision	recall	f1-score	support
dridex	0.12	0.09	0.10	1173
trickbot	0.93	0.95	0.94	15560
macro avg	0.53	0.52	0.52	16733
weighted avg	0.88	0.89	0.88	16733
accuracy			0.89	16733

Table 6.14. Gradient Boosting: metrics scores

resp_port	0.525217
orig_bytes	0.139065
conn_state	0.103688
resp_ip_bytes	0.078511
service	0.063560
resp_bytes	0.029272
orig_ip_bytes	0.021292
orig_pkts	0.013211
resp_pkts	0.012537
protocol	0.008316
history_3	0.002108
history_8	0.000870
history_6	0.000752
history_5	0.000623
history_4	0.000609
history_7	0.000186
history_1	0.000096
history_2	0.000083
history_0	0.000003

Table 6.15. Gradient Boosting: feature scores

	precision	recall	f1-score	support
dridex	0.26	0.63	0.37	1173
trickbot	0.97	0.87	0.92	15560
macro avg	0.62	0.75	0.64	16733
weighted avg	0.92	0.85	0.88	16733
accuracy			0.85	16733

Table 6.16. K-Nearest Neighbours: metrics scores

make the model highly unstable.

Instead, as far as the random forest model is concerned, the results remain almost unchanged. For the dridex class there was a notable lowering of the precision and a slight improvement in the recall which can be summarized in a change of 2% while for trickbot there was a lowering of the precision dictated by the expansion of the dataset, but which did not change its performance is very high (3%).

On the contrary, the model with KNN had a clear improvement, passing from an accuracy of 13% with the three classes to one of 85% with the two classes. The other metrics have also increased significantly for the dridex class compared to before the dataset enlargement as demonstrated by the summary table 6.17.

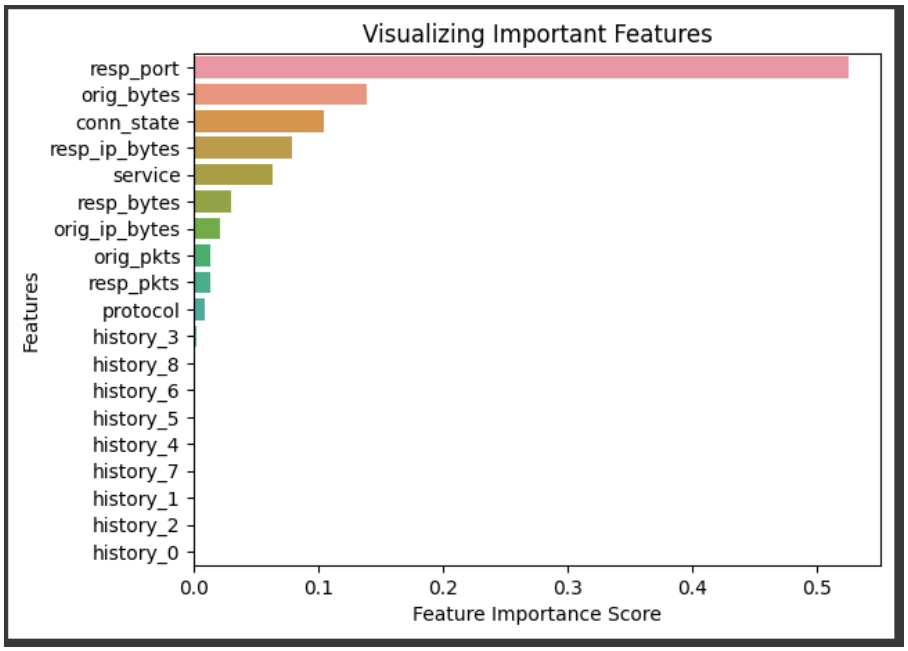

Figure 6.16. Gradient Boosting: feature importance

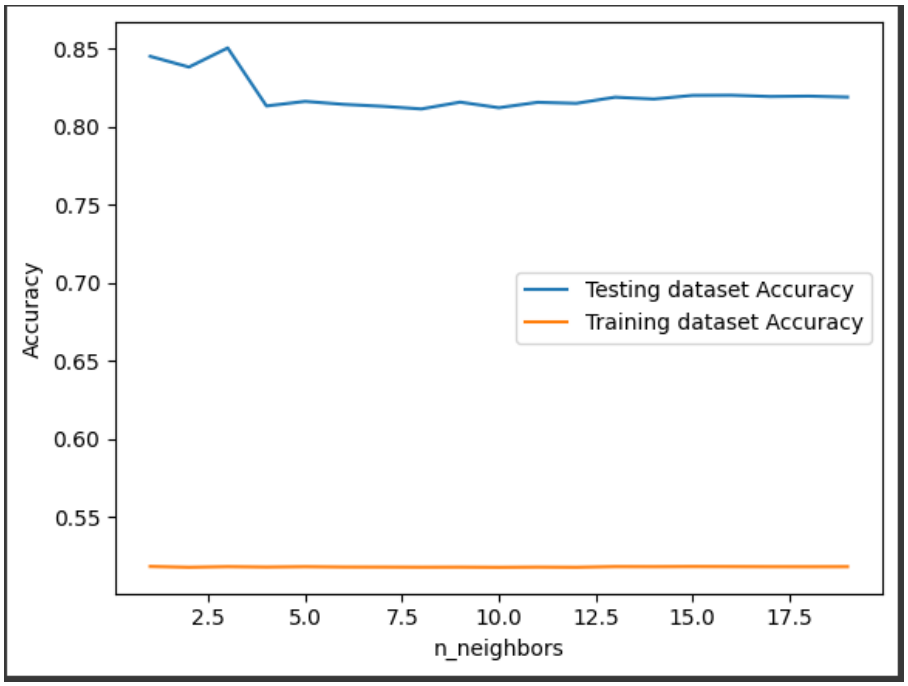

Figure 6.17. KNN: Accuracy per k-value 1 to 20 (train and test)

On the basis of the previous considerations, it can be seen how important data mining is since the simple insertion of other example cases of a class has improved slightly in some cases (random forest) and worsened a lot in others (gradient boosting) contrary to all desired effect, which is why in the subsequent tests a better distribution of data between train and test was followed and subsequently the test with one of the best cross-validation methods trying to avoid the data mining phase.

	precision	recall	f1-score
dridex	0.04 ->0.26	0.55 ->0.63	0.07 ->0.37
trickbot	1.00 ->0.97	0.88 ->0.87	0.94 ->0.92
macro avg	0.52 ->0.62	0.72 ->0.75	0.51 ->0.64
weighted avg	0.99 ->0.92	0.88 ->0.85	0.93 ->0.88
accuracy			0.88 ->0.85

Table 6.17. K-Nearest Neighbours evolution after dataset expansion

6.4 Final results with expanded dataset (Trickbot and Dridex)

Based on the previous analysis, numerous tests were performed in order to improve the performance of the models and to search for the best model for this type of task.

There were in particular two very important tests, one of these obtained the maximum f1-score for the dridex class while the other had the best overall performance among the two classes. The two tests were obtained with splits of different datasets, in the event that the classification values of the dridex class were higher, the train dataset was very unbalanced in favor of that class with a percentage of 77.332%, while that more balanced was obtained with a more balanced and broader train with percentages of 40.132% for the dridex class and 59.868% for the trickbot class (figures 6.18 and 6.19).

In this paragraph, I will analyze the second test in particular, which recorded the highest

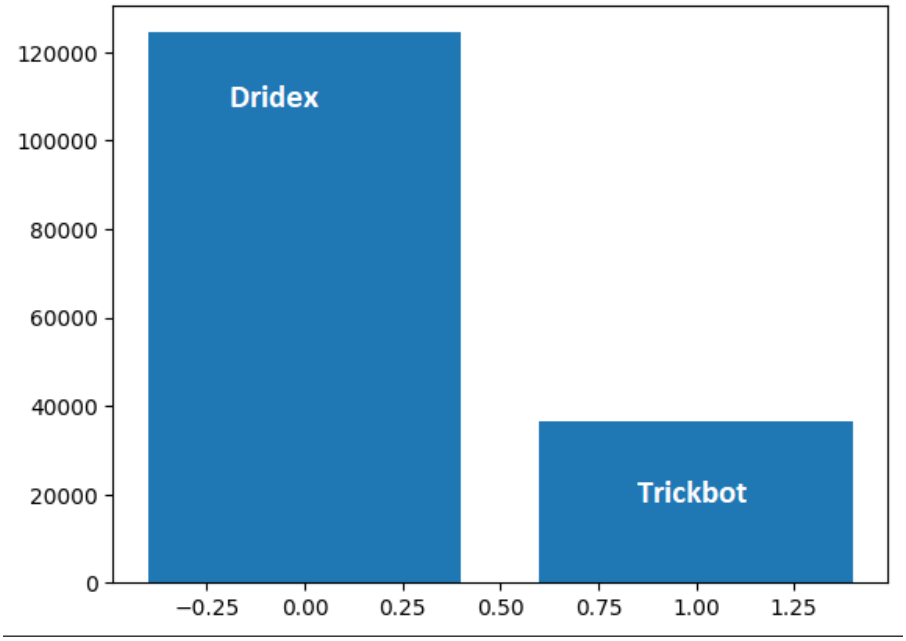

Figure 6.18. Distribution of classes in the dridex imbalanced train split

overall performance and not the highest performance compared to the dridex family, because from the first it was deduced that although more information can be given to the model than the dridex class, manages to greatly improve its classification, this topic will be further explored in the "conclusions" chapter on the basis of other research.

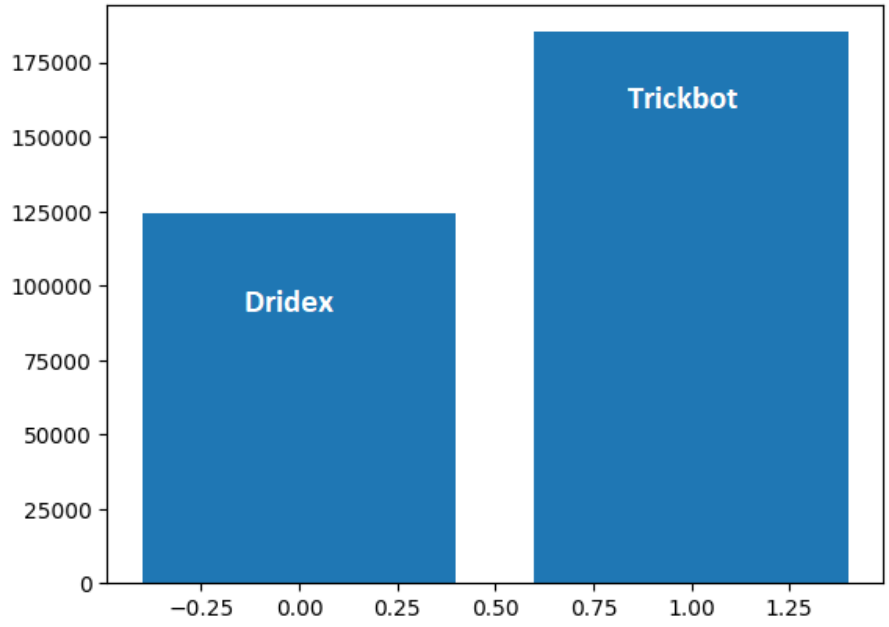

Figure 6.19. Distribution of classes in the balanced train split

6.4.1 Random Forest

The test conducted on the random forest model produced an almost uniform result as the value of n_estimators varied, in particular, the oscillation on the test is about 1% while the train is even less (figure 6.20). Based on the previous analysis, the following metrics were extrapolated

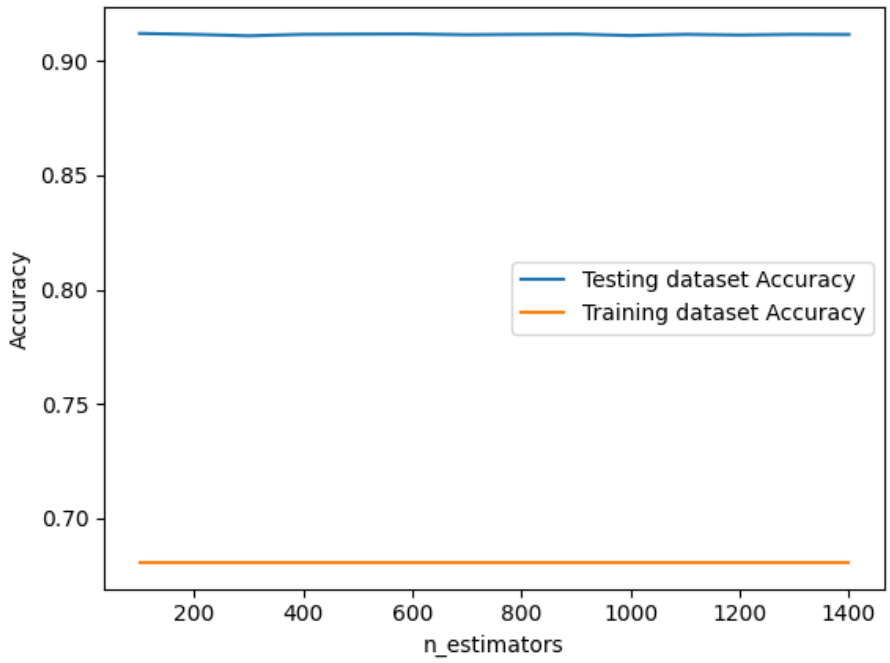

Figure 6.20. Random Forest: Accuracy per n_estimators 1 to 1500 (train and test)

for the value of n_estimators equal to 1400 and expressed in the table 6.18. The importance of the features recognized by the model remained almost the same from the previous tests and are reported in the figure 6.21 and numerically in the table 6.19.

	precision	recall	f1-score	support
dridex	0.59	0.20	0.30	1617
trickbot	0.92	0.99	0.95	15560
macro avg	0.76	0.59	0.63	17177
weighted avg	0.89	0.91	0.89	17177
accuracy			0.91	17177

Table 6.18. Random Forest: metrics scores

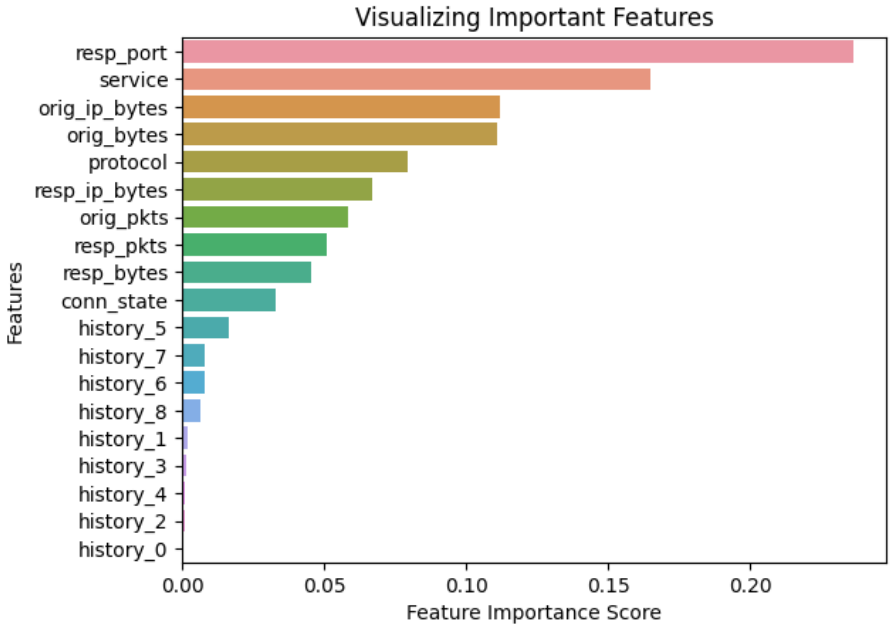

Figure 6.21. Random forest: feature importance

resp_port	0.236060
service	0.164920
orig_ip_bytes	0.108314
orig_bytes	0.110667
protocol	0.079200
resp_ip_bytes	0.066845
orig_pkts	0.058085
resp_pkts	0.050894
resp_bytes	0.045524
conn_state	0.032782
history_5	0.016406
history_7	0.008043
history_6	0.007696
history_8	0.006154
history_1	0.001789
history_3	0.001147
history_4	0.000972
history_2	0.000867
history_0	0.000010

Table 6.19. Random Forest: feature scores

6.4.2 Gradient Boosting

The model obtained from gradient boosting is almost always the same for all tests, the only ones that differ more are the ones we will analyze below with the one obtained from the unbalanced dataset for dridex. In particular, for the latter, the result is much lower, and also the weights of the features seem to have had a regression.

Returning to the analysis example case, the best hyperparameters obtained are the following:

- learning_rate= 0.1
- n_estimators = 300
- max_depth = 9

Through the previous hyperparameters, a train score of 0.672804333221098 and a test score of 0.9148279676311346 were obtained, with the consequent metrics shown in the table 6.20. As

	precision	recall	f1-score	support
dridex	0.39	0.30	0.34	1617
trickbot	0.93	0.95	0.94	15560
macro avg	0.66	0.63	0.64	17177
weighted avg	0.88	0.89	0.88	17177
accuracy			0.89	17177

Table 6.20. Gradient Boosting: metrics scores

previously mentioned, the values assumed by the feature weights seem to have regressed towards the previous models and are represented in the graph in the figure 6.22 and in the table 6.21.

service	0.020103
orig_bytes	0.204257
resp_port	0.403415
protocol	0.010159
orig_pkts	0.006414
resp_ip_bytes	0.141377
resp_bytes	0.012543
resp_pkts	0.200905
orig_ip_bytes	0.025192
conn_state	0.004293
history_7	0.058701
history_6	0.002363
history_8	0.001933
history_3	0.001776
history_5	0.000258
history_1	0.000170
history_4	0.000161
history_2	0.000017
history_0	0.000007

Table 6.21. Gradient Boosting: feature scores

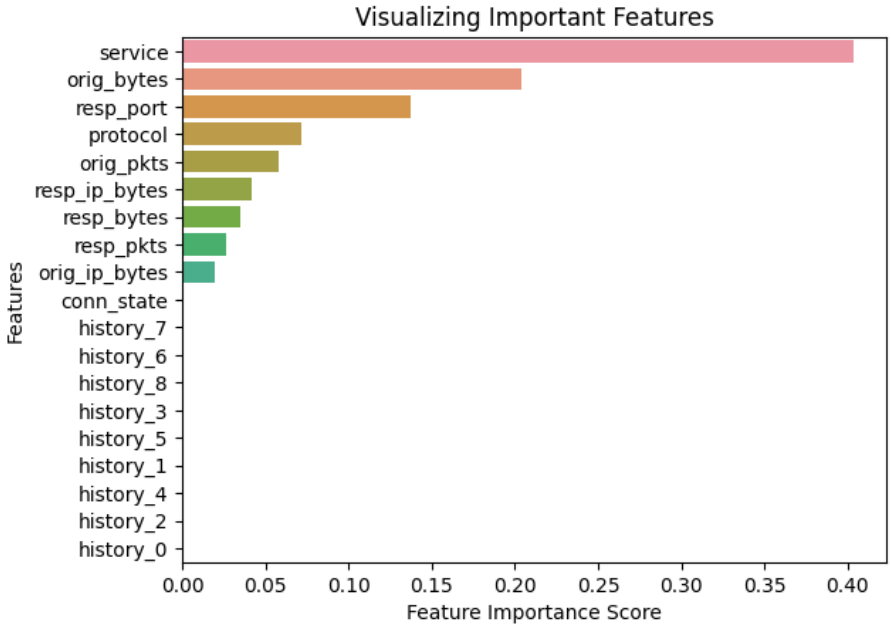

Figure 6.22. Gradient boosting: feature importance

6.4.3 K-Nearest Neighbours

Finally, the best results were recorded for the model obtained from KNN, obtaining 68% of the F1-score on the macro average. All the metrics are listed in the table 6.22, while figure 6.23 shows the trend graph of the train and test accuracy recorded throughout the experimentation.

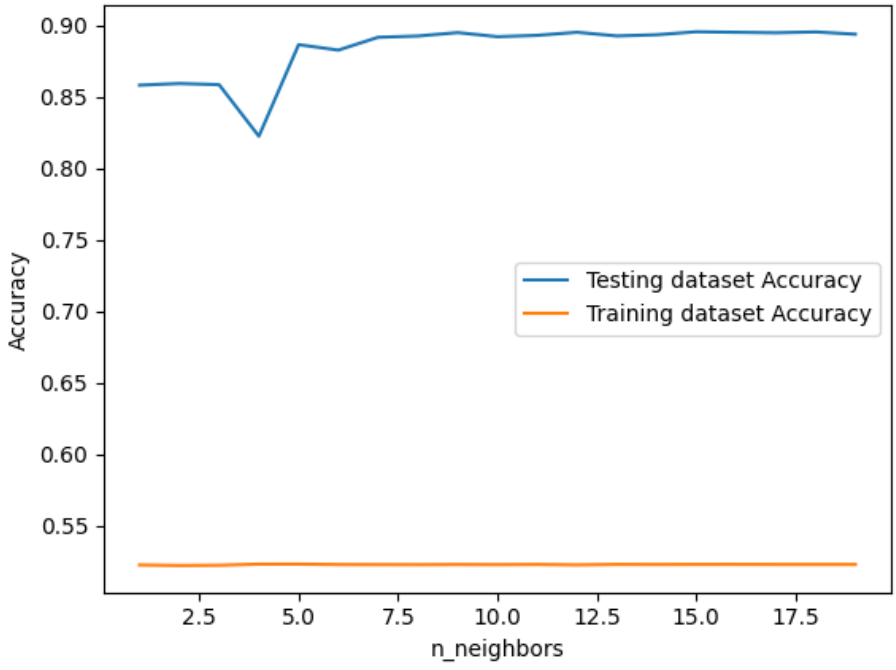

Figure 6.23. KNN: Accuracy per k-value 1 to 20 (train and test)

	precision	recall	f1-score	support
dridex	0.44	0.40	0.42	1617
trickbot	0.94	0.95	0.94	15560
macro avg	0.69	0.67	0.68	17177
weighted avg	0.89	0.90	0.89	17177
accuracy			0.90	17177

Table 6.22. K-Nearest Neighbours: metrics scores

6.4.4 Comments

The results of this experiment aimed at improving the performance of the models in the classification of the dridex class and in the search for the best algorithm to carry out this type of test has produced good results as it has gone from a maximum F1-score of 24 % registered with the random forest at 42% with KNN algorithm. It might seem like a slight improvement when speaking unequivocally of the F1-score but if you inspect in detail you can see that, at the expense of a 46% difference in accuracy, there was a 26% improvement in recall which overall brought an improvement of 10% on the macro average recall and 6% on the macro average of the F1-score. Normally one would have expected an improvement with a greater margin but the dridex malware family has had a great evolution over the years thus leading to an additional difficulty in its classification.

Another point of observation is to be placed in the example with an unbalanced split train on the dridex class which should have led to a greater sensitivity of the system to it but which reported only a 12% more on the recall and a 10% more on the F1 -score.

All these results will be better analyzed in the conclusions chapter by comparing them with the research of the evolutions of the dridex malware during the decade.

The next tests will focus on the Trickbot and Dridex malware families following the operational practices used and refined during the analysis of the Trickbot and Dridex malware families.

6.5 Results with expanded dataset (Trickbot and Ramnit)

This test was conducted on the Trickbot and Ramnit malware families, following as a guideline the path previously done for the Trickbt and Dridex malware families, in order to better understand what the issues are in classifying the Ramnit malware family.

Since the amount of data concerning these two families is smaller than the previous ones, the tests conducted took advantage of the shorter training time requirement to carry out longer tests by increasing the ranges of the hyperparameters.

With regard to the dataset, the samples relating to the ramnit family have also been slightly increased compared to the first two tests, the table 6.23 shows the pcaps that have been added in this phase.
The distribution of classes in the train (figure 6.24) and test (Figure 6.25) splits is shown in the following graphs, while the percentages are as follows for train and test.
Train:

- Trickbot = 25321 (60.720%),
- Ramnit = 16380 (39.280%).

Test:

Pcap name	Number of flow
2018-01-09-Ramnit	95
2018-01-28-Rig-EK-Ramnit	6
2018-01-29-Rig-EK-Ramnit	213
2018-01-30-Rig-EK-Ramnit	4

Table 6.23. New pcaps added to the previous dataset

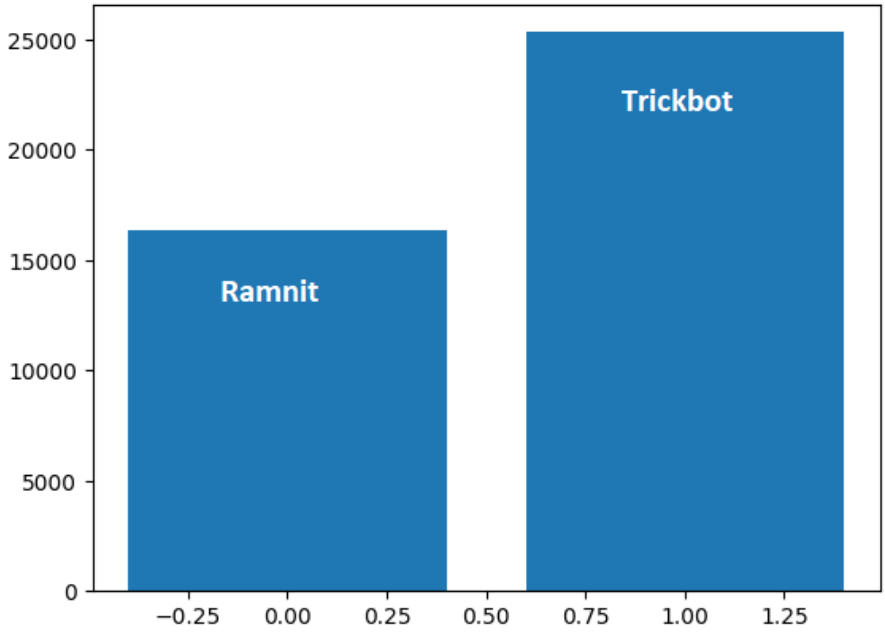

Figure 6.24. Distribution of classes in the train split

- Trickbot = 15560 (96.923%),
- Ramnit = 494 (3.077%).

6.5.1 Random Forest

As anticipated previously, as the size of the data to be processed by the model decreased, a corresponding decrease in training times was noted, therefore it was possible to increase the range of variation of the hyperparameters reaching a maximum number of n_estimators equal to 2900.

The figure 6.26 shows the accuracy trends which elect 100 as the best value of n_estimators followed immediately by 2600/2800/2900 with equal merit, whose accuracy is 0.9755201195963623 (n_estimators = 100) and 0.9750218014202068 (for the remaining ones).
. The variation is very low, and the metrics obtained reflect the result as for all four cases they are equivalent, the table 6.24 lists their values. Instead, the feature scores also in this case are not very homogeneous and are listed in the table 6.25 and represented in the figure 6.27.

6.5.2 Gradient Boosting

Tests conducted on the gradient boosting model turned out to be similar to those conducted with the previous two families, producing poor results. The best hyperparameters obtained produced a train score of 0.9204335747594007 and a test score of 0.9768904945807898 and were the following:

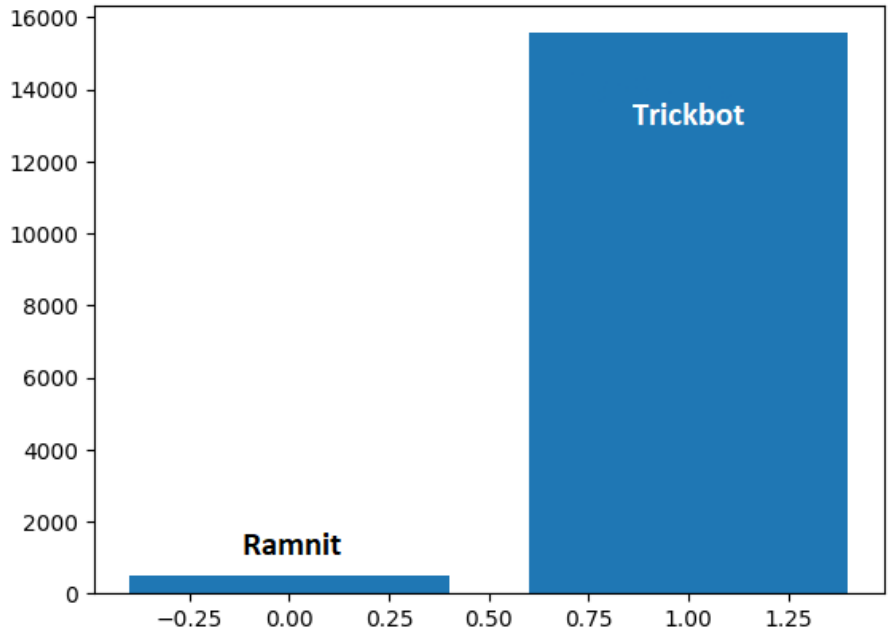

Figure 6.25. Distribution of classes in the test split

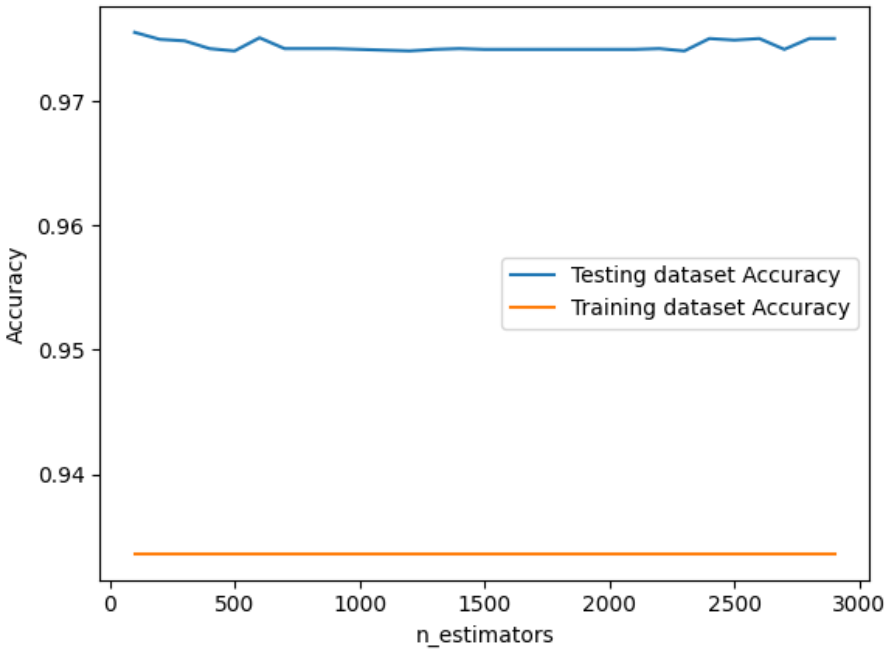

Figure 6.26. Random Forest: Accuracy per n_estimators 100 to 2900 (train and test)

- learning_rate= 0.1
- n_estimators = 400
- max_depth = 12

Related to the previous hyperparameters, the model obtained poor metric values in the classification of the ramnit class and are shown in the table 6.26. Even the values of the weights associated with the features from the model are very unbalanced between them and above all the

	precision	recall	f1-score	support
ramnit	0.80	0.25	0.38	494
trickbot	0.98	1.00	0.99	15560
macro avg	0.89	0.62	0.68	16054
weighted avg	0.97	0.98	0.97	16054
accuracy			0.98	16054

Table 6.24. Random Forest: metrics scores

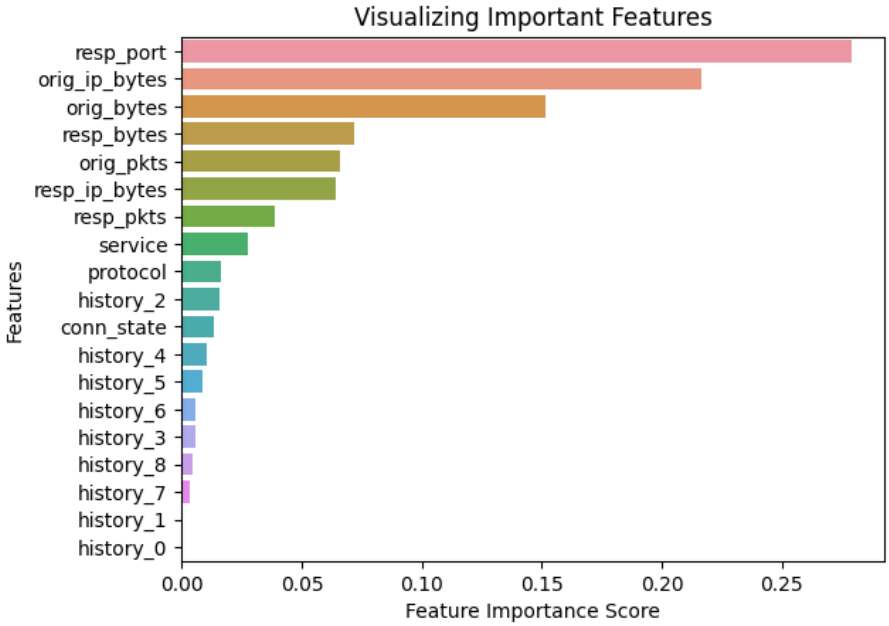

Figure 6.27. Random forest: feature importance

resp_port	0.278866
orig_ip_bytes	0.216341
orig_bytes	0.151444
resp_bytes	0.072044
orig_pkts	0.065697
resp_ip_bytes	0.064072
resp_pkts	0.039068
service	0.027577
protocol	0.016183
history_2	0.015483
conn_state	0.013189
history_4	0.010357
history_5	0.008928
history_6	0.005916
history_3	0.005644
history_8	0.004608
history_7	0.003579
history_1	0.000652
history_0	0.000353

Table 6.25. Random Forest: feature scores

	precision	recall	f1-score	support
ramnit	0.34	0.27	0.30	494
trickbot	0.98	0.98	0.98	15560
macro avg	0.66	0.63	0.64	16054
weighted avg	0.96	0.96	0.96	16054
accuracy			0.96	16054

Table 6.26. Gradient Boosting: metrics scores

higher values are on features that can be easily deceived by malware during their evolution. They are represented in the graph in the figure 6.28 and in the table 6.27.

resp_port	0.410486
resp_ip_bytes	0.179586
orig_bytes	0.103567
orig_ip_bytes	0.081407
conn_state	0.050056
history_8	0.044416
resp_bytes	0.038024
orig_pkts	0.031445
protocol	0.022282
resp_pkts	0.018940
history_4	0.006405
history_5	0.005261
service	0.003506
history_6	0.001304
history_0	0.001101
history_3	0.001099
history_2	0.001091
history_7	0.000025
history_1	0.000000

Table 6.27. Gradient Boosting: feature scores

6.5.3 K-Nearest Neighbours

The range of hyperparameters has also been extended for the KNN model, the maximum value of k has gone from 20 to 40, recording a downward curve as k increases. The figure 6.29 shows the accuracy values as k increases. For this purpose, two tests were performed on the metrics with the k values equal to 2 (table 6.29) and 20 (table 6.28).

	precision	recall	f1-score	support
ramnit	0.51	0.31	0.39	494
trickbot	0.98	0.99	0.98	15560
macro avg	0.74	0.65	0.69	16054
weighted avg	0.96	0.97	0.97	16054
accuracy			0.97	16054

Table 6.28. K-Nearest Neighbours: metrics scores with k = 20

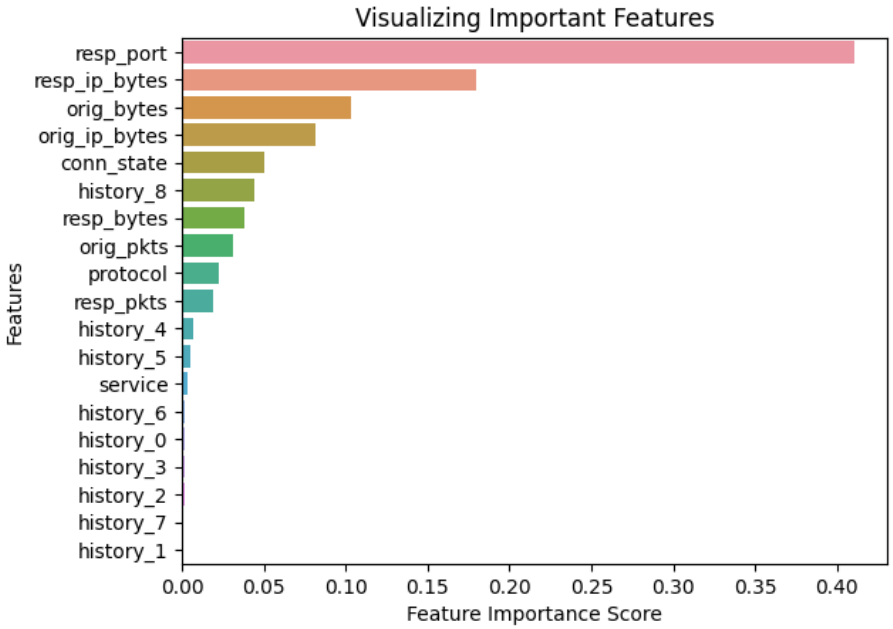

Figure 6.28. Gradient boosting: feature importance

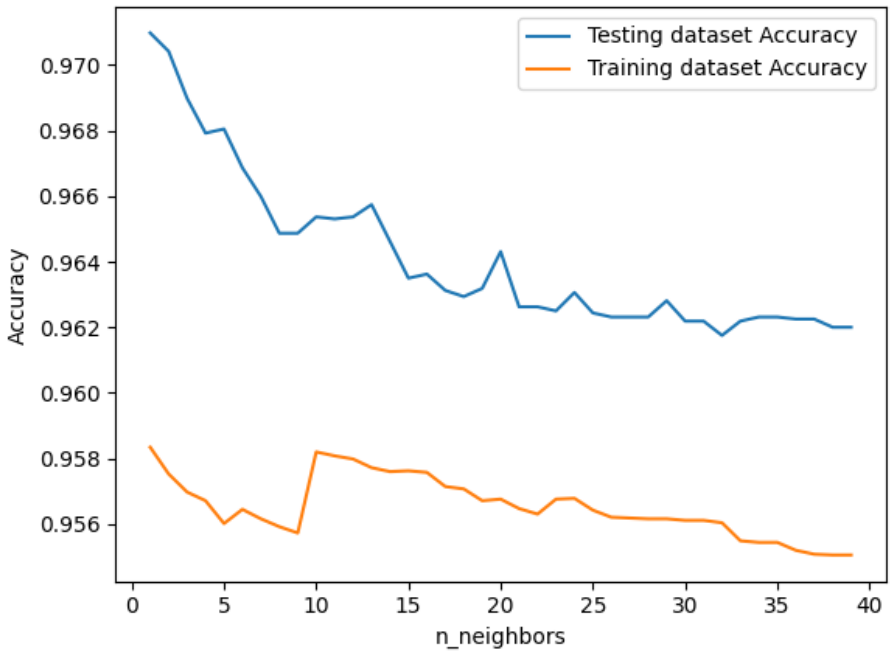

Figure 6.29. KNN: Accuracy per k-value 1 to 20 (train and test)

6.5.4 Comments

The test discussed in this paragraph underlined how much the difference between the amount of data of the two malware families (Trickbot and Ramnit) had an impact on the model and the metrics it produced.

Observing the values produced by all three models, it can be seen that the Trickbot malware family always obtains excellent results, while very low values are obtained for the Ramnit malware family. Furthermore, from the curves of the random forest model, it can be seen that also

	precision	recall	f1-score	support
ramnit	0.52	0.45	0.48	494
trickbot	0.98	0.99	0.98	15560
macro avg	0.75	0.72	0.73	16054
weighted avg	0.97	0.97	0.97	16054
accuracy			0.97	16054

Table 6.29. K-Nearest Neighbours: metrics scores with k = 2

in this case there are no oscillations on the train, suggesting a bias on the trickbot class.

In summary, the results produced by this test are totally inefficient for the classification of the Ramnit malware family, which is why the next test will be conducted with the two malware families with the least samples (Dridex and Ramnit) verifying how a better balance can impact between classes on the classification results of the various models.

6.6 Results with expanded dataset (Dridex and Ramnit)

This test was conducted on the Dridex and Ramnit malware families in order to have a complete view of the trends of the models developed with Random Forest, Gradient Boosting, and KNN. This test is the last of the pairs and following the trend of the previous ones, results were expected in line with the previous ones but thanks to it it was possible to identify the main problem of the previous tests.

The results obtained from this test, despite the strong class imbalance, are clearly positive for the classification of the Dridex malware family, completely unexpected behavior compared to the previous ones, while for the Ramnit class, the results are on average low even if slightly better than the previous ones.
As anticipated before, also in this case, since the samples of the Ramnit malware family are very few, there is a considerable difference in size both in train and in test between the two classes, below are the percentages for both and the their graphic representation (train 6.30 and test 6.31).
Train:

- Dridex = 124426 (88.367%),
- Ramnit = 16380 (11.633%).

Test:

- Dridex = 1730 (77.788%),
- Ramnit = 494 (22.212%).

6.6.1 Random Forest

As in the previous test, the training times are much lower in this one too, which is why in this test the range covered by the hyperparameters is larger and follows the previous one, with the n_estimator values reaching a maximum of 2900.
As can be seen in figure 6.26, the trend of the accuracy on the test is oscillatory with its peak recorded (0.8295863309352518) with the number of estimators equal to 1900, moreover it can be seen that from the value 2000, the behavior of the accuracy begins to vary becoming a decreasing

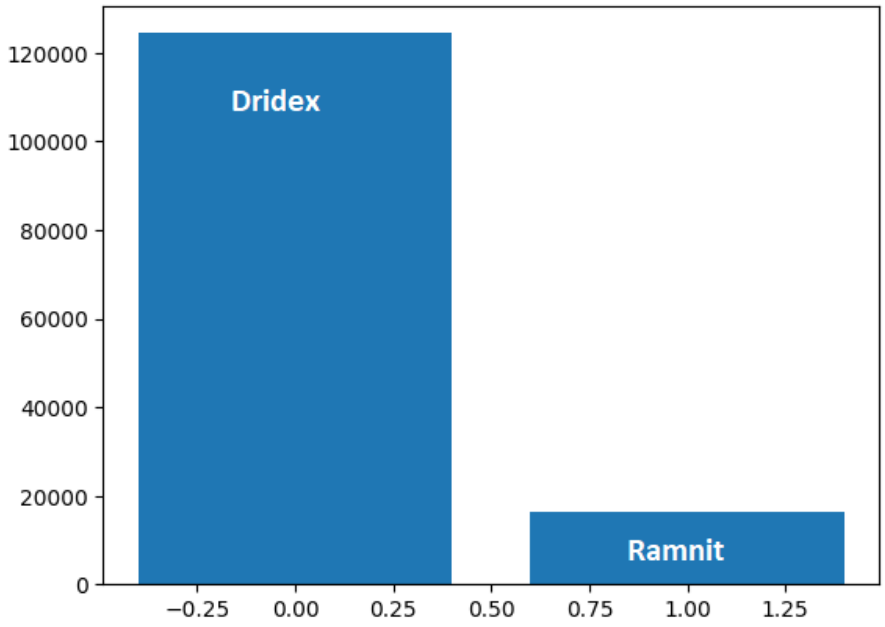

Figure 6.30. Distribution of classes in the train split

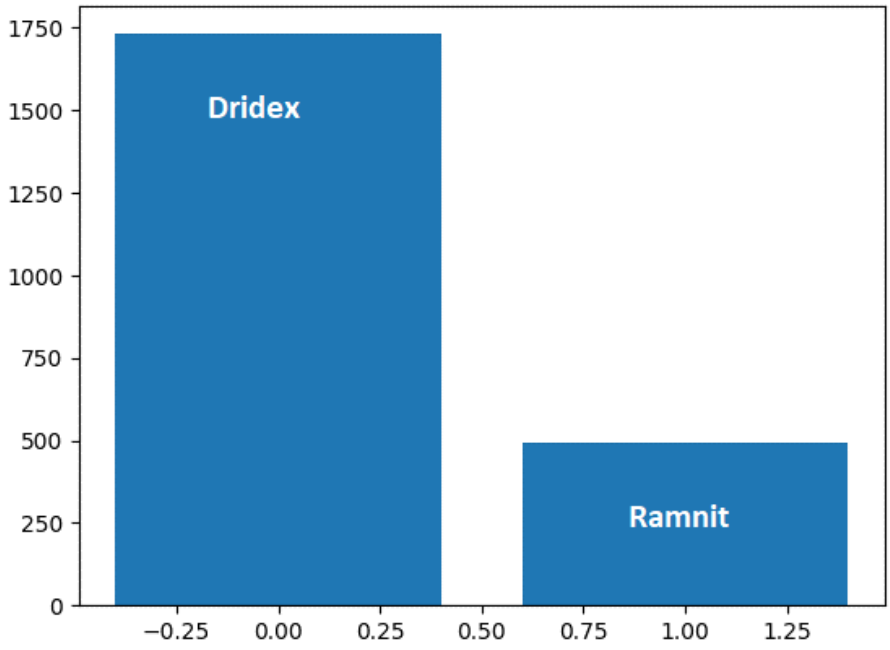

Figure 6.31. Distribution of classes in the test split

curve, suggesting a possible overfitting.

So, the last useful value for which to extrapolate the best metrics of the model turns out to be 1900 and the values are reported in the following table 6.30.

The model is particularly balanced also in the weights attributed to the features, they do not exceed 16% and overall, they are very uniform. This means that all are important in determining the class. In table 6.31 they are shown in the corresponding numerical format while in figure 6.33 they are more easily viewable thanks to the histogram.

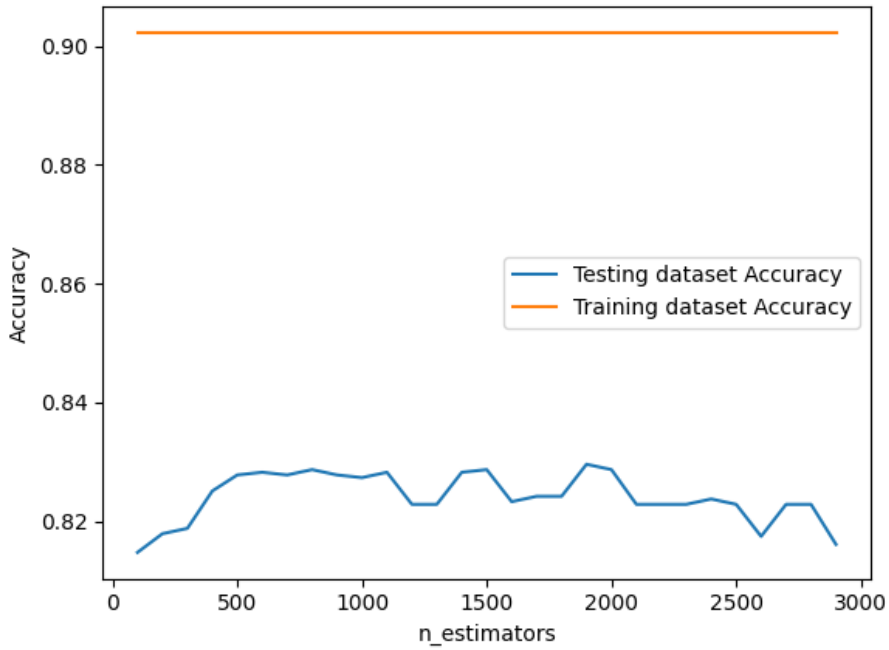

Figure 6.32. Random Forest: Accuracy per n_estimators 100 to 2900 (train and test)

	precision	recall	f1-score	support
dridex	0.85	0.92	0.89	1730
ramnit	0.62	0.44	0.51	494
macro avg	0.74	0.68	0.70	2224
weighted avg	0.80	0.82	0.80	2224
accuracy			0.82	2224

Table 6.30. Random Forest: metrics scores

resp_port	0.165695
orig_ip_bytes	0.164490
history_2	0.109854
orig_bytes	0.101802
resp_ip_bytes	0.088401
resp_bytes	0.076112
resp_pkts	0.060660
protocol	0.048767
orig_pkts	0.042580
service	0.041473
history_0	0.031878
conn_state	0.020500
history_3	0.015171
history_1	0.013605
history_5	0.010205
history_4	0.008805

Table 6.31. Random Forest: feature scores

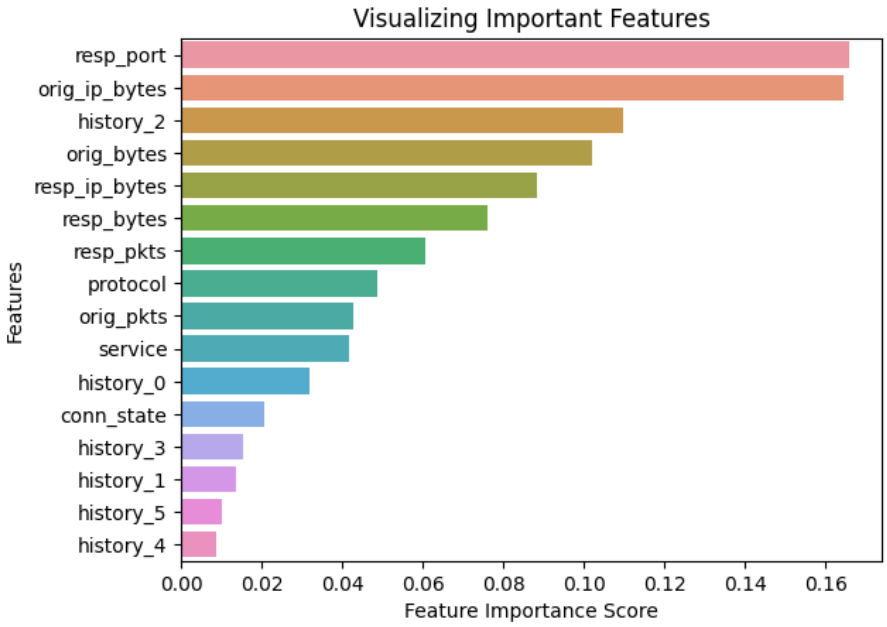

Figure 6.33. Random forest: feature importance

6.6.2 Gradient Boosting

Testing with the Gradient Boosting model produced slightly higher metric values than those produced with Random Forest. Differently from what one might imagine from the values of the metrics; the model is not very complex compared to the hyperparameters which have turned out to be better. The best hyperparameters obtained produced a train score of 0.8914676015208097 and a test score of 0.8471223021582733 and were the following:

- learning_rate= 0.1
- n_estimators = 1400
- max_depth = 3

It can immediately be seen that both the learning rate and the maximum number of depths are extremely low.

Instead, the metric values show a slight improvement in both the classification of the Dridex family and the Ramnit family, especially for the recall values of the latter. The values are shown in the table 6.32. As could have been previously understood by seeing the values of the hyper-

	precision	recall	f1-score	support
dridex	0.87	0.93	0.90	1730
ramnit	0.67	0.51	0.58	494
macro avg	0.77	0.72	0.74	2224
weighted avg	0.83	0.84	0.83	2224
accuracy			0.84	2224

Table 6.32. Gradient Boosting: metrics scores

parameters, especially the maximum depth, the weights associated with the features are strongly unbalanced. The model tends to mainly use history values and response port to classify the two malware families. They are represented in the graph in the figure 6.34 and in the table 6.33.

history_2	0.320453
resp_port	0.243389
resp_bytes	0.079560
orig_bytes	0.077164
orig_ip_bytes	0.076155
resp_ip_bytes	0.075463
resp_pkts	0.057931
history_1	0.023191
orig_pkts	0.020551
history_0	0.008362
service	0.007129
protocol	0.005593
conn_state	0.003215
history_3	0.001343
history_5	0.000404
history_4	0.000097

Table 6.33. Gradient Boosting: feature scores

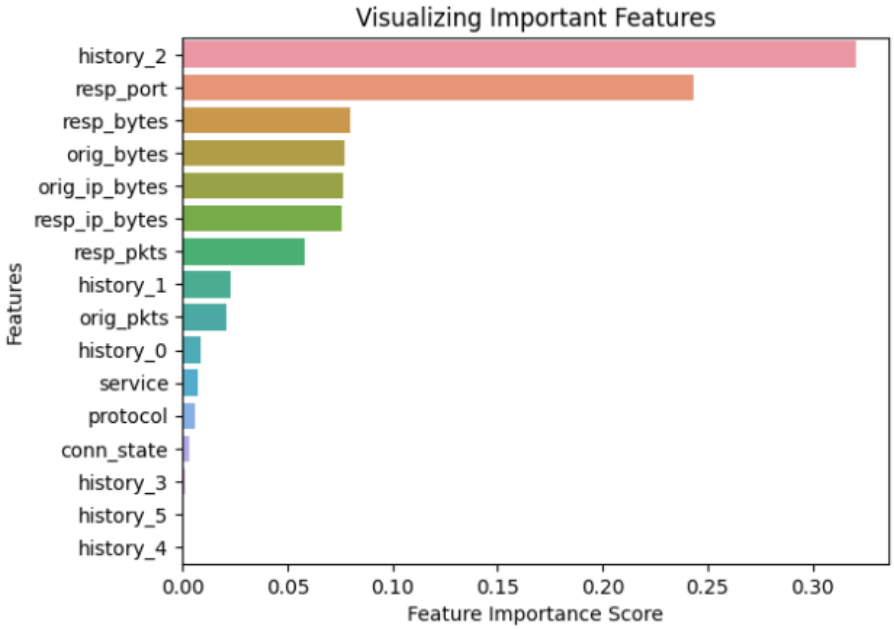

Figure 6.34. Gradient boosting: feature importance

6.6.3 K-Nearest Neighbours

Also, in this case, the range of values that can be assumed by k is increased to 40, taking the example of the previous test.
The model produced an excellent learning curve, with a rising edge followed by a downward trend both jagged. This curve indicates gradual learning to then begin to run into the phenomenon of overfitting. The k values equal to 16 and 17 were the ones that recorded the best accuracy values equal to 0.82958633.
The figure 6.35 shows the accuracy values as k increases. Also, the values of the metrics assumed for k equal to 16 and 17 are the same and for this reason, they have been reported only once in the following table 6.34.

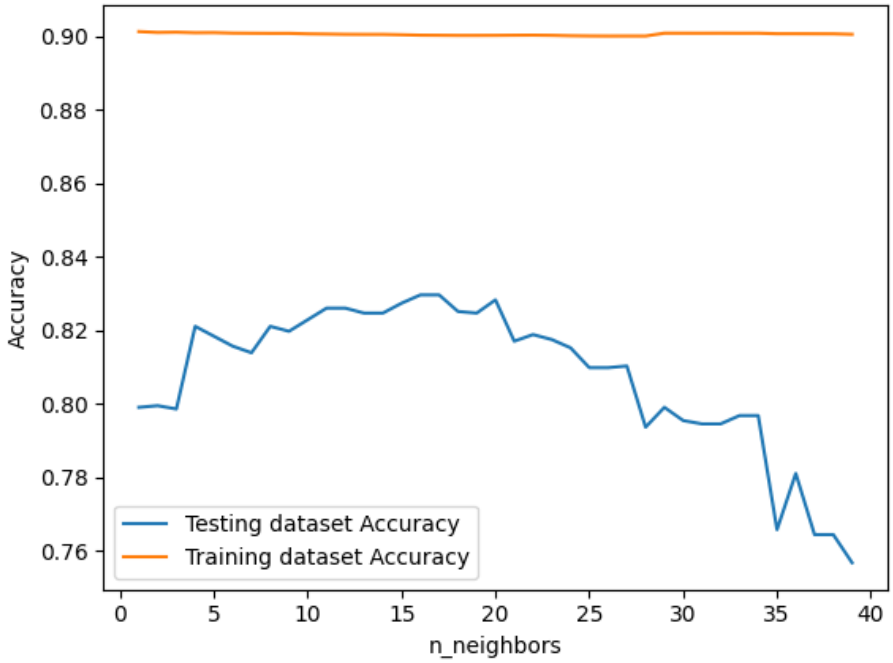

Figure 6.35. KNN: Accuracy per k-value 1 to 40 (train and test)

	precision	recall	f1-score	support
dridex	0.87	0.92	0.89	1730
ramnit	0.65	0.51	0.57	494
macro avg	0.76	0.71	0.73	2224
weighted avg	0.82	0.83	0.82	2224
accuracy			0.83	2224

Table 6.34. K-Nearest Neighbours: metrics scores with k = 16 and 17

6.7 Results with Repeated Stratified K-Fold Cross Validation

6.7.1 Introduction

In this section, I will analyze the most significant of the tests performed with the Repeated Stratified K Fold cross-validator on all three classes of malware families.

In particular, many tests of this type have been carried out, initially, they were carried out in a way that produced unique metrics and were not divided by the three types of classes but including all possible metrics both for macro and for weighted but, after careful consideration, it is considered more important to reduce the study metrics but at the same time explore them on all three classes.

The tests were conducted first on all three families of malware and then, in order to have an additional method of comparison, also on just the families of Trickbot and Dridex.

Since it is difficult to define how many parts to subdivide the dataset into, tests were carried out with a variable number of subdivisions, first in sequence and then, once the right stability and independence were achieved, in parallel to reduce execution times which turned out to be very long (from 2 to 24 hours). Being very long and branched tests, it was considered appropriate to summarize as much as possible and to deal with only the most significant in the content of this

thesis. For all the tests carried out, the values from 30 to 70 with an increase of 10 were used as the number of splits, in some cases tests were also carried out on different values such as 5, 10, and 75 but the results obtained from them were in line with the previous ones. The choice of this range is due to the online reproduction of the quantity and distribution of the packages on the number of pcaps, as they appear to be a total of 79 but for 5 of them (derived from stratosphere IPS) their size is much higher.

The choice not to use the Gradient Boosting model for this type of test is dictated by the lack of effectiveness found in the previous tests compared to the amount of time necessary for its configuration, in fact, in this paragraph only the tests conducted with the Random Forest and KNN models will be reported as they turned out to be the most efficient.

6.7.2 Random Forest

The tests conducted on the Random Forest model were produced with a number of estimators equal to 1000 on the basis of the results produced by the previous tests. This value turned out to be a good intermediate value as it was neither too low to have a good level of detail nor too high with the risk of going overfitting.
Before this value, tests were also carried out which had lower results but also considerably reduced execution times, about 15-25% more performance for accuracy and precision at the cost of almost double the time.

The following table 6.35 shows the average values of the metrics produced through the cross-validation with the Repeated Stratified K-Fold method divided by the number of splits, each of which has a repetition equal to 3.

Splits	Accuracy	Precision	Recall	F1-score
30	0.6588938877	0.78881351782	0.42917367359	0.42805994184
40	0.65889693926	0.78893467469	0.42916154205	0.427995380838
50	0.65889999270	0.78938459931	0.429205471297	0.4280652851
60	0.6589019861	0.78948411845	0.429191230768	0.42801547243
70	0.65890200564	0.78870815602	0.42918598243	0.427975557821

Table 6.35. Random Forest: Repeated Stratified K-Fold Cross Validation mean metrics for split

6.7.3 K-Nearest Neighbours

The tests conducted on the KNN model were produced with the value of k equal to 10 as from the previous tests it turned out to be an intermediate value so it is not too small to have more distances to compare nor too large incurring error problems in the distinction between classes.

Unlike the tests with the Random Forest model, with the KNN model, the execution times are significantly lower (enter values) but the metric values are affected. In the table 6.36 it can be seen how the values of the metrics are very different from those produced by its counterpart.

6.7.4 Comments

In this paragraph, the use of the Repeated Stratified K-Fold Cross Validation has been discussed as an alternative to the data mining phase as it is too time-consuming.

From the first tests carried out with the KNN model, low results were obtained compared to both the last tests with the help of data mining and the first ones.

Splits	Accuracy	Precision	Recall	F1-score
30	0.5082078972	0.79878330669	0.42807574889	0.360580910108
40	0.50820182336	0.79872637866	0.428080483086	0.36055465907
50	0.5081957078	0.7994258277	0.42808307432	0.360552482712
60	0.50820080235	0.79941445206	0.4281066201	0.36057180123
70	0.50820283288	0.79914806923	0.428115518232	0.36056020113

Table 6.36. K-Nearest Neighbours: Repeated Stratified K-Fold Cross Validation mean metrics for split

This observation suggested how important the data mining phase was as a first step before classification but subsequently, with the first tests conducted on the Random Forest model, it was noted that the results of the metrics were much more similar to the first ones obtained with the data mining phase, in fact, the obtained F1-score values are very similar.
In the previous sections, the average values obtained from the metrics were analyzed, but to have a more detailed view of how the model behaves, it is also necessary to pay attention to the single values with respect to the classes relating to the malware families.
The following tables (Table 6.37, Table 6.38, Table 6.39, Table 6.40) list the two best examples obtained from the two models, for each metric three values are listed that correspond to the three classes.

Model	Split	Accuracy
KNN	30	0.50944
RF	50	0.66631

Table 6.37. Best accuracy produced by KNN and Random Forest models.

Model	Split	Precision		
KNN	30	0.43263	0.97222	0.99453
RF	50	0.65015	0.96153	0.75287

Table 6.38. Best precision produced by KNN and Random Forest models with values divided by class.

Model	Split	Recall		
KNN	30	0.99951	0.07099	0.22845
RF	50	0.95107	0.08474	0.29287

Table 6.39. Best recall produced by KNN and Random Forest models with values divided by class.

Model	Split	F1-score		
KNN	30	0.60388	0.13232	0.37155
RF	50	0.77233	0.15576	0.42170

Table 6.40. Best F1-score produced by KNN and Random Forest models with values divided by class.

For the KNN model, the number of splits that recorded the best values is 30 and by consulting the values of each class (in order Trickbot, Dridex, and Ramnit) it can be seen how for the first class the results are relatively low but compared to the two others are undoubtedly better.
For the Dridex class, as well as in the first tests with data mining, it is noted that the model has a high precision but a very low recall which leads to a very low classification of this malware family.
Finally, the Ramnit class follows the trend of the Dridex class with very high precision and recall, which, although higher than the Dridex family, is still too low to obtain a good final classification value.

Instead, for the Random Forest model, improvements can be noted in the results of the various metrics on all classes, they are more uniform in terms of precision while for recall they remain almost similar, bringing the values of the F1-score per class from 2 to 17 %.

Even though the average values of the metrics seemed lower, or very similar in the case of the Random Forest model, comparing them to the first results obtained with data mining; and analyzing the values of the individual classes we notice how the results are completely different. In fact, it can easily be seen that the models obtain the same trends found by the latest data mining tests even if with lower values. In particular, the Trickbot class is the one with the best classification, the Dridex class always has high precision and very low recall and finally, the Ramnit class obtains overall low but consistent values. All these behaviors are the same as those found during the various tests previously analyzed, therefore the models obtained with these tests are able to learn these important aspects even if the final values are lower.

In summary, the tests carried out with the use of the cross-validator Repeated Stratified K-Fold have produced very interesting results, in the first part of the tests by analyzing only the results of the metrics it was understood that these were clearly lower than the results produced with the phase of data mining, but subsequently analyzing the learning of the model concerning the different classes it was noted that it had reported the same behaviors found in the previous tests, although with lower final values. Lastly, the use of this technique has proved to be a good counterpart to the use of the data mining phase which on the one hand slightly lacks the final results but on the other hand, greatly decreases the times.

6.8 Final Results

6.8.1 Introduction

During the tests carried out in paragraph 6.6 it was noted how the model changed learning by improving the recognition of the Dridex class during its evolution over the different years, this refuted all the hypotheses that had previously been made from the test results.
Being a single test compared to many, this could seem like a mere fortuitous event in which the model had learned just as we wanted it to, which is why other tests of this type were followed to see if similar results were obtained again. In fact, the tests have produced very similar results confirming before, so I started a phase of analysis of the dataset and of the previous tests in order to understand why there was this great disparity by subdividing the dataset.
The only explanation that I have given is due to the dataset, re-analysing the dataset I noticed that a problem could be given by the different types of pcap between those treated by the two sources (Stratosphere IPS and malware-traffic-analysis) and by their different concentration of flow per pcap.
In fact, Stratosphere IPS produces significantly larger pcaps than malware-traffic-analysis, this imbalance between the two would lead the models to give more importance to the data obtained from the first source by specializing the algorithm in recognition and classification based mainly on those captures, and being that Stratosphere IPS does not have captures of the same malware family that we deal with in this thesis during their evolution over the years, this leads the algorithm to specialize in that malware version making it less sensitive to learning from its subsequent versions.
Observing precisely the individual results of the Trickbot and Dridex classes, it can be seen that the models are able to deal better with the first than the second, this is dictated by the evolution of the two example cases of malware families, Trickbot has received fewer substantial evolutions compared to Dridex which has continued to evolve and mask its malicious attachment content.
From these considerations, new tests were carried out in order to confirm or refute these hypotheses.

6.8.2 Trickbot and Dridex without Stratosphere IPS pcaps

The first tests carried out focused only on the Trickbot and Dridex families as it was precisely on these that the behavior of the algorithm could easily be seen and furthermore, since the dataset was reduced, more tests could be carried out in less time.
These tests have been set up in order to resize the dataset excluding the contributions of Stratosphere IPS and considering only those of malware-traffic-analysis, it is not a good idea to decrease the size of the dataset as in doing so we decrease model learning but in this example case it was important to understand how it reacted and if indeed, given the difference in the types of pcap, this could lead the model into error.
Furthermore, in order to further reduce the times it was decided to use the Repeated Stratified K-Fold Cross Validation which, as we learned previously, manages to reduce the times due to data mining to the detriment of slightly lower performance without however changing the behavior of the model.
The Repeated Stratified K-Fold Cross Validator was set up with 10 splits and 3 repetitions and with the two models that I also used during the previous paragraph with several estimators for Random Forest equal to 1000 and values of k for KNN equal to 10, 21, and 30.
Contrary to what would normally be expected in machine learning from the decrease in the dataset, the model recorded significantly higher metric values, managing the evolution of malware over the years even in the case of the Trickbot and Dridex families.
The Random Forest model was the best performing obtaining the following average metric values:

- **Mean Accuracy**: 0.883869321082185
- **Mean Precision**: 0.864191265201137
- **Mean Recall**: 0.8211042493959468
- **Mean F1-score**: 0.839009048152071

Furthermore, tests were conducted on the values of the metrics both in the train and test phases, the recorded values of which are as follows:

- **Average train accuracy**: 0.9112302194997448
- **Average test accuracy**: 0.8838693210821847
- **Average train f1 score**: 0.9088246239581391
- **Average test f1 score**: 0.8802846821414719
- **Average train macro f1 score**: 0.8778881549891456
- **Average test macro f1 score**: 0.839199109652306

As can be seen, the model appears to have much higher metrics than the previous tests and this is also reflected in the individual classes that will be analyzed at the end of this section.

Instead, for the model developed with KNN the results were slightly worse, but still comparable with the best ones produced by the previous phases. The hyperparameter that yielded the best results out of the three was the value 10 for K which averaged the following metric values:

- **Mean Accuracy**: 0.7501020929045431
- **Mean Precision**: 0.7169129021706336
- **Mean Recall**: 0.775857327235529
- **Mean F1-score**: 0.7210024336079129

Model	Hyp	Accuracy	Precision		Recall		F1 score	
RF	1000	0.90045	0.85763	0.91257	0.73511	0.95773	0.79166	0.93460
KNN	10	0.79479	0.57718	0.90803	0.76557	0.80495	0.65816	0.85339
KNN	21	0.78407	0.55489	0.92670	0.82492	0.76986	0.66348	0.84103
KNN	30	0.77794	0.54893	0.90669	0.76785	0.78144	0.64019	0.83942

Table 6.41. Best results produced by KNN and Random Forest models with values divided by class.

Finally, the table 6.41 shows the values of the metrics of the best tests for each model of each class. The order of class is as first Trickbot and second Dridex.

As previously mentioned, the best values of the metrics produced are from the Random Forest model and it can be seen that the issues relating to recall have considerably improved as a result, also improving the values of the F1-score.
For the sake of completeness, the table also shows the best results for all three tests carried out with KNN with different K values, but as can be seen, the difference is small but it can be seen that as K increases, performance decreases.

6.8.3 Trickbot, Dridex, and Ramnit without Stratosphere IPS pcaps

In this section the last test carried out will be analyzed, it was conducted on all three classes of malware excluding pcaps with Stratosphere IPS origin with the use of Repeated Stratified K-Fold Cross Validation dividing the dataset into 10 splits and with 3 reps.
Given the high results from the previous test, it was decided to resume testing with the Gradient Boosting model, following the hyperparameters of the Random Forest model. Also, in this case, the models used are Random Forest, Gradient Boosting, and KNN respectively with the number of estimators equal to 1700 and a value of K equal to 21. The decision to consider only these two hyperparameters is dictated by previous experiences, I was inspired by the models of the last experimentation by increasing the values as the dataset has also increased in size and class, taking an intermediate value to the already extensively tested ranges.
As anticipated, the model obtained with KNN obtained lower metric values than those recorded by Random Forest, while slightly higher results were obtained than the previous test with the Dridex and Trickbot classes. The average values of the metrics obtained are listed below:

- **Mean Accuracy**: 0.7542426814484219
- **Mean Precision**: 0.7316494138408959
- **Mean Recall**: 0.780693973365937
- **Mean F1-score**: 0.7466474173558512

The model obtained with Random Forest reaffirms itself as the most performing, with an average performance difference on all metrics per class of about 11.4%. On average, the model recorded metric values equal to:

- **Mean Accuracy**: 0.8853645926639684
- **Mean Precision**: 0.8829514075872171
- **Mean Recall**: 0.8532535987389908
- **Mean F1-score**: 0.8647551482138447

Instead, the Gradient Boosting model obtained performance similar to Random Forest, which proved to be a stronger contender against the Random Forest model compared to KNN. On average, the metric values are the following:

- **Mean Accuracy**: 0.84162116940288
- **Mean Precision**: 0.8467102698053989
- **Mean Recall**: 0.7772265521145941
- **Mean F1-score**: 0.7947646673008734

Furthermore, further investigations on the values of the metrics of Random Forest have been produced here as well, distinguishing them in the two phases of train and test, placing greater emphasis on the macro-ones, and are as follows:

- **Average train accuracy**: 0.9112302194997448
- **Average test accuracy**: 0.8838693210821847
- **Average train f1 score**: 0.9088246239581391
- **Average test f1 score**: 0.8802846821414719
- **Average train macro precision**: 0.9142280675373732
- **Average test macro precision**: 0.8838376660009145
- **Average train macro recall**: 0.8879883678632484
- **Average test macro recall**: 0.8528917353444776
- **Average train macro f1 score**: 0.8778881549891456
- **Average test macro f1 score**: 0.839199109652306

In addition to the average values obtained from the various metrics for train and test, graphs were also produced on their progress during the two phases and were divided by metric: accuracy (Figure 6.36), precision (Figure 6.37), recall (Figure 6.38), F1-score weighted (Figure 6.39) and F1-score macro (Figure 6.40).

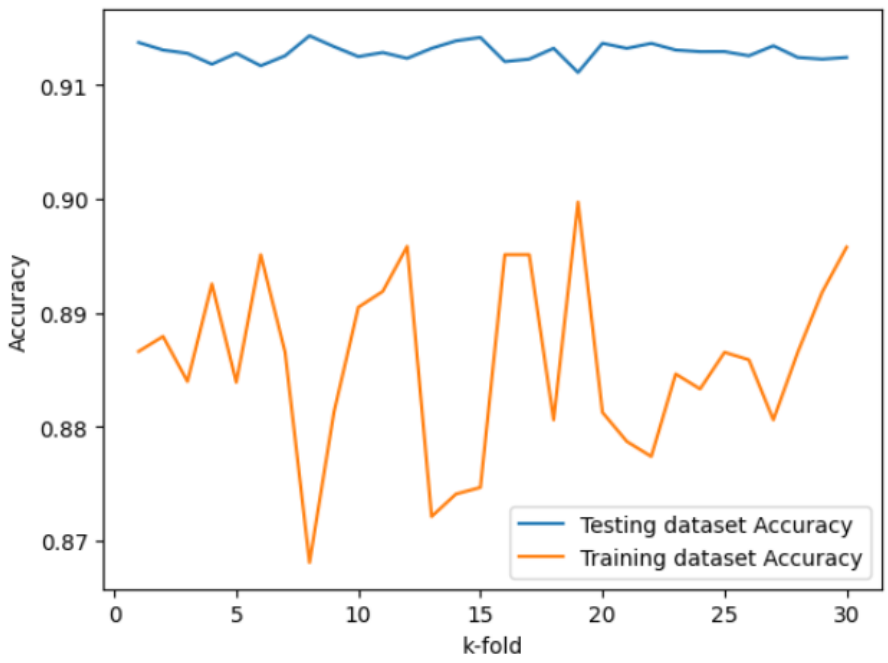

Figure 6.36. Random Forest: Accuracy train and test per k-fold

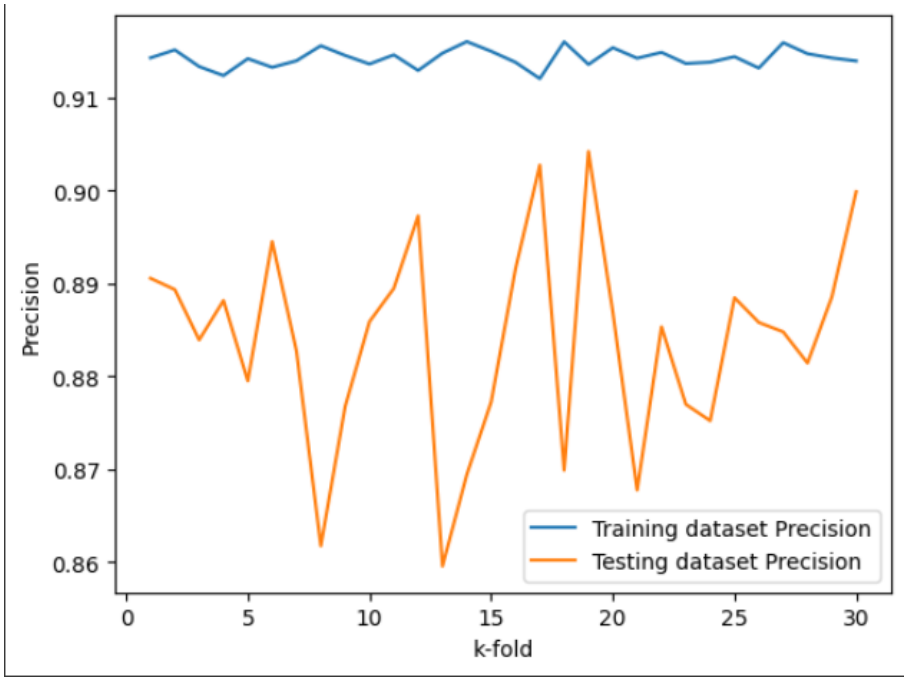

Figure 6.37. Random Forest: Precision train and test per k-fold

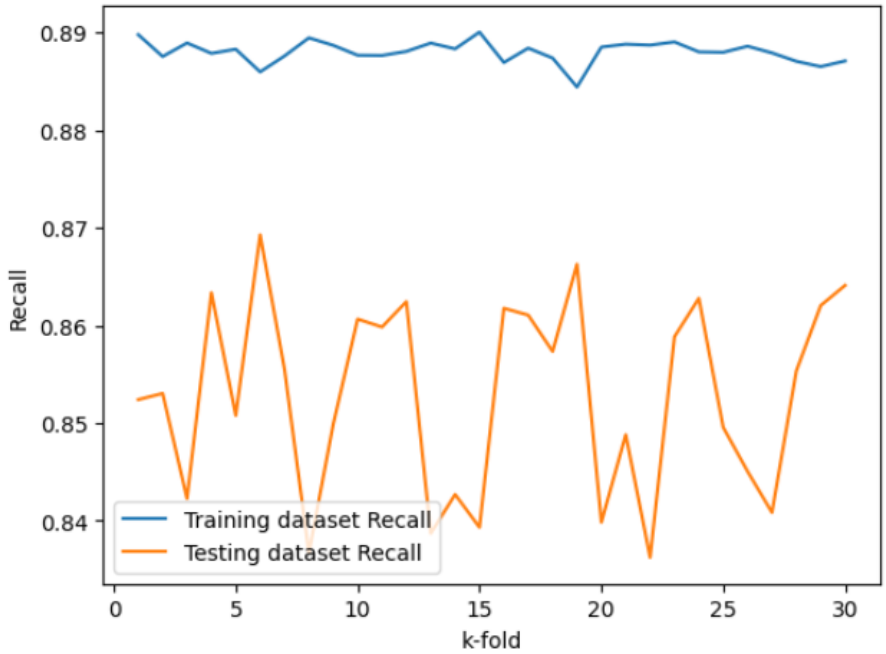

Figure 6.38. Random Forest: Recall train and test per k-fold

Furthermore, the following tables (Table 6.42, Table 6.43, Table 6.44, Table 6.45) shows the best values recorded by the two models with the subdivision by class. The classes are listed in the following order: Trickbot, Dridex, and Ramnit.
Above all in this case, the performances are clearly higher than in the previous tests using the three classes, instead of comparing it with the previous test with only two classes the performances remain almost the same, exclusive of a slight decrease in some values, thus obtaining an excellent result.

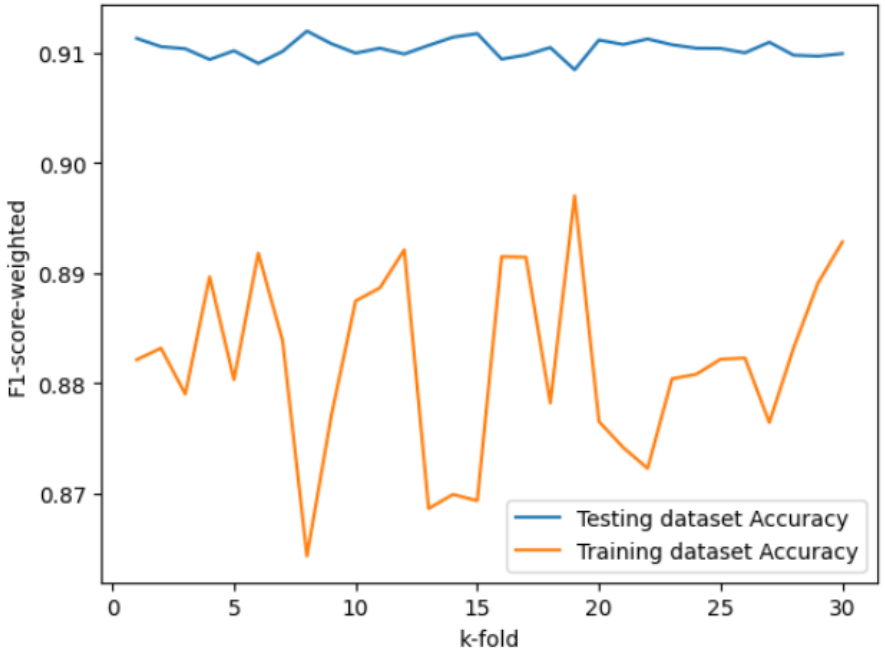

Figure 6.39. Random Forest: F1-score weighted train and test per k-fold

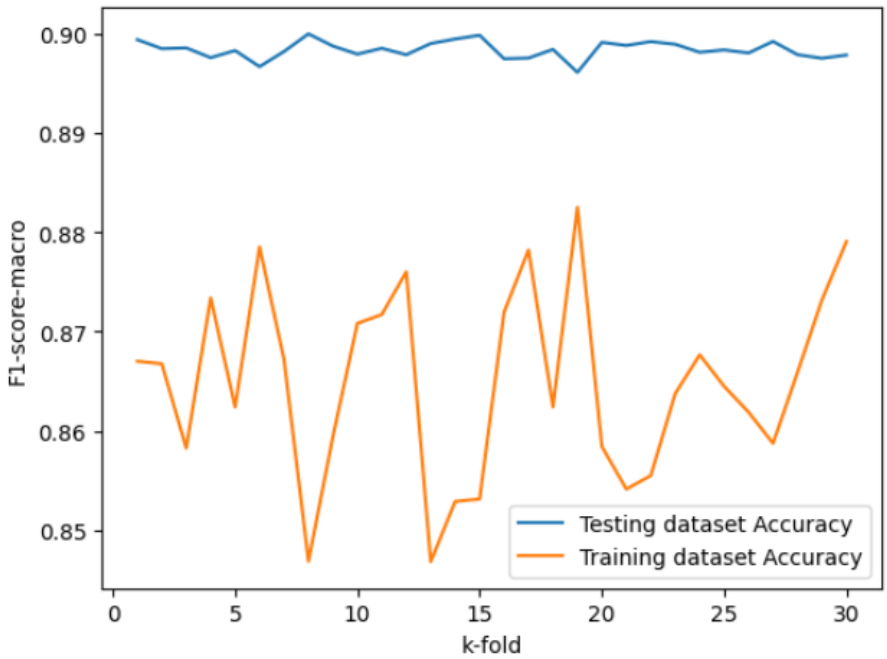

Figure 6.40. Random Forest: F1-score macro train and test per k-fold

6.8.4 Comments

In this last paragraph, the latest tests conducted were analyzed, the purpose of which was to find out if the problems relating to the classification of the two malware families (Dridex and Trickbot) could be attributed to the heterogeneity of the data present in the dataset, being this

Model	Hyperparameter	Accuracy
RF	1700	0.9003957783641161
GB	1700	0.8562953197099539
KNN	21	0.7725774555042848

Table 6.42. Best accuracy produced by KNN, Gradient Boosting, and Random Forest models.

Model	Hyp.	Precision		
RF	1700	0.8705036	0.92822967	0.90281827
GB	1700	0.8504672	0.9009434	0.8487626
KNN	21	0.53829787	0.81938326	0.89390244

Table 6.43. Best precision produced by KNN, Gradient Boosting, and Random Forest models with values divided by class.

Model	Hyp.	Recall		
RF	1700	0.7202381	0.92380952	0.95773196
GB	1700	0.5400593	0.90952381	0.95463918
KNN	21	0.75074184	0.88151659	0.75644995

Table 6.44. Best recall produced by KNN, Gradient Boosting, and Random Forest models with values divided by class.

Model	Hyp.	F1 score		
RF	1700	0.78827362	0.92601432	0.92946473
GB	1700	0.66061706	0.90521327	0.89859292
KNN	21	0.62701363	0.84931507	0.81945221

Table 6.45. Best F1-score produced by KNN, Gradient Boosting, and Random Forest models with values divided by class.

last generated by the merger of two distinct datasets produced by two different sources.

From the analysis of the values of the metrics produced by the two tests in question and from the analysis of the models, we can say for sure that the data present in the previous dataset with Stratosphere IPS source actually created difficulties for the models in classifying the Dridex and Ramnit malware families, in since by excluding them, clearly superior results are obtained.
As anticipated just before, the best hypothesis is that since the previous dataset mainly consisted of samples with Stratosphere IPS origin and since they did not trace the evolution of the malware families in question, the models produced recorded great difficulties in managing a possible evolution of behaviors of the Dridex and Ramnit malware families.

Finally, having to highlight which model is more capable of managing this type of task, the choice falls on Random Forest which was the model that turned out to be more performing between the three. The Random Forest model during the last tests recorded higher fit times than KNN (0.75 compared to 36.5) but this is mainly dictated by the size of the dataset, the KNN model increases the fit times as the samples grow while the Random Forest is less prone to these kinds of performance issues.

On the other hand, the Gradient Boosting model achieved very good metrics, securing second place in the rankings among these three models with metrics closely aligned with those obtained by the Random Forest model, albeit slightly lower.

Chapter 7

Conclusions

The objective of this thesis was to create a tool that was able to classify malware families using machine learning techniques. In particular, the tool had to be able to determine the family of malware only through network traffic, which moreover could not be inspected in the contents due to encryption and avoid dealing with sensitive information. In order to better manage the classification, the model had to obtain high precision and recall values, to have respectively fewer false positives and undetected malware; these metrics combine together in the F1-score which has become the most important metric for evaluating the proposed model.

Various phases were carried out for its realization, the first were data pre-processing and data mining which reduced the complexity of the data and reconstructed the flows and, lastly, highlighted which were the most significant in terms of correlation. With the results of the previously discussed phases, several solutions of dataset splits were developed which were fed to three different machine learning models, which are Random Forest, Gradient Boosting and KNN. This last phase of model training and testing underwent many rounds of refinement by changing dataset arrangements, tuning hyperparameters and using cross-validators.

In the results chapter, the metrics obtained were reported and analysed, as well as how they evolved during the experimentation phase. From the first tests, the models had found difficulty in managing two families of malware, which was also possible to notice during the data mining phase as, again for these two families, results had not been obtained as well as for the other family. Through a specific analysis of these two malware families, in particular the Dridex family, it was discovered that they have had a much more marked continuous evolution over the years than the other family. In the last phase of experimentation, with the technical skills developed during the previous phases and with the new knowledge of the behaviours of these malware families, excellent results were obtained on all the metrics for managing the task for which it was designed. The models that obtained the best metrics among all the tests performed during the experimentation phase were Random Forest and Gradient Boosting. On the other hand, KNN was more useful when data mining results were used and less with the use of cross validators, while Random Forest always had good results but especially in the last phase it proved to be the best. Unlike the other two models, the Gradient Boosting model exhibited a more oscillatory performance. In some cases, it was slightly better than the Random Forest, while in others, it achieved very low results. However, on average, its behaviour proved to be very similar to the Random Forest model, just as in the last test, earning it second place among the three models.

The proposed model, with respect to the traditional malware analysis techniques and the related works analysed in the early stages of the study on the state of the art, lays the foundations for the construction of instrumentation that can help malware analysts or even replace the current methods for the detection and classification of malware over long periods handling their evolution. The main advantage lies in avoiding the time and costs of static or dynamic analysis of the malware by an analyst and the comparison with its previous and subsequent versions. On the other hand, however, it must be said that it cannot help to discover particular behaviours of malware as it does not produce any evidence on the reason why it has been labelled as belonging

or not to that class, especially in the case of an algorithm KNN extension.

7.1 Future Works

The solution presented in this thesis is able to classify malware by inspecting network flows, almost allowing an on-the-wire classification without violating privacy terms. This solution could potentially be used in the future in the field of information security applications. The latest configuration introduced with the use of the Random Forest algorithm and the use of the Repeated Stratified K-Fold cross-validator obtained excellent results, especially with a resized dataset.

Various limitations were noted during the testing phase. The first of these was due to data mining, a very slow and expensive solution which led to little improvement in tracking the evolution of malware. This phase could be very important to improve the performance of the model, but the use of a powerful system dedicated solely to that purpose should certainly be taken into consideration. Furthermore, it has been found that the metrics used during the data mining phases badly manage the evolution of the various classes, in future developments, it would be useful to try to better distribute the weights of the equation by following the important features extracted from the model or even think to another equation or a different approach.

A further problem that was encountered during the testing phase is certainly the management of particular malware which, during its evolution, has used very complex evasion techniques, totally changing its behaviour. An example case was that of the Dridex family which has undergone a profound evolution over the years, in its first versions it was based on a malicious link sent in phishing emails, but subsequently evolved to counter anti-malware solutions sending malicious word documents that implemented the infection via script, then moving on to encrypted documents in which the user had to manually enter the password, and finally changing the format of attached documents by disguising the content with the .pdf extension. With the necessary precautions, the model was able to manage these cases, but it would be useful to verify if with other types of malware, as those present in the dataset are all Trojans aimed at the banking sector, and families are able to promptly manage any evolutions.

Another problem is dictated by the dataset, as widely discussed the datasets that are based solely on network captures and that are publicly available are few, above all it is difficult to find different datasets with as many different sources. A future work could certainly be the integration of different datasets built manually or, even better, with the cooperation of different institutions or universities.

Finally, it would be very useful to be able to compare the current results with a more complex type of machine learning such as ANNs. In particular, LSTM, Long-Short Term Memory, has been developed in recent years as an evolution of RNN networks whose advantage is learning over long but also short time sequences, hence the name, and storing them in memory. These networks are an evolution of RNN networks as they solve the vanishing gradient problem that led RNN networks to have difficulties in capturing long-term dependencies. This has been overcome with a different architecture based on three gates: input, output and forget gates. These last perform a very important function, namely the management of information that must be dropped going through the network. In this way, the neural network manages both long-term and short-term memory and therefore we can pass the information to the network and retrieve it in much later stages to identify the context of the prediction. This article [47] explains in more detail the impact of gradient vanishing and how LSTM works as well as the presence of its tutorial.

Appendix A

User Manual

The project is mainly based on the Google Colab Python block notes structure, there is also a Python script that was used during the data mining phase using HPC and Google Colab resources at the same time.
For these reasons, the code documentation is mainly in .ipynb format, except \Matrix_for_Datamining.py. Each file is accompanied by its internal documentation, which explains the inputs and outputs, the purpose of each module, and the procedure used. It's important to highlight that when it comes to .ipynb files, as it's possible to manage the installation of necessary dependencies within the files, there's no need for any specific actions to ensure their resolution. You simply need to follow the order of code blocks within the module. However, this does not apply to .py files used in the HPC infrastructure, which require the installation of the following libraries: pandas, numpy, and math.
To install all dependencies, use the command **pip install <library >**. All libraries have their own official page with documentation included; should you need more libraries, they will automatically be downloaded using the command described above. Packages are also available on conda.
Since the project follows a specific flow it is necessary to execute them in the following order:

- \Matrix_for_Datamining.py
- \Datamining_results_calculation_based_on_nflow.ipynb
- \Machine_learning_no_cross_validation.ipynb
- \Machine_learning_cross_validation.ipynb

For each module, the inputs and outputs are defined, and the former must be correctly respected for correct functioning.
To run the program in an HPC-like environment, the following command must be executed from the project folder:

python Matrix_for_Datamining.py

Instead, for the Google Colab environment it is necessary to open it and execute one by one each code section. Remember to upload the necessary input files, it is recommended to use the one present in the repository correlated to this thesis (https://www.dropbox.com/scl/fo/a9ok5tsmw\l9kagd7g3i4c/h?rlkey=bin9dtgi7npukp99jqr2eidq5&dl=0).

1. For \Matrix_for_Datamining.py, the output files of the Zeek framework must be defined at each run, and the data frames necessary for each file divided by malware family and their respective dictionary with the file names must be defined. It is recommended to use the files located in the "code, results, and test documentation" section within the "pcap to csv" folder instead of using Zeek framework to regenerate the .txt file and after converting them into .csv format. There are three subfolders with their respective files for each family. You

should load them one family at a time and specify the filename to be used within the code. By way of illustration, the list of files used for the Trickbot family has been included in the code.
As output a series of testing-.csv files shall be created, and they are the correlation matrix of each pair of input files, as the name suggests.

2. For \Datamining_results_calculation_based_on_nflow.ipynb the files must be loaded in the right folder and the number of packages per flow must be defined, it is important to execute them separately for each family.
The output of this file is a dictionary of the most significant files of each family that can be used for the next steps in order to improve machine learning results.

3. For \Machine_learning_no_cross_validation.ipynb it is needed to load the .csv files necessary for training separately in the train and test folders and execute the various blocks of code following the defined procedure and modify the values of the hyperparameters of the various algorithm models based on the type of experimentation.
The possible hyperparameters are n_estimators, learning_rate, max_features, max_depth, and n_neighbors.
The output of this file is the statistics of the evaluated models and their related plots.

4. Finally, for \Machine_learning_no_cross_validation.ipynb it is needed to load the dataset files, follow the procedure, and possibly modify the values relating to the cross-validator and the hyperparameters of the models to carry out tuning on them.
In addition to the previous file, there are the following hyperparameters for the cross-validator: n_splits and n_repeats.
The output of this file is the statistics of the evaluated models and their related plots.

Appendix B

Programmer Manual

The project is based on four code files, each one of which is oriented to a specific phase of the proposed model. Since they follow a precise flow, in order to replicate the same structure, it is necessary to execute them in the following order:

- \Matrix_for_Datamining.py
- \Datamining_results_calculation_based_on_nflow.ipynb
- \Machine_learning_no_cross_validation.ipynb
- \Machine_learning_cross_validation.ipynb

Each file has its own documentation inside of it and as many comments, so that they can be used for future work. Below are the functions divided by file and their description.

B.1 \Matrix_for_Datamining.py

This module aims to create useful correlation tables between the pcap files in the dataset belonging to the same family, which have already been processed by the Zeek framework and appropriately converted into CSV format.

Input:

- Output from the Zeek framework converted into CSV,
- List of files to correlate.

Output:

- Correlation tables.

To ensure proper functioning, data frames containing the files to be correlated must be defined, one family at a time, along with their respective dictionary compiled with their names. This dictionary will be used to automate the writing of the output file as a correlation of the two input files. Once defined, the distances between each packet in file f1 and each packet in file f2 are calculated, generating a matrix of size equal to the dimensions of files f1 and f2. The following distances are used for this calculation:

- Levenshtein Distance,
- Cosine similarity,
- Binary distance,
- Interflow distance.

These four distances are used in the following formula, which determines each entry in the correlation table:

$S = 3\ B{+}2\ ld{+}6\ cosine{+}1\ interflow.$

Note: The third code block is a script for the automatic conversion of an output file from the Zeek framework into CSV format.

B.2 \Datamining_results_calculation_based_on_nflow.ipynb

This module is used after constructing correlation tables to restructure them with a specified number of flows defined in the first block of code.

Input:

- Correlation tables (.csv),
- File name dictionary.

Output:

- Occurrence dictionary.

By defining the value of "nflow," the tables are merged based on their value and consolidated into a single table for each malware family.

After the previous steps, correlations are sought by sorting and filtering the tables (in the example case, it was decided to lower the correlation threshold from 12, the maximum value, to 11 because the results obtained could not surpass the value of 11.5. The minimum value chosen is 6, which filters out many results with very low correlation).

After calculating the correlation, the most significant PCAPs are selected based on the average distance values and the previously calculated occurrence.

B.3 \Machine_learning_no_cross_validation.ipynb

This module aims to convert datasets into a format acceptable by machine learning models by encoding categorical data and then feeding them to Random Forest, Gradient Boosting, and K-Nearest Neighbors models.

Input:

- Training dataset,
- Test dataset.

Output:

- Metrics results obtained from Random Forest,
- Metrics results obtained from Gradient Boosting,
- Metrics results obtained from K-Nearest Neighbors.

The initial code blocks handle the concatenation of all the CSV files present in their respective train and test folders, creating the two train and test CSV files.

After concatenation and saving the two train and test files, they are processed to make them compatible with machine learning models. This involves converting non-numeric data into numeric data. In the example case, the following operations are performed:

- For the **"History"** parameter, Binary Encoding is applied, which transforms the categorical variable by calculating all the values it takes and converting it into a series of columns to handle all cases.
- For the parameters **"protocol, service, conn_state,"** Ordinal Encoding is applied, counting the possible values and encoding them in the starting column.
- For the parameters **"orig_bytes and resp_bytes,"** cases where "-" values are present as null values are handled and converted to "0."

After converting the categorical features, two train and test CSV files are generated, and class distributions are calculated concerning the entire train and test dataset.

Finally, machine learning models are trained and tested as follows:

- **Random Forest**: After defining the range of the hyperparameter **"n_estimators"** and the increment step, the model is trained and tested, creating accuracy curves for both train and test datasets. Based on these models, the value of "n_estimators" is selected for in-depth testing, generating metric results and feature weights.
- **Gradient Boosting**: An initial testing phase is conducted with default hyperparameter values, and the obtained metrics are extracted. Then, ranges for the hyperparameters **"learning rate," "n_estimators,"** and **"max_depth"** are defined, along with their increment steps. Subsequent tests are performed to search for the best hyperparameters within the defined ranges. After identifying them, the obtained metrics and feature weights are reported.
- **K-Nearest Neighbors**: After defining the range of the hyperparameter **"n_neighbors"** and the increment step, the model is trained and tested, creating accuracy curves for both train and test datasets. Based on these models, the value of "n_neighbors" is selected for in-depth testing, generating metric results.

Finally, if necessary, there are two code blocks to list the files that constitute the training and test datasets.

B.4 \Machine_learning_cross_validation.ipynb

This module aims to convert datasets into a format acceptable by machine learning models by encoding categorical data and then feeding them to Random Forest and K-Nearest Neighbors models.

Input:

- Training dataset,
- Test dataset.

Output:

- Metrics results obtained from Random Forest,
- Metrics results obtained from K-Nearest Neighbors.

The initial code blocks are responsible for concatenating all the CSV files that will constitute the dataset.

After concatenation and saving the dataset files, they are processed to be usable with machine learning models, which means converting non-numeric data into numeric data. In the example case, the following operations are performed:

- For the **"History"** parameter, Binary Encoding is applied, which transforms the categorical variable by calculating all the values it takes and converting it into a series of columns to handle all cases.
- For the parameters **"protocol, service, conn_state,"** Ordinal Encoding is applied, counting the possible values and encoding them in the starting column.
- For the parameters **"orig_bytes and resp_bytes,"** cases where "-" values are present as null values are handled and converted to "0."

Finally, machine learning models are trained and tested in two major blocks using the Repeated Stratified K-Fold cross-validator:

1. In the first major block, after defining n_splits and n_repeats for the Repeated Stratified K-Fold cross-validator and the model on which to apply the cross-validator (the Random Forest is available, and KNN is commented out with pre-configured hyperparameter values), training and testing are performed, and subsequently, metric values (not differentiated by classes) such as:

   - fit_time,
   - score_time,
   - test and train_accuracy,
   - test and train_precision_weighted,
   - test and train_recall_weighted,
   - test and train_f1_weighted,
   - test and train_precision_macro,
   - test and train_recall_macro,
   - test and train_f1_macro.

   Based on these values, charts are generated showing trends over various repetitions performed by the cross-validator.

2. In the second major block, after defining n_splits and n_repeats for the Repeated Stratified K-Fold cross-validator and the model on which to apply the cross-validator (Random Forest is available, and KNN is commented out with pre-configured hyperparameter values), training and testing are performed. As a result, metric values for accuracy, precision, recall, and f1-score for different classes are displayed.

Bibliography

[1] AVAtlas, The threat intelligence platform , https://portal.av-atlas.org/

[2] ermetict, IBM Cost of Data Breach, https://ermetic.com/blog/cloud/ibm-cost-of-a-data-breach-2022-highlights-for-cloud-security-professionals/

[3] Malwarebytes, Ransomware revenue significantly down over 2022, https://www.malwarebytes.com/blog/news/2023/01/ransomware-revenue-significantly-down-over-2022

[4] Malwarebytes, What is malware?, https://www.malwarebytes.com/malware?lr

[5] J. T. Force, "Security and Privacy Controls for Information Systems and Organizations", NIST SP800-53r5, September 2020, DOI 10.6028/NIST.SP.800-53r5

[6] J. Barriga and S. G. Yoo, "Malware Detection and Evasion with Machine Learning Techniques: A Survey", International Journal of Applied Engineering Research, vol. 12, September 2017, pp. 7207–7214. https://www.researchgate.net/publication/320255510_Malware_Detection_and_Evasion_with_Machine_Learning_Techniques_A_Survey

[7] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A Survey on Malware Detection Using Data Mining Techniques", ACM Computing Surveys, vol. 50, June 2017, pp. 1–40, DOI 10.1145/3073559

[8] IDS/IPS Evasion Techniques, Alan Neville, https://anev.redbrick.dcu.ie/IDS_IPS_Evasion_Techniques.pdf

[9] F. A. Aboaoja, A. Zainal, F. A. Ghaleb, B. A. S. Al-rimy, T. A. E. Eisa, and A. A. H. Elnour, "Malware Detection Issues, Challenges, and Future Directions: A Survey", Applied Sciences, vol. 12, August 2022, p. 8482, DOI 10.3390/app12178482

[10] Ö. A. Aslan and R. Samet, "A Comprehensive Review on Malware Detection Approaches", IEEE Access, vol. 8, January 2020, pp. 6249–6271, DOI 10.1109/ACCESS.2019.2963724

[11] Y. Tang, B. Xiao, and X. Lu, "Using a bioinformatics approach to generate accurate exploit-based signatures for polymorphic worms", Computers & Security, vol. 28, November 2009, pp. 827–842, DOI 10.1016/j.cose.2009.06.003

[12] H. Borojerdi and M. Abadi, "MalHunter: Automatic generation of multiple behavioral signatures for polymorphic malware detection", Proceedings of the 3rd International Conference on Computer and Knowledge Engineering, ICCKE 2013, Mashhad, Iran, October 31 - November 1, 2013, pp. 430–436, DOI 10.1109/ICCKE.2013.6682867

[13] J. Newsome, B. Karp, and D. Song, "Polygraph: automatically generating signatures for polymorphic worms", 2005 IEEE Symposium on Security and Privacy (S&P'05), Oakland (CA, USA), May 08-11, 2005, pp. 226–241, DOI 10.1109/SP.2005.15

[14] R. Perdisci, W. Lee, and N. Feamster, "Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces", Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, San Jose (CA, USA), April 28-30, 2010, p. 26. https://dl.acm.org/doi/10.5555/1855711.1855737

[15] Y. Fukushima, A. Sakai, Y. Hori, and K. Sakurai, "A behavior based malware detection scheme for avoiding false positive", 2010 6th IEEE Workshop on Secure Network Protocols, Kyoto, Japan, October 05, 2010, pp. 79–84, DOI 10.1109/NPSEC.2010.5634444

[16] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant, "Semantics-aware malware detection", 2005 IEEE Symposium on Security and Privacy (S&P'05), Oakland (CA, USA), May 08-11, 2005, pp. 32–46, DOI 10.1109/SP.2005.20

[17] A. Khalilian, A. Nourazar, M. Vahidi-Asl, and H. Haghighi, "G3MD: Mining frequent opcode sub-graphs for metamorphic malware detection of existing families", Expert Systems with Applications, vol. 112, June 2018, pp. 15–33, DOI 10.1016/j.eswa.2018.06.012

[18] A. G. Kakisim, M. Nar, and I. Sogukpinar, "Metamorphic malware identification using engine-specific patterns based on co-opcode graphs", Computer Standards & Interfaces, vol. 71, April 2020, p. 103443, DOI 10.1016/j.csi.2020.103443

[19] W. C. Arnold and G. Tesauro, "Automatically Generated WIN32 Heuristic Virus Detection", Virus Bulletin Conference 2000, Orlando (FL, USA), September 28-29, 2000, pp. 51 – 60. https://api.semanticscholar.org/CorpusID:18981787

[20] Y. Ye, T. Li, Q. Jiang, and Y. Wang, "CIMDS: Adapting Postprocessing Techniques of Associative Classification for Malware Detection", IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 40, February 2010, pp. 298–307, DOI 10.1109/TSMCC.2009.2037978

[21] D. Bilar, "Opcodes as Predictor for Malware", Int. J. Electron. Secur. Digit. Forensic, vol. 1, May 2007, pp. 156–168, DOI 10.1504/IJESDF.2007.016865

[22] S. Naval, V. Laxmi, M. Rajarajan, M. S. Gaur, and M. Conti, "Employing Program Semantics for Malware Detection", IEEE Transactions on Information Forensics and Security, vol. 10, August 2015, pp. 2591–2604, DOI 10.1109/TIFS.2015.2469253

[23] B. A. AlAhmadi and I. Martinovic, "MalClassifier: Malware family classification using network flow sequence behaviour", 2018 APWG Symposium on Electronic Crime Research (eCrime), San Diego (CA, USA), May 15-17, 2018, pp. 1–13, DOI 10.1109/ECRIME.2018.8376209

[24] M. Piskozub, R. Spolaor, and I. Martinovic, "MalAlert: Detecting Malware in Large-Scale Network Traffic Using Statistical Features", SIGMETRICS Perform. Eval. Rev., vol. 46, January 2019, pp. 151–154, DOI 10.1145/3308897.3308961

[25] M. Piskozub, F. De Gaspari, F. Barr-Smith, L. Mancini, and I. Martinovic, "MalPhase: Fine-Grained Malware Detection Using Network Flow Data", Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, Hong Kong, China, June 7-11, 2021, pp. 774–786, DOI 10.1145/3433210.3453101

[26] J. Ha and H. Roh, "Experimental Evaluation of Malware Family Classification Methods from Sequential Information of TLS-Encrypted Traffic", Electronics, vol. 10, December 2021, p. 3180, DOI 10.3390/electronics10243180

[27] javatpoint, Decision Trees, https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm

[28] Analytics Vidhya, Understand Random Forest Algorithms With Examples (Updated 2023), https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

[29] Rohith Gandhi, Support Vector Machine - Introduction to Machine Learning Algorithms, https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

[30] javatpoint, K-Nearest Neighbor(KNN) Algorithm for Machine Learning, https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning

[31] IBM, What are neural networks?, https://www.ibm.com/topics/neural-networks

[32] D. McGrew and B. Anderson, "Enhanced telemetry for encrypted threat analytics", 2016 IEEE 24th International Conference on Network Protocols (ICNP), Singapore, 8-11 November, 2016, pp. 1–6, DOI 10.1109/ICNP.2016.7785325

[33] Cisco, Cisco Encrypted Traffic Analytics, https://www.cisco.com/c/en/us/solutions/collateral/enterprise-networks/enterprise-network-security/nb-09-encrytd-traf-anlytcs-wp-cte-en.pdf

[34] A. Thakkar and R. Lohiya, "A review of the advancement in intrusion detection datasets", Procedia Computer Science, vol. 167, 2020, pp. 636–645, DOI https://doi.org/10.1016/j.procs.2020.03.330

[35] O. Barut, Y. Luo, T. Zhang, W. Li, and A. Peilong Li, NetML: A Challenge for Network Traffic Analytics, https://arxiv.org/abs/2004.13006

[36] Malware-traffic-analysis, https://www.malware-traffic-analysis.net/

[37] Stratosphere IPS, https://www.stratosphereips.org/

[38] Computational resources provided by hpc@polito, which is a project of Academic Computing within the Department of Control and Computer Engineering at the Politecnico di Torino, http://www.hpc.polito.it

[39] Google Colaboratory, https://research.google.com/colaboratory/faq.html#whats-colaboratory

[40] Zeek Framework, https://docs.zeek.org/
[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python", Journal of Machine Learning Research 12, vol. 12, October 2011, pp. 2825–2830. https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html
[42] geeksforgeeks, Stratified K Fold Cross Validation, https://www.geeksforgeeks.org/stratified-k-fold-cross-validation/
[43] Machine Learning Mastery, Repeated k-Fold Cross-Validation for Model Evaluation in Python, https://machinelearningmastery.com/repeated-k-fold-cross-validation-with-python/
[44] Matplotlib, Visualization with Python , https://matplotlib.org/
[45] seaborn, statistical data visualization, https://seaborn.pydata.org/
[46] v7labs, F1-score, https://www.v7labs.com/blog/f1-score-guide
[47] Siddharth M., Analytics Vidhya, Let's Understand The Problems with Recurrent Neural Networks, https://www.analyticsvidhya.com/blog/2021/07/lets-understand-the-problems-with-recurrent-neural-networks/