uc3m | Universidad Carlos III de Madrid

Bachelor's Degree in Computer Science and Engineering.
2018-2019

Bachelor's Thesis

"Intelligent Android Malware Family Classification using Genetic Algorithms and SVM"

Sara Yuste Fernández Alonso

Pedro Isasi Viñuela

Yago Sáez Achaerandio

Leganés, July 2019

ABSTRACT

As of April 2019, Android was the most popular mobile operating system amongst smartphone users[1]. Its high popularity, combined with the extended use of smartphones for everyday tasks as well as storing or accessing sensitive and personal data, has made Android applications the target of numerous malware attacks over the last few years and in the present.

The malware attacks have been perfected to target specific vulnerabilities in the operating system or the user; thus specializing in types of malware and families within each type. The malware is usually distributed in infected applications (or APKs), which contain malicious behaviours that can be found looking into their code (known as static analysis) or analysing the behaviour of the application while running (known as dynamic analysis).

This document describes the implementation of an intelligent system that aims to classify a series of malicious APK samples obtained from the free repository *ContagioDump*. These samples are classified inside the type and family they belong to.

To create the classifier system, a Support Vector Machine (SVM) is implemented using Python's library *Scikit Learn.* A series of attributes are extracted from the samples of malicious APK by analysing the code of the APKs via static analysis, using Python's library *Androguard*, which contains a parser that allows to interact with all the relevant parts of the APK file.
The attributes obtained are very high in number, and for that reason a Genetic Algorithm is used to optimize the attributes that the SVM uses in the learning process. The algorithm codifies a subset of attributes from all the attributes extracted in the static analysis, and is evaluated using the accuracy score obtained when training the SVM with said subset.

As a result, a subset of attributes and a trained model for the classification are obtained. This model is then tested with a new set of malware samples, belonging to all the families classified in the learning.

The present document contains the explanation of the process of designing, creating and testing the system. It is developed as bachelor's thesis for computer science and engineering degree in Universidad Carlos III de Madrid.

Keywords

Genetic algorithms, Neural networks, SVM, Android, APK, malware, Artificial Intelligence.

RESUMEN

En abril de 2019, Android era el sistema operativo móvil más popular entre los usuarios de *smartphones*[1]. Debido a su popularidad, y el uso extendido para tareas diarias entre los usuarios, que además usan sus teléfonos móviles para guardar y acceder datos sensibles y privados, Android ha sufrido un gran número de ciber ataques en los últimos años, y sigue recibiendo ataques constantemente.

Estos ataques maliciosos se han ido especializando para atacar a vulnerabilidades específicas del dispositivo o de los usuarios, diferenciándose en tipos y familias de *malware*. Este malware se distribuye habitualmente en aplicaciones (o APKs) infectadas. Es posible analizar el comportamiento malicioso de estas aplicaciones infectadas, bien analizando el código de la aplicación (conocido como análisis estático) o estudiando el comportamiento de la aplicación mientras es ejecutada (análisis dinámico).

El presente documento describe la implementación de un sistema inteligente de clasificación de muestras de malware en tipos y familias de malware. Las muestras utilizadas son una serie de APKs infectadas obtenidas del repositorio gratis *ContagioDump*.

La creación del sistema clasificador se ha llevado a cabo desarrollando un programa que usa una Máquina de Soporte Vectorial (SVM, por sus siglas en inglés), haciendo uso de la librería de Python *Scikit Learn*. Las muestras de APKs maliciosas se analizan de forma estática para obtener una serie de atributos, usando la librería de Python *Androguard*, que proporciona un parser y una interfaz para interactuar y utilizar todas los elementos relevantes del código de las APKs.
El número de atributos obtenidos en dicho análisis es muy alto, por lo que se utiliza un algoritmo genético para optimizar el proceso de aprendizaje de la SVM, seleccionando un subgrupo de atributos que se usan en el aprendizaje. El algoritmo genético codifica el subgrupo de atributos a usar, y es evaluado según el porcentaje de acierto obtenido al entrenar la SVM con el subgrupo codificado.

Como resultado del trabajo, se obtienen un subgrupo de atributos óptimos en los que basar el análisis de una APK, y un modelo clasificador entrenado. Este modelo se pone a prueba con una nueva serie de muestras de aplicaciones maliciosas, representativas de todos los tipos y familias analizados anteriormente.

Este documento incluye la explicación del proceso de diseño, creación y evaluación del sistema implementado. El sistema ha sido desarrollado como Trabajo de Fin de Grado de la carrera de Ingeniería Informática en la Universidad Carlos III de Madrid.

Palabras clave

Algoritmos genéticos, Redes de neuronas, SVM, Android, APK, malware, Inteligencia Artificial.

AKNOWLEDGMENTS

I’d like to thank all the people who have helped and supported me during the creation of this work, without whom this thesis could have never been possible.

To my supervisors, Pedro Isasi and Yago Sáez, for all their corrections, advice and support during all the development of this thesis.

To my boyfriend Santi, and friends, who have been supportive and lifted me up through the most stressful moments.

I would also like to thank all the professors and staff from the university that have contributed to making my university experience valuable and positive. To the cleaning and maintenance staff that keeps the campus a pleasant space, the staff from dining and café areas, and all my professors, both lecturers and lab professors.

TABLE OF CONTENTS

TABLE OF FIGURES

INDEX OF TABLES

1. INTRODUCTION

This document describes the implementation of an intelligent Android malware classification system. The objective of the developed system is to classify a series of malicious Android applications (or APKs) samples in the types and families of malware which they belong to.

In order to develop this system, the code from the APK samples is analysed to obtain information about its behaviour and characteristics, and a Support Vector Machine is used to classify the samples analysed. A genetic algorithm is implemented to optimize the learning of the Support Vector Machine, by selecting a subset of characteristics or features from the samples which the SVM uses to train.

1.1. Motivation

As of April 2019, Android operating system was the most popular operating system amongst smartphone uses with 70.22% of the market share[1]. The everyday use of smartphones has been growing unsteadily for the past few years, and more and more users are now using their smartphones to store and access personal and sensitive data, such as banking information. It has also become an essential asset for users, who rely on their smartphones for many of their daily tasks.

For these reasons, Android OS is often targeted by malicious applications that aim to steal data or damage the device. New families of malware are discovered daily, and cyber security experts often struggle trying to keep up with all the new malicious applications, evaluating their risks and which users they might affect.

Although there are many effective extended methods to detect whether an application is malicious, even available in the market (most mobile antivirus services perform this task), there haven't been so many attempts at trying to identify the family of malware an application belongs to, which could eventually help cyber security experts determine the threat it poses more efficiently, and therefore act on it sooner.

One of the reasons why this last task is not so commonly found in antivirus or other similar services is the continuously changing nature of malware families. There are new versions coming out almost on a daily basis, and trying to find a deterministic way to decide which family a malicious APK belongs to seems almost impossible and not scalable.

Artificial Intelligence methods present an alternative way to perform this classification. Due to the ability of AI algorithms to learn, adapt and generalize, an AI based system could bring an scalable and adaptable solution to this problem.

1.2. Objectives

The objective of the work describe here is implement an intelligent classifier that provides an efficient and scalable solution to the problem described before, with the final purpose of classifying Android malware samples in their right type and family of malware.

In this document, an intelligent classification using AI algorithms (artificial neural networks, more precisely a Support Vector Machine) is proposed. Furthermore, to obtain a more efficient system, a Genetic Algorithm is used as an optimizer for the features used in the learning.

By using these two methods in combination, an adaptable solution is found, which performs a classification of the malicious apps in an effective and adept way. This solution can be used in further research and help cyber security analysts in the early detection of threats.

The project will be developed entirely using open source means, to allow easier further implementations, investigation and improvements. For the development of the project, a series of malicious APK samples are obtained from the free online repository ContagioDump, and the implementation of the system is done in Python programming language using the libraries *Scikit Learn* and *Androguard*.

1.3. Document Structure

The present document is structured as it follows:

First, a **STATE OF THE ART** is detailed. This section gives an explanation of all the techniques and theory relevant for the work explained in this document, along with some similar work developed in the area.

then, an exhaustive definition of the developed system is given. this definition is divided in three sections, ordered from more high level detail to low level detailed functioning of the system: first section is **SYSTEM ANALYSIS**, where the most important components and requirements for the system are identified. in the second place, the **SYSTEM DESIGN** is shown, including the technologies, detailed components and classes of the system. lastly, the **IMPLEMENTATION** of the system is included, with low level detail of all the components of the system.

Later, the tests used to measure the performance of the system are detailed in the **EVALUATION** section, followed by an analysis of the **MANAGEMENT** (budget and planning) and **LEGAL AND SOCIO-ECONOMIC ENVIRONMENT** related to the work described in this document. This last section aims to understand the legal and socio economic implications this work has, related to topics such as intellectual property or data protection.

Finally, a series of personal and technical **CONCLUSIONS** are presented.

2. STATE OF THE ART

2.1. Android

Android is a mobile operating system (OS) is based in the Linux operating system, developed by Android Inc., first meant to improve the operating system of digital cameras. In 2004, Android Inc. decided to use the OS in mobile phones. The company was later acquired by Google in 2005.

The first public version of Android OS was launched in 2007. Earlier that same year, Apple had launched the first iPhone. Unlike iPhone's operating system iOs, Android could power many different phone models. It quickly gained popularity amongst smartphone users.

Android was ranked as the most popular mobile operating system in April 2019 by **Net MarketShare[1]**. Due to its high popularity, it has quickly become an interesting target for numerous malware attacks.

2.1.1. APK

An Android Package Kit (APK) is the file format used for distributing and installing mobile apps in the Android operating system.

APKs can be installed from a computer or from the mobile device. The most common installation method is using the device's official store application Google Play Store, although the installation can be done manually.

There are other alternative Android app stores, but some of these are not reliable as they contain malicious software (malware) apps.

An APK file contains all the source code for a certain application. When an android app contains malicious behaviours, it is possible to analyse the source code found in the APK file to find these behaviours.

DEX

DEX (Dalvic Executable) is the compiled code of an Android program. An Android application is defined by the .dex files which are then zipped to a single .apk file, along with other elements that are not relevant for the work described in this document.

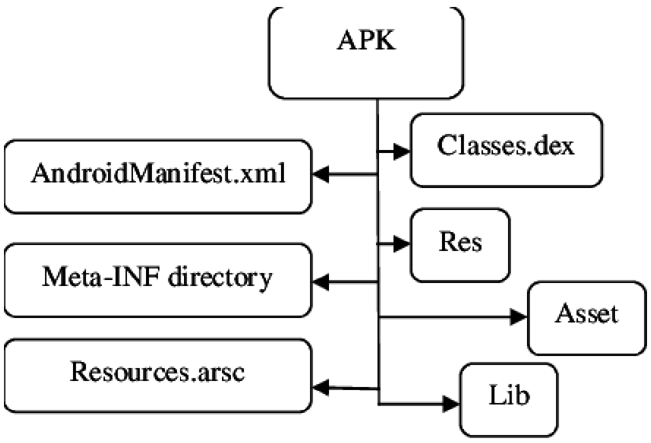


Figure 1: Structure of an APK

2.1.2. Android Malware

Malicious software (or malware) applications are applications that seek to find vulnerabilities in the system or the user and exploit them to either cause damage to the system or the device, or obtain sensitive information.

With the growth of popularity and usage of personal mobile devices, malware targeted at mobile operating systems has become increasingly popular over the last few years. Mobile phone users store vast amounts of personal information (contacts, pictures, credentials) in their devices, and use their mobile phones for many daily activities (such as business, social, information search).

By infecting a personal mobile device with malware, the attacker can gain access to user's most sensitive and personal information. Furthermore, if the device is damaged or the user loses access to it, it can cost the user a high timely or economical investment to recover.

Android operating system has become a preferred target to attackers for two main reasons: first, because it is the most popular mobile operating system amongst smartphone users. Second, because an Android allows to view the user's activity in real time; thus an attacker can intercept a safe application's launch and display the malicious app instead, without the user noticing. This vulnerability becomes particularly interesting for some types of malware as it will be explained later on.

For these reasons, Android attackers have developed different specialized malware applications aimed to attack different vulnerabilities.

Android malware types

Malware applications can be grouped in types attending to which vulnerability is attacked. For each type, there are certain "families" of malware applications that behave similarly. There are also "versions" of the families previously mentioned. The scope of this work is limited to identifying types and families of malicious APKs.

Some of the most popular malware types are listed and explained below.

Bankers

These malicious apps are aimed at stealing the user's banking related information that is on the user's device.

The target of these malware apps are mobile banking apps. Mobile banking apps are applications that allow a user to access their bank accounts comfortably from their smartphones, and perform any transactions with them.

Banker malware typically impersonates the user's mobile banking app, by using similar interface and logo, and then captures the user's credentials (account number, log in details) as they attempt to log in. This allows an attacker to directly steal from the user's bank account.

Overlay of banker malware on top of official baking app is shown on Figure 1.

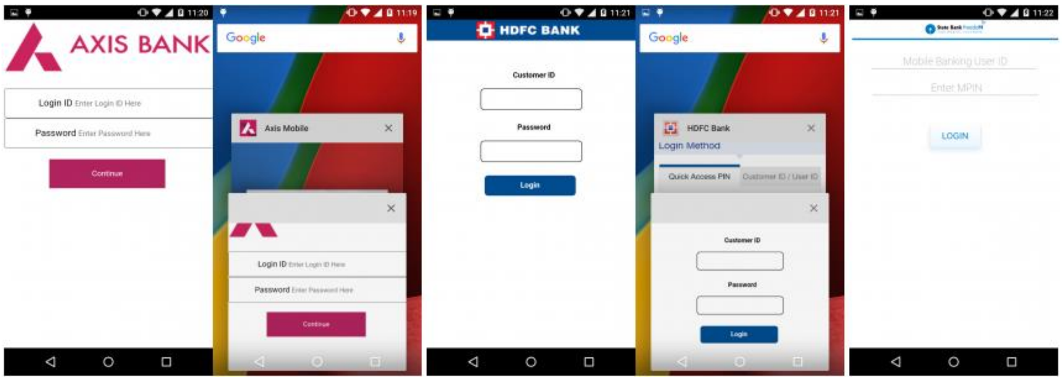


Figure 2: Overlay on banking Apps

Ransomware

Ransomware apps encrypt all or some of the data stored in the user's device, preventing the user from accessing it until they agree to pay a ransom, usually via an anonymous internet payment.

Spyware

Spyware is a type of malware that infiltrates in the user's device, then collects and stores information from the user, including internet usage data and personal or sensitive information,. Its usual purpose is to then sell the user's internet usage data, capture credit card or bank information, or steal the user's personal identity.

Adware

Adware hides in the user's device and serves the user advertisements. It sometimes also stores information about the user's behaviour and preferences to later use this information to target the user with certain ads. This software generates revenue either by getting paid by the advertisers to display a certain advertisement, or via "pay-by-click" if the user clicks on the advertisement.

Exploit

Exploit malware takes advantage of vulnerabilities in the software or security flaws to gain access to private networks and scale privileges. This can allow a remote intruder to access a device or a network remotely. Sometimes it is used to infiltrate other malware like Trojans or Spyware. Exploit malware is sometimes sent to the user via email or other web sites, or the user is lured into executing the exploit via social engineering.

Trojan

A Trojan is a type of malware that is usually hidden or disguised as legitimate software. Once the user has installed the Trojan in the system, it gains access to the user's data and can delete, block, modify, copy or disrupt the normal functioning of the device. There are several types of Trojans, such as Backdoors, which provide remote access to the device to the attacker, or Exploits, as explained above. Any of the malware types described before can be installed in the device via a Trojan.

Android malware families

Figure 2 shows some of the families that can be found in each of the malware types described above. Due to the amount of android malware apps, there is a gross number of families of each type; thus there are many other families that aren't shown in this figure.

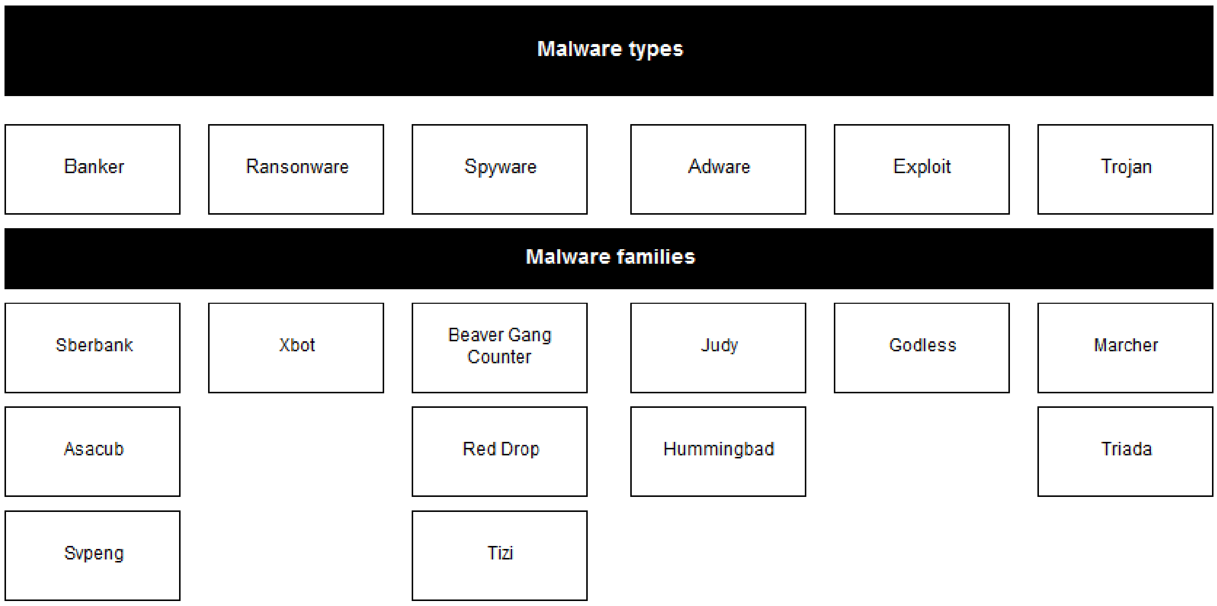


Figure 3: Malware types and families

Android malware analysis: static and dynamic

An Android APK can be analysed to detect malicious behaviours using two methods: Static and dynamic analysis.

Static analysis uses the code of the application to extract attributes that can determine the way the application performs. The code can be found in the .dex component of an apk, which contains the compiled code of the whole application (as seen in 2.1.1). The attributes are extracted without running the application. Some examples of these attributes are: system calls, permissions the application requests, calls to the Android library, all methods inside an APK, and how they interact within each other, flow diagrams (which illustrate all the function in the APK's code, taking the *main( )* function as start point, trying to represent the whole functionality of an APK in a graph). There are other attributes that can be extracted without running the application which are not found in the compiled code, such as the size of the APK.

Dynamic analysis, on the other hand, extracts attributes from the application while it's running. These can be network traffic, battery usage, sent SMS and phone calls, information leaks, etc. Dynamic analysis is costly in time and memory when performed. However, it is unaffected by techniques that aim to make static analysis more difficult, such as code obfuscation, which consists in transforming the code into a semantically equivalent version (with identical functionality) but much harder to understand by an analyst. Obfuscation can also be used for other purposes, such as protecting intellectual property. Some obfuscation methods can be found in *Different obfuscation Techniques for Code protection*[2]

Static analysis provides a quick, low cost way to analyse APKs, and has been used in multiple occasions for malware analysis (as shown on 2.3.1). However, dynamic analysis can be more reliable, as it is unaffected by code obfuscation techniques.

Some authors use **hybrid** approaches, where static and dynamic analysis are combined to extract a richer set of attributes compared to either analysis performed separately.

2.2. Artificial Intelligence

Artificial Intelligence is an area of computer science that aims to simulate the intelligent behaviours of humans in machines. It includes a wide range of techniques that can mimic some human abilities (such as: computer vision, speech recognition, natural language processing); exploit the machine's high computational power to improve efficient solving of problems, such as optimization or decision models; and can implement learning.

2.2.1. History

The origin of Artificial Intelligence (AI) is unclear, as it is based on the work of many mathematicians and scientists who started theorizing about machines that could solve complex problems inspired by human-like reasoning since the 17th century[3]. However, most authors[3][4][5] place the work of English mathematician Alan Turing during the 1940s and 1950s as the starting point of AI.

In 1936, Alan Turing published "On Computable Numbers, with an Application to the Entscheidungsproblem"[6], a paper where he proposed a "universal machine"; a computer capable of solving any computable function, nowadays known as Turing machines, which provided the basis for the theory of computation.

During the Second World War, he worked on breaking the machine the Germans were using to encrypt all their messages, *Enigma*. Turing and his team designed and built *the Bombe,* a machine that could decipher *Enigma's* code, based on his previous work.

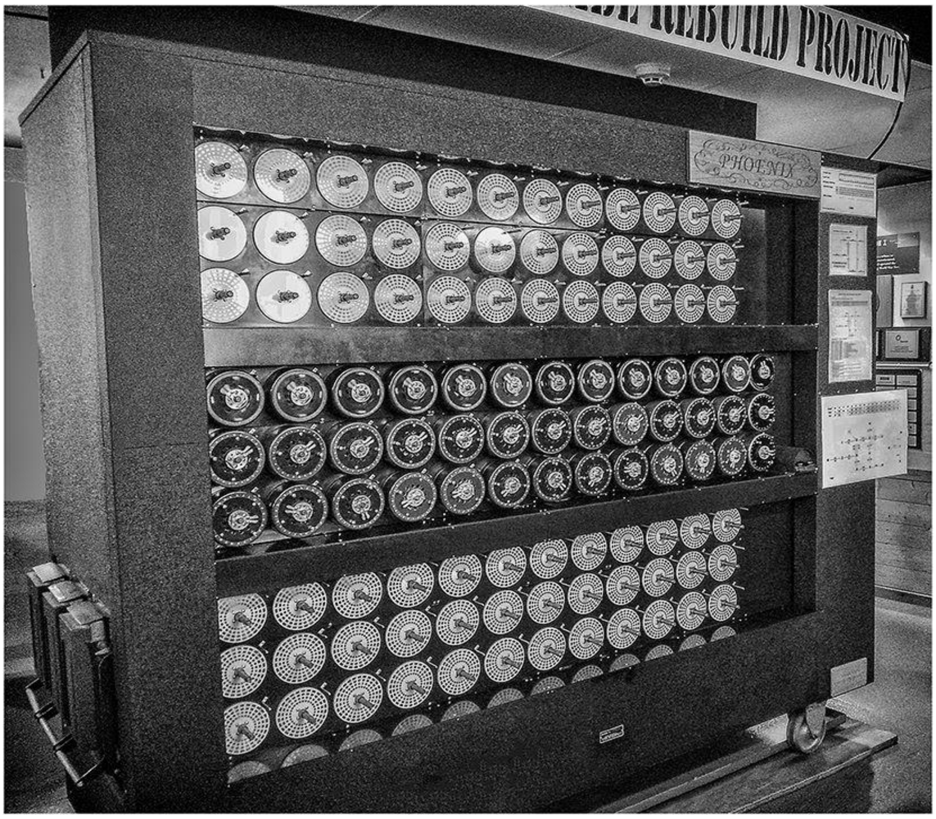


Figure 4: The Bombe

After his work during WWII, he became more interested in the concept of being sentient. This thought was the foundation for his later research regarding machine intelligence. In 1950, he published “Computer Machinery and Intelligence”[7], a research paper where he theorized about the creation of machines capable of “thinking”. On the same paper, he proposed what is known as *the Turing Test* to test the machine’s intelligence. The fundamental idea of this test, which he called “The Imitation Game”, was to test the machine’s intelligence based on its ability to make a human believe that it (the machine) is human, when engaging in conversation.

The first reference to the term “Artificial Intelligence” was made during the Dartmouth Conference, organized by computer scientist John McCarthy in 1956. Since then, numerous computer scientists and researchers have worked on the field of AI, trying to solve different problems. One of the most famous AI problems was creating a machine that was capable of playing Chess. The first paper about developing a chess playing program was written by Claude Shannon in 1950[8]. It wasn’t until 1997 that IBM’s Deep Blue defeated the then world Chess Champion.

The interest on the field of Artificial Intelligence has experienced progressions and regressions over time[9]. It was popular until the 1960s, but the little progress in the learning capabilities of the existing models resulted in a decrease in interest until the 1980s, when some successful applications were achieved, as well as more funding was provided.

From its origins in the 1950s, there are two approaches that can be differentiated within Artificial Intelligence: the first one based on logic, using formal rules to manipulate symbols; and the second one based on biology, such as artificial neural networks, which are inspired by the functioning of biological brains.

For the first 20 years after 1950, research was focused on the logic based approach mentioned above. Although the first mathematical model of neurons dates back to 1943, the biology based approach didn’t receive much attention during that period. In the 1980s, a new algorithm for learning in neural networks was reinvented[10] (it had already been proposed in 1963[11]), resulting in an increased interest in this type of algorithms.

However, as researchers became more interested in Artificial Intelligence, the field grew and new algorithms and techniques appeared, creating new more complex divisions within the field as the logic based as opposed to biology inspired division mentioned above.

In the next section of this document, a detailed explanation of the different techniques and algorithms known in AI is presented.

2.2.2. Techniques

The objective of this section is to explain the different techniques or algorithms known in the field of Artificial Intelligence in depth. Since it’s such a broad field of study, it is often hard to find a clear way of organizing and presenting all techniques. In this document, a division proposed by Francesco Corea[12] will be used as reference.

First, each sub division and some of the most relevant elements of such sub division are explained. The full “map of AI” with the techniques classified on each division is shown after the explanations. Finally, a more detailed explanation for the two algorithms used in the work proposed in this thesis is given.

As proposed by Corea, the field of AI can be divided attending to two dimensions: **AI paradigms** (approaches to solve problems) and **AI problem domains** (types of problems).

AI Problem domains

When looking at the types of problems an AI approach can solve, Corea defines five main domains: **Perception**, **Reasoning**, **Knowledge**, **Planning** and **Communication**.

To better understand each domain, it is interesting to compare it with the human cognitive ability it aims to mimic.

Perception includes the problems that in humans “solve” using their senses. It includes the techniques capable of operating with sensorial inputs (sounds, images, etc) by converting them to a usable format; for example Natural Language Processing and Computer Vision. Natural Language Processing allows a machine to understand, process and even create information in the form of human speech (that is, not structured data expressed in a human language). Computer Vision includes all techniques that allow a machine to process and understand images captured with a camera, similarly to human vision.

Reasoning refers to the capability to solve problems. This includes the capability of, given a problem definition, being able to offer a solution to said problem, in a similar way to how humans solve mathematical problems. An example of reasoning can be found in tagging pictures: deciding if a picture given has a cat or a dog in it. Here, the definition of the problem is the description of the image and the question “does this picture have a cat or a dog in it?”; and the solution would be saying “it has a cat” or rather “it has a dog”.

Knowledge is the capability to represent and understand the world. The world is the reality that affects the machine, and can be narrowed down in some cases. For example, some automations executed by robots only take into account the actions and elements the robot can interact with; if the robot is tasked with loading boxes to a truck, “the world” is limited to the actions it can perform (*up, down, grab, release,* for example) and the elements it can interact with (*box, truck*).

Planning is the capability of setting and achieving goals. An example for a planning problem would be deciding a plan to get from point A to point B, as when planning a road trip.

Communication is the capability of understand language and communicate. It differs from Natural Language Processing(NLP) in the communication between the machine and the human; NLP problems aim to process the language input, not necessarily trying to communicate.

AI Paradigms

AI Paradigms refer to the different approaches or types of algorithms that exist in the field. Corea divides this dimension in three main types: **Symbolic**, **Statistical** and **Subsymbolic**.

Symbolic approaches use logic based and knowledge based algorithms to solve problems. It manipulates symbols, with inference and search algorithms, to build rules, ontologies, plans or goals. Sometimes referred to as "GOFAI" (Good Old Fashioned AI), as author John Haugeland named it in his book "Artificial Intelligence: The Very Idea"[13].

Subsymbolic AI (also known as "connectionist AI") was originally inspired by the biological brain. It creates connections between nodes, creating a network, and performs calculations in the connections of said network that provide a solution. The outcome model could be compared to a connection map, opposed to the rule tree or plan that is obtained with symbolic AI. To compare it with the biological brain, it assigns conductivity properties (or weight) to the connections between neurons (or nodes) and then modifies this conductivity for each connection until, when a problem is passed through the network, the outcome is a solution to the problem.

Compared to symbolic AI, Subsymbolic AI provides less knowledge and understanding upfront and is more difficult to explain, but performs better for perceptual problems. It is also more scalable, and more robust against noise. The opacity of this paradigm is a problem known by researchers[14], but that doesn't deny its many applications in AI problems.

Figure 4 illustrates the different understanding of the solution provided by the models obtained with symbolic and subsymbolic AI.

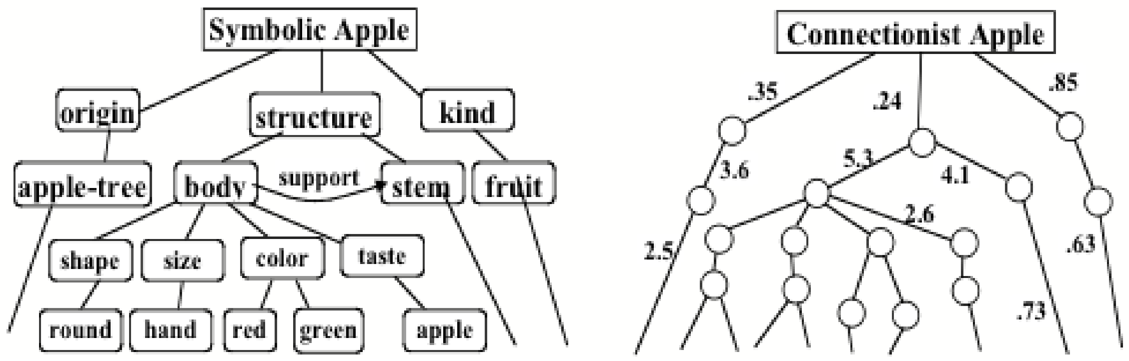


Figure 5: Symbolic VS Sub symbolic AI

Finally, **Statistical** AI uses probabilistic methods and mathematical tools to build models that reflect information about data. Machine learning algorithms, which have become increasingly relevant over the years[15], are a subdivision of this approach.

Machine Learning

Machine Learning algorithms are broadly used due to their ability to provide general solution to a problem using specific samples as reference (or “learn”) and their applications in many different kinds of problems.

The functioning of machine learning algorithms can be summarized as the creation of mathematical models using sample data (“training” data), that describe patterns found in said data. Once the model has been created, it can be shown similar data that wasn’t used to train it, and find a similar patterns in it, which allows the model to recognize this new data. This capacity of being able to create general solutions from specific examples is known as learning.

A metaphor to understand the process of machine learning can be created with the learning process of a human. When a human is learning to read and write, he or she is shown a series of “perfect” letters of words – usually generated in a computer, and asked to write them down. By seeing these symbols repeatedly, the human is eventually able to recognize them even when they’re not perfect, such as in handwriting.

In a similar way, a machine learning algorithm can be shown a series of letters, or numbers, and it will find patterns in the symbols. After the learning, if the algorithm is shown a new type of handwriting containing the same symbols the training data provided, it will be able to generalize the pattern learned and recognize said symbols.

In order to learn from the training data, a series of “features” or characteristics must be extracted from it. These features describe the data point and are usually expressed as a vector of values x_i which define the data point. For example, in the example given, the features extracted from a picture of a handwritten symbol could be: RGB value for each pixel in the image, dimensions of image, etc. These features must be defining of all possible images in the training data and it is required that they can be extracted from all data points.

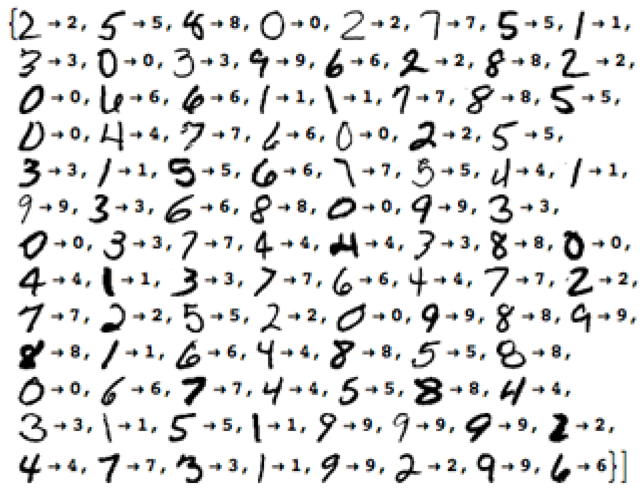


Figure 6: Example of automated handwriting recognition using Machine Learning

Within machine learning, are three types of learning: Supervised, unsupervised and reinforcement learning.

SUPERVISED LEARNING

Supervised learning uses solved examples to train from. The training data provided contains not only examples of the problem, but a tag or value with the solution that would be valid for said example.

In the handwriting example explained before, supervised learning would need a series of example handwritten numbers of letters, along with a tag for the value they represent. The set shown in Figure 5 would be a valid training set for supervised training.

UNSUPERVISED LEARNING

Unsupervised learning lacks a value or tag for each example or sample in the training data. It is given a series of data points or samples, and finds patterns in them according to the distance between the points. For this learning to work it is needed to define the calculation of distance between points.

An example of an unsupervised learning would be trying to sort a drawer with pens and pencils. One can sort them by color, shape, size… since there is no prior definition to which sorting is correct, the groups which will be formed are unpredictable.

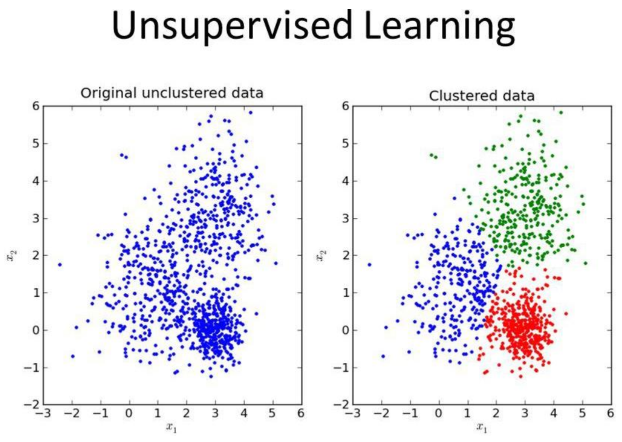


Figure 7: Unsupervised Learning

REINFORCEMENT LEARNING

Reinforcement learning assigns a value of “reward” or “punish” for each action possible. The algorithm will tend to maximize reward (or minimize punishment). It can be compared to the process of training a pet to learn certain tricks; giving treats when the pet does the trick correctly, so the pet learns to perform the trick more often in order to get the reward.

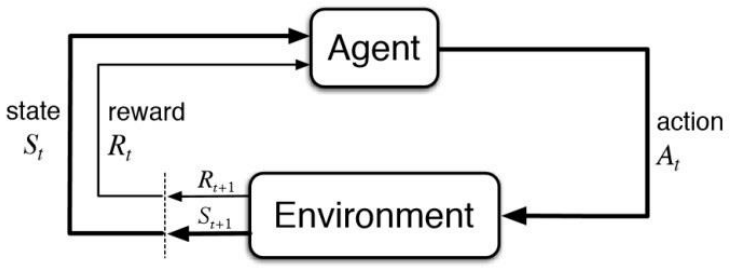


Figure 8: Reinforcement learning

Map of AI

The divisions explained above can be better understood when presented in the “Map of AI” proposed in the article by Corea[12] (Figure 8). As seen on this map, some fields are not entirely belonging to one of the divisions inside a dimension, and many of them can be used for a wide range of problem domains. The two techniques used in the development of the model are further explained down below.

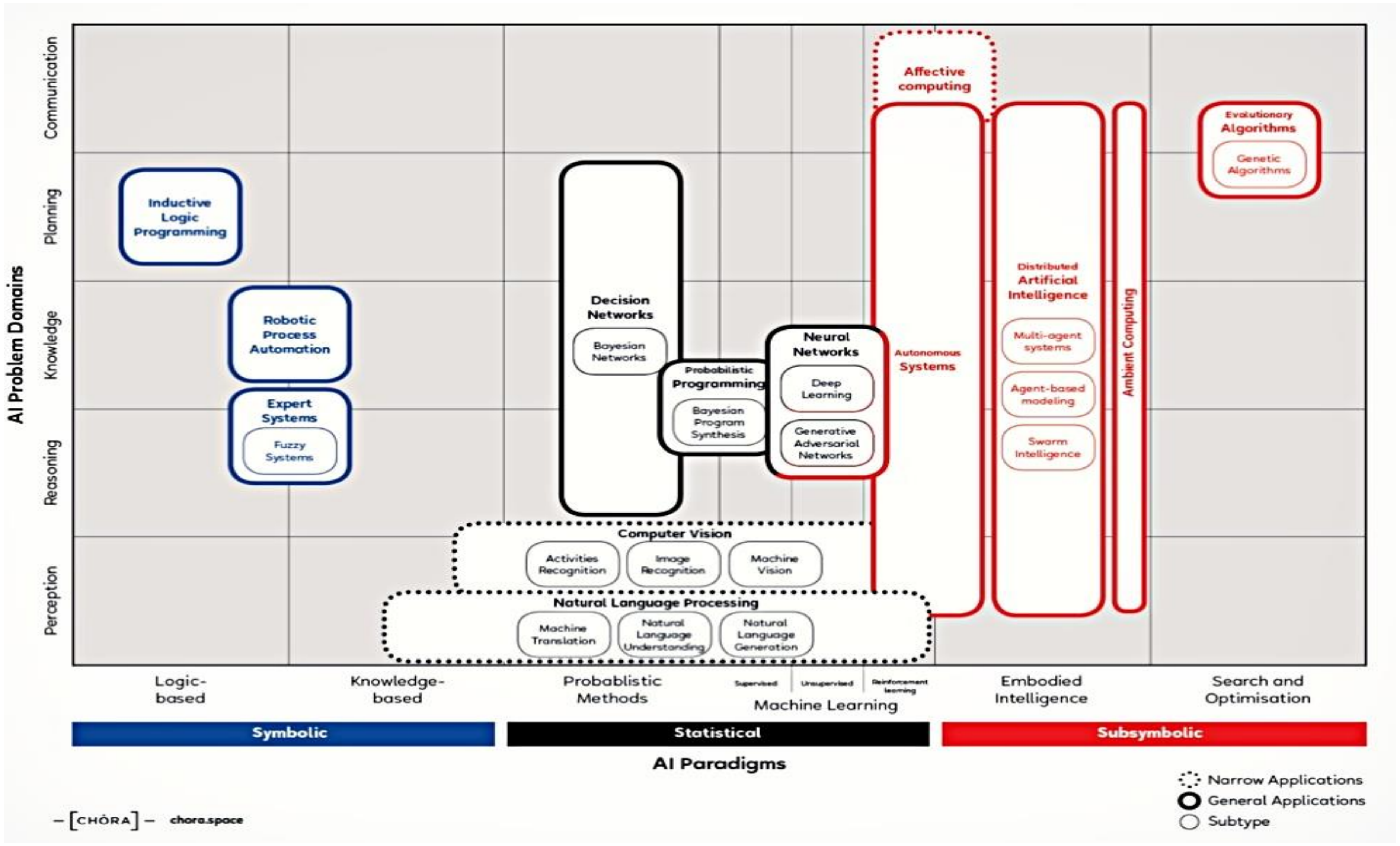


Figure 9: Map of AI (Corea, 2018)

Genetic algorithms

Genetic algorithms are a subset of evolutionary algorithms, used to solve optimization problems.

Evolutionary algorithms are inspired by biological evolution, where individuals compete for resources, randomly combine their features to create new individuals and suffer random alterations (mutations) in time. Each individual can be defined as more or less fit (fitness score) to adapt to the environment, and therefore more or less likely to survive and pass on its features to the next generation.

Similarly, a genetic algorithm contains a population of individuals and a series of operators that can be applied to each individual. Each individual codifies a possible solution to the optimization problem at hand (called "**genome**"), and the possible operators are **mutating** the individual, calculating the fitness value for each individual and **selecting** the best individual who will contribute to the creation of new individuals, combining the individual's codification with other individual (**crossover**), and finally replacing or adding new individuals to the population.

As well as in biological evolution, any possible way of selecting which individuals combine their features, mutating, or even competing (or collaborating) can be implemented. The population will tend to "evolve" towards the optimal solution, which would be the "perfect" individual.

Artificial Neural Networks: SVM

Artificial neural networks are a family of algorithms loosely based on the architecture and functioning of the biological brain.

An artificial neural network (ANN) is formed by several layers of nodes or neurons. The neurons on each layer are connected to the neurons on the next layer, with a "connectivity strength" (called "weight) associated to each connection.

One of the first neural networks proposed was the simple perceptron. This model has only one layer and one neuron in said layer. The input data is a vector of values x_i which are connected to the neuron with a corresponding vector of weights w_i for each connection.

The neuron then combines both vectors as follows:

$$f(x) = \sum_{i=1}^{N} (x_i w_i) + \theta$$

$$f(x) = x_1 w_1 + x_2 w_2 + \cdots + x_N w_N + \theta$$

Where θ , or "bias", is independent from all input values.

After that combination is performed, the output value is used in the *activation function*, a function that defines the solution or output given by the model. For example, this function can be a simple threshold function such as:

$$f(x) = \begin{cases} 1 \; if \; \sum (x_i w_i) + \theta > 0 \\ -1 \qquad\qquad otherwise \end{cases}$$

Where 1 and -1 are different classes. For example, if trying to classify a handwritten symbol as a number or a letter, 1 could mean it is a number and -1, a letter.

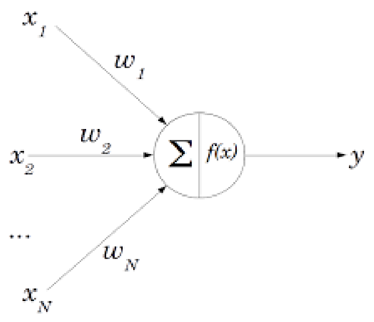


Figure 10: Simple perceptron

A neural network connects several neurons which can behave like the model explained above, in several successive layers. The fist layer is called input layer, the last layer is called output layer, and the layers in between are known as hidden layers. It receives a vector of values x_i with a series of weights w_i and combines them in successive layers by applying a function to the values and weight in each neuron. The output of each layer is then processed by the next layer in a similar way.

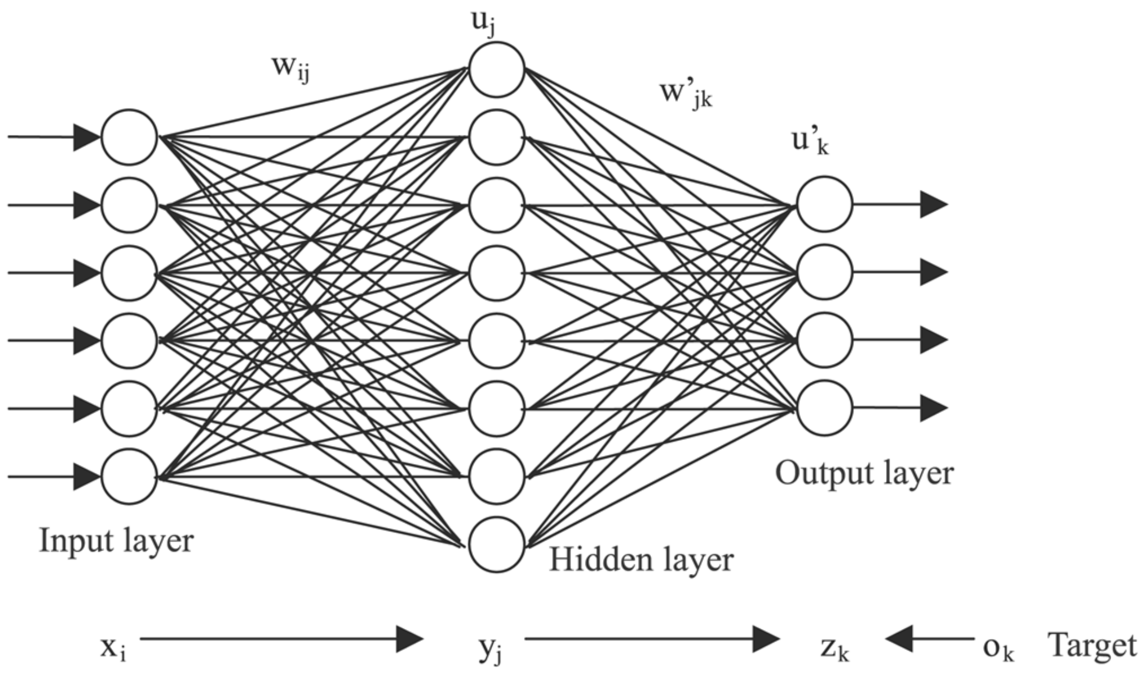


Figure 11: Artificial Neural Network

Support Vector Machine (SVM)
A support vector machine is a type of Artificial Neural Network which is widely used in classification problems.

The goal of the SVM is to find a hyperplane in an *n* dimension space (n being the number of features) that separates the data points of different classes.

It can be imagined as a wall (in a 3 dimension space) that separates two species of animals in a room; the goal of the learning is to move the wall along the room until there are no two animals from different species together. There are many hyperplanes that give a solution, the goal is to find the one that maximizes the distance between all data points of each class: this is done so future data points can be classified with more confidence.

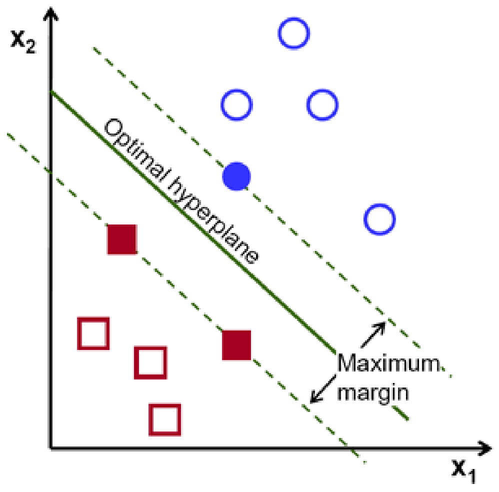


Figure 12: Support Vector Machine

Support vectors are data points close to the hyperplane and influence the latter's position. They help maximizing the distance between data points of different classes.

The process of learning in a SVM can be seen as changing the position of the hyperplane and support vectors until the maximum distance between data points is achieved.

2.3. Similar work

Due to the popularity of Android amongst users, Android devices face constant threats as new malware appears. For this reason, many researchers have attempted to develop an automated, smart solution to identify malicious APKs efficiently, providing models that can adapt to such growing and changing environment.

2.3.1. Static and dynamic Android malware analysis

As explained in section 2.1.2, based on the attributes used to analyse the APK sample, analysis can be divided in static and dynamic. Static analysis extracts features from the application without running it; these can include system calls, size of application, permissions, etc. It is less time and resource consuming than dynamic analysis, but in some cases, such as obfuscation of the code of the application (a technique often used by attackers to make malware APKs harder to detect), it doesn’t perform as well. Some examples of static malware analysis are the work from Sahs and Khan[16], who used a SVM to classify whether a sample was malicious using permissions and API calls; Shabtai[17] used permissions as well, but also framework methods and classes to classify if a sample was malicious or not; Yerima et al. [18] used a Bayesian classifier and extracted features such as permissions, API calls and Linux commands to determine if an application was malicious; and finally, Xiaoyan et al.[19] extracted permissions from the APK code, and used a linear SVM in comparison to other classifiers such as RandomForest, Bayes or J48 decision tree to determine if the application was malicious. SVM gave the best results.

Dynamic analysis extracts attributes from the running application, such as network traffic, battery usage, etc. This method needs more resources and time that the static approach, but it is not affected by obfuscation in the code. Some authors like Wei et al.[20], who used the tool *DroidBox* to extract features from the behaviour of the application while running in a sandbox environment, focused on the network behaviour of the malware. They achieved about 93% accuracy comparing different algorithms using data mining open source libraries WEKA and FastICA. Ham and Choi[20] extracted features divided in categories: network, SMS, CPU power usage, process, memory Native and Virtual Memory. They then compared different techniques such as SVM, Naïve Bayes, RandomForest, to determine if an application was malicious, and concluded that SVM gave the best results, obtaining almost 100% accuracy in some cases, although it gave some false positives in benign applications.

There are also hybrid approaches, which use both static and dynamic attributes to determine if an application is malicious. For example, the work of Patel and Buddhadev[22], who extracted features such as API calls and permissions from static analysis and used them in combination with network traffic which was captured with dynamic analysis. They then used a Genetic Algorithm based machine learning technique to create a rule based system. They finally obtained a 96.43% detection rate to detect malicious applications.

2.3.2. AI applied to Android malware analysis

There are several works that apply AI to malware analysis in Android. Due to the adapting ability of AI algorithms, these techniques offer very valuable results for malware analysis.

Most of the tools developed are aimed at being able to differentiate malicious from legit APKs. Pektas et al.[23] use online machine learning in an attempt to detect new Android malware, by extracting a series of attributes using Cuckoo Sandbox environment, which performs a hybrid analysis. They obtained an 89% accuracy using this method.

Sharma and Sahay[24] propose an approach to identify metamorphic malware comparing the performance of different classification algorithms using the tool *WEKA*. They extracted features using static analysis, and tested their model with unknown malware. The highest accuracy obtained was 97.95%, using RandomForest.

Altyeb[25] extracted the permissions from the Android app, and then performed feature selection with information gain algorithm, and finally compared NaiveBayes, RandomForest and J48 to classify Android applications as malware or goodware. The algorithm achieved the highest precision of 89,8% accuracy with lowest false positive rate of 11%.

2.3.3. Evolutionary algorithms in Android malware analysis

Some authors use evolutionary algorithms in the analysis of malware applications in Android. Since evolutionary algorithms are mainly used for optimization, in most cases they are used in combination with some classification technique.

Zubair et al.[26] developed a family malware classification framework based on the network behaviours of the malware samples, and then propose a classification framework based on network behaviour in which they analysed the applicability of various evolutionary (as well as non-evolutionary) algorithms. Their work was focused on malware family classification. They concluded that evolutionary algorithms such as supervised classifier system provided an effective solution for malware family classification.

Firdaus et al.[27] used static analysis to extract features from a series of applications, to then apply statistical and genetic search to select optimal features for various classifiers to detect Android malware. They tested the following classification algorithms: NaiveBayes, Functional Trees, J48, RandomForest and Multilayer Perceptron, obtaining the best results with Functional Trees. Their work is restricted to identifying malicious apps.

2.3.4. Android malware family classification

Although most studies are focused on determining whether an application is malicious or not, some work has been done in classifying the malware samples in their corresponding families, such as the previously mentioned by Zubair et al.[26].

Li et al.[28] created a machine learning based system called DroidADDMiner, which used API data dependence paths to detect, classify (types) and characterize (families) Android malware. The system gave a 98% detection rate, and 96% accuracy when classifying the samples in their families.

Yusoff and Jantan[29] developed a malware classification framework based on malware target and operation behaviour, and used a genetic algorithm to optimize the classification system as well as help in malware prediction. They experimented with a series of classification algorithms; Naïve Bayes, SVM, Decision Tree and KNN.

3. SYSTEM ANALYSIS

In this section, the system is analysed in order to determine all the requirements and conditions that will need to be met in the development. Throughout this section, the problem presented in this document is examined and a series of high level definitions of the system are given, aiming to outline the solution for the problem. The design of the solution will be explained in the next section, **SYSTEM DESIGN**.

The section will first describe the approach taken for the implementation of the system, referring to the choices made regarding analysis and learning possibilities. Later, a series of user requirements that the system shall be compliant with are proposed. The operating environment of the system is then presented. Finally, use cases and traceability matrix are explained.

3.1. Approach

Before designing the system, the developer must take into account that there are several approaches that can be used for analysing Android malware. As explained in **STATE OF THE ART**, the extraction of attributes can be static, dynamic or hybrid. Also, the learning process can be supervised, unsupervised or reinforced.

For the implementation of the work described in this document, a series of malicious APKs were obtained from the malware repository ContagioDump. This APKs are already classified inside of their types and families, and are available for free in said repository. More details about these APKs will be given in the section 5.

The extraction of attributes was performed via **static analysis**. The reason to choose this analysis was the benefits it provides regarding time and resource consumption, as well as the results that it has proven to give as seen in similar work in section 2.

For the learning, since the APKs had already been classified in their types and families, a **supervised learning** approach was used. The learning was performed by a Support Vector Machine, and a Genetic Algorithm was used to optimize the attributes for the learning. The SVM was chosen as algorithm for the classification based on the work examined in section 2.3, where it became clear that many authors concluded this algorithm gave the best results when analysing Android malware. This process is explained in depth in section 5.

3.2. Requirements

Based on the objectives of the work presented here, a series of user requirements will now be defined. These requirements will define the functionalities of the system to implement, with the following format:

- **ID**: Used to identify each requirement. This ID will use the format UR-XXX, where UR stands for “User Requirement”, and is followed by a three digit number starting on 001 and increasing in one for each requirement.
- **Description**: Detailed description of the requirement’s objective.
- **Justification**: Why should the requirement be met; why is it included in the system.

- **Priority**: Each requirement will have either low, medium or high priority, to help to plan the development process.

❖ UR-001
  - Description: design an intelligent system that can classify malware APK samples in types and families of malware.
  - Justification: The goal of the project is to develop and Android malware classification system
  - Priority: High

❖ UR-002
  - Description: The whole project will only use open source tools.
  - Justification: The project should not require software license. Open source code allows for easy future work improvements and later research.
  - Priority: High

❖ UR-003
  - Description: Use a genetic algorithm to optimize the parameters for the learning.
  - Justification: The project aims to explore the performance of genetic algorithms in malware classification, and how they can improve the learning process.
  - Priority: High

❖ UR-004
  - Description: Use an AI classification technique for the learning.
  - Justification: The project's objective is to provide an intelligent scalable solution for Android malware analysis.
  - Priority: High

❖ UR-005
  - Description: The genetic algorithm will have a limit of 1000 learning cycles.
  - Justification: The genetic algorithm needs stopping criteria. In the work proposed, this criterion can only be time or evaluation dependant.
  - Priority: Medium

❖ UR-006
  - Description: Once trained, maximum running time for the system will be limited to 1 minute.

- Justification: Although the training and refining process can take a long time, once a final model is found the system needs to provide a solution given a sample within reasonable time.
- Priority: Medium

❖ UR-007
- Description: The output of the system is a subset of attributes and the trained model.
- Justification: The system will provide an optimal subset of attributes and a trained SVM as a result of training and testing.
- Priority: High

❖ UR-008
- Description: The system will use a total of 1175 malicious APK samples for training and testing.
- Justification: raining data needs to be sufficient.
- Priority: High

❖ UR-009
- Description: The malicious APK samples will belong to the malware types: banker, ransomware, spyware, adware, trojan, exploit.
- Justification: Training data needs to be diverse and representative of the problem.
- Priority: Medium

❖ UR-010
- Description: For each malicious APK belonging to a malware type as described in UR-008, there will be representative samples of at least one family within the malware type.
- Justification: Training data needs to be diverse and representative of the problem.
- Priority: High

❖ UR-011
- Description: All malware samples will be obtained from the free repository ContagioDump.
- Justification: UR-002
- Priority: High

- UR-012
  - Description: The system will be developed entirely in Python programming language and open source libraries.
  - Justification: UR-002
  - Priority: High

- UR-013
  - Description: The attributes extracted from the APKs will be obtained via static analysis.
  - Justification: Static analysis is less time and resource consuming than dynamic analysis.
  - Priority: High

- UR-014
  - Description: The attributes extracted from the APKs will be: for each call to an Android library present in the APK, number of times said call is implemented in the code; permissions asked by the APK; size of the APK.
  - Justification: The attributes proposed are representative of the behaviour of the APK and can be obtained via static analysis.
  - Priority: Medium

- UR-015
  - Description: The training dataset will have a common attribute format for all APKs.
  - Justification: The attributes must exist on all samples in order to learn from them. Since not all APKs have the same usage of Android libraries nor permissions, the chosen format must represent all possibilities.
  - Priority: High

- UR-016
  - Description: Format of training dataset. Dataset will contain a series of rows representing each APK, where each row will be a list of values for:
    - All possible calls to an Android library: value equals number of times the APK makes a certain call
    - All possible permissions: Boolean, True if the APK asks the permission and False if it doesn’t
    - size of the APK: numeric value
    - Class: type and family of the APK)
  - Justification: UR-015

- Priority: High

❖ UR-017

- Description: Codification of the genetic algorithm
  - The genome of the genetic algorithm will describe a subset of the attributes obtained with the static analysis
- Justification: The genetic algorithm must codify a solution for the problem.
- Priority: High

❖ UR-018

- Description: Implementation of the classifier
  - The classifier will be implemented using Python's opensource library Scikit Learn
- Justification: UR-002
- Priority: Medium

❖ UR-019

- Description: Integration genetic algorithm and classifier model
  - Fitness score for the genetic algorithm corresponds to the accuracy score obtained by the classifier when training with the subset of attributes codified in the genome
- Justification: The genetic algorithm is used to optimize the learning process.
- Priority: High

❖ UR-020

- Description: Test dataset
  - A sub dataset with enough samples representative of each family and type of malware will be extracted from the original training dataset and will not be used for training.
  - Said dataset will be used to test the system
- Justification: The system must prove to be compliant with a representative sample of malicious APKs.
- Priority: High

❖ UR-021

- Description: Testing the system

    - The results obtained with the SVM and GA will be tested using the test dataset described in UR-020.
    - A series of tests [\ref evaluation] will be performed to ensure the quality of the system
- Justification: UR-020.
- Priority: High

3.3. Operating Environment

The system has two main modules: static analysis of APKs and intelligent malware classification.

The first one provides the data that is used to train the second. Before the data can be used by the second, it is pre-processed and two datasets are created: training and test. In the second module, a genetic algorithm (GA) selects a series of attributes from the training dataset that are then used by a Support Vector Machine (SVM) to obtain a classification model. The accuracy obtained with the SVM is then fed back to the GA, which uses it as fitness score to evolve. The test dataset will later be used in the evaluation of the system (see **EVALUATION**).

A visual schematic overview of the system design is shown in Figure 12. The symbols used in this schematic are explained in Table 1.

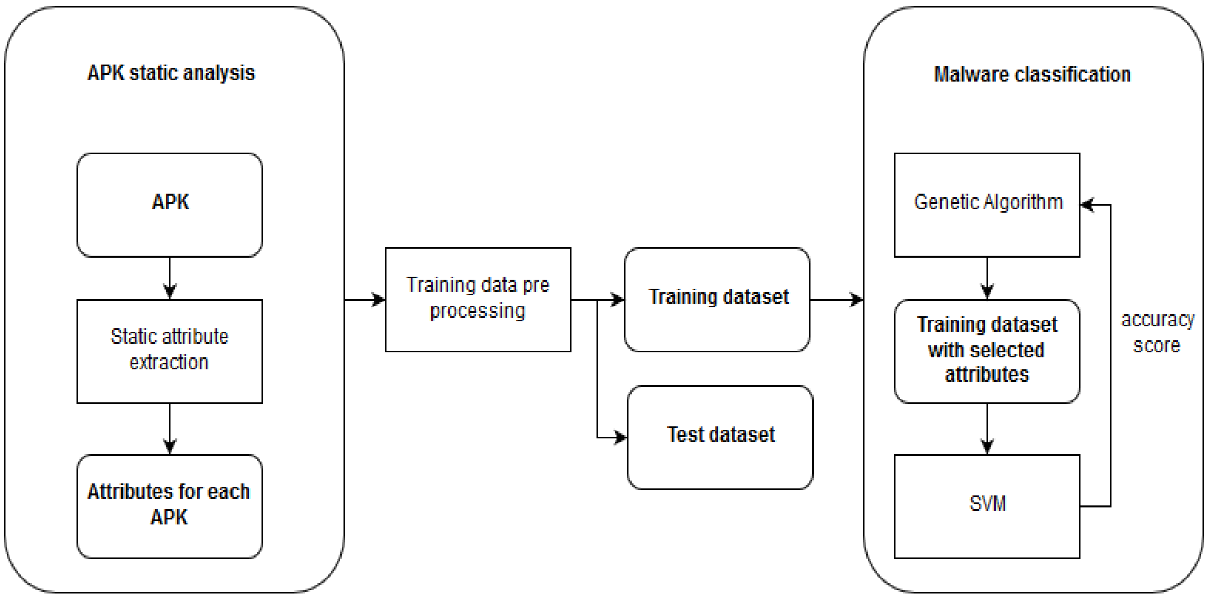


Figure 13: Operating Environment

Symbol	Meaning
▭	Process
▢	File
→	Direction of data flow

Table 1: Operating Environment notation

3.4. Use cases

Use cases define the possible interactions that a user can have with the system. This section is usually included in reports about software engineering projects. However, the work described in this document allows no possible interactions with a user. Therefore, there are no user cases that can be defined.

3.5. Traceability Matrix

Traceability matrix provides an overview of the relationship between user requirements and use cases. Since there are no use cases possible for this project, there will be no traceability matrix shown.

4. SYSTEM DESIGN

This section will explain the design of the system implemented. Firstly, an overview of the design of the system is shown, followed by an explanation of the technologies used in the project is given, followed by a general schematic overview of the architecture of the system using a component diagram. An explanation about the classes in the system, along with a class diagram, are given in the next section. Lastly, this section includes a flowchart showing the behaviour of the system.

4.1. Design overview

The system to implement must be a classifier for malicious APKs, that given a sample classifies it inside its type and family of malware. The attributes will be extracted from each sample via static analysis (see 2.3.1), which consists on analysing the APK's code without running it. Since the number of attributes obtained is too big, once the attributes have been extracted, a genetic algorithm will be used to optimize the attributes used in the classifier.

This optimization will be done by using each individual from the genetic algorithm to codify a subset of attributes, train the classifier with said attributes, and then measure the accuracy of the trained model. The accuracy obtained will be then fed back to the genetic algorithm as fitness score for each individual.

4.2. Technologies

The whole project was developed using open source technologies, and the data was obtained from free samples available on the site ContagioDump. All the code was implemented in Python programming language.

4.2.1. Python

Python is a general purpose programming language which has gained popularity for data science and machine learning implementations over recent years. It was used for all the different modules of the project. The APK analysis was implemented using Python's library *Androguard*, and the classification algorithm was implemented using Python's library *Scikit Learn*.

Androguard

Androguard is a Python tool that allows to interact and work with Android files. It can be used through a CLI or graphical frontend, or as a library inside of own code. For this system, it was used as a library inside the code for APK analysis.

Scikit Learn

Scikit Learn is an open source machine learning library for Python. It integrates several algorithms and tools for different purposes such as classification, regression, clustering, dimensionality reduction, model selection pre-processing. It was used to implement the classifier model (Support Vector Machine).

4.3. System Architecture

In this section, the architecture of the system will be explained using a component diagram. According to UML[30], a component diagram shows components of the system, along with their relationships, interfaces, or ports between them.

4.3.1. Introduction to component diagram

A component is a logical or physical unit that represents a functionality within the system. The idea behind component based design is that if needed, the components can be deployed and re deployed independently. A component is represented as shown on Figure 13:

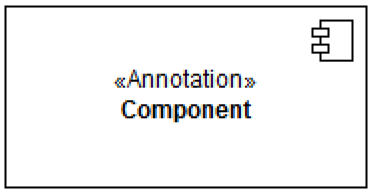


Figure 14: Component notation

Components that work together to achieve the same functionality can be grouped in subsystems. The definition of these subsystems was outlined in section 3. When two components are in the same subsystem, they will be represented as shown on Figure 14:

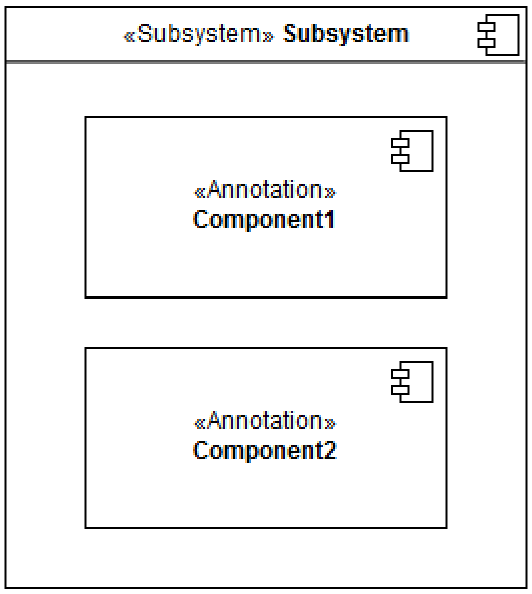


Figure 15: Subsystem notation

Moreover, components can have dependencies between them. This dependency occurs when a component uses a functionality that other component performs. Dependencies are represented as shown on figure 15 below:

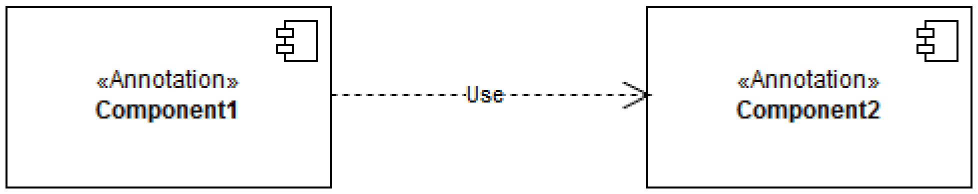


Figure 16: Dependency between components

4.3.2. Component diagram

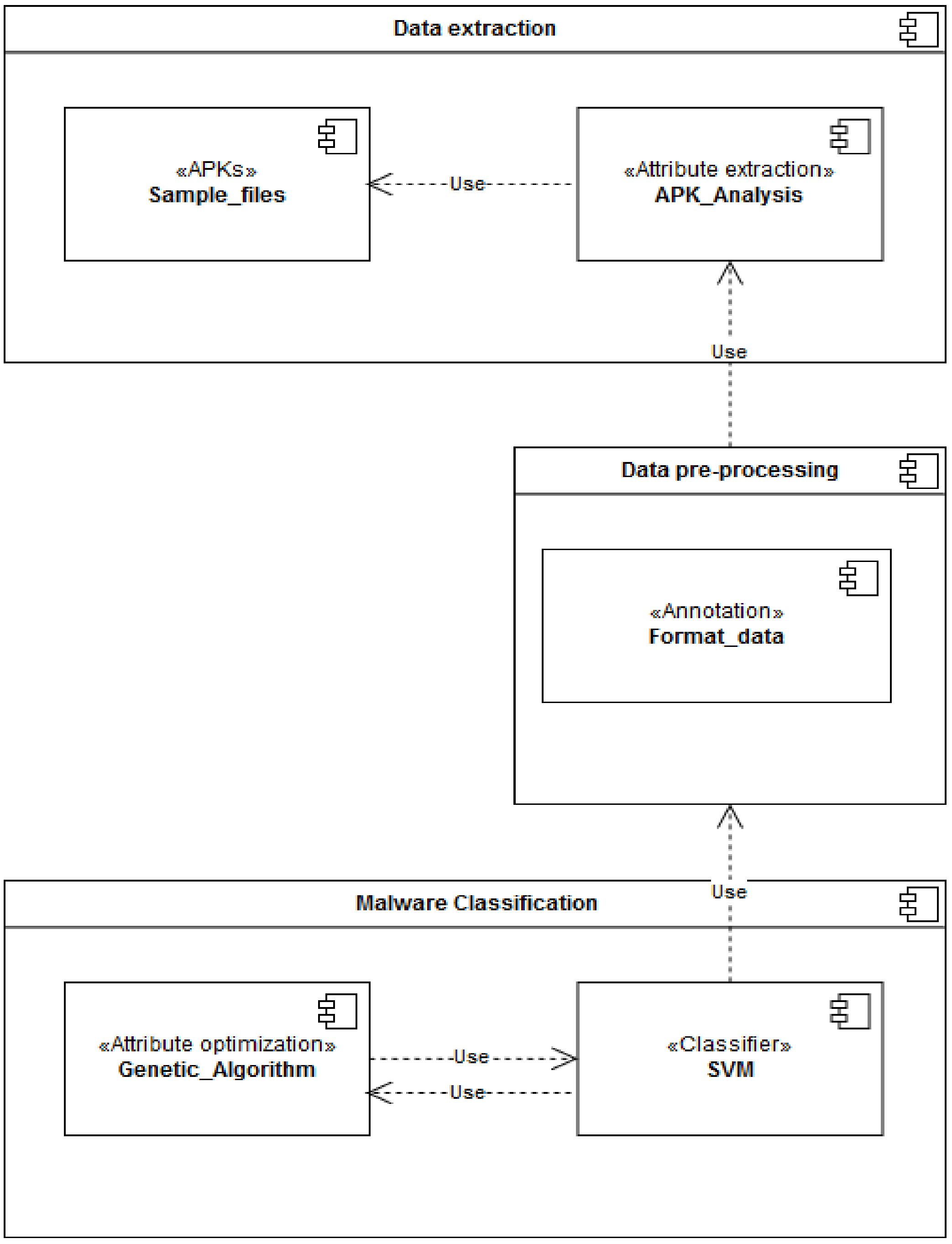


Figure 17: Component diagram

4.4. Classes

This section contains a detailed specification of the classes within the system. These classes are first identified and then, a class diagram with all the classes in the system and relationships between them is provided.

4.4.1. Identification of classes

As explained before, the system has two main functionality units: APK analyser and classification of malware, which uses a genetic algorithm and SVM. Between the two functionalities, a third is added to prepare the data for training.

The classes in the system correspond to these functionalities:

Analyse: Extracts the attributes for the learning from the APK code with static analysis.

PrepareData: Used to format and pre-process the attributes extracted with Analyse and create data for training and testing.

Classifier: Contains the functionality of the SVM used for classifying the samples. It communicates with the Individuals in the population as they define the attributes to use for the learning (therefore the format of the training and test dataset), and calculates the fitness for each individual. It also needs the data created in *PrepareData* to select the relevant rows of information in the file created by said class.

Individual: Each individual is defined by a genome and a fitness value.

Population: It's a list of Individuals.

4.4.2. Class diagram

This section illustrates all the classes in the system, their methods, attributes and the relationships between them. The notation used to describe a class is shown in figure 17:

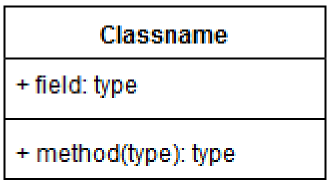


Figure 18: Notation of classes

Where *fields* refer to the attributes of the class, and a method will be described with the data type it receives as parameters and the return value. *Type* is a data type.

The relationship between two classes is shown with an arrow that points from the class using a value or data from another class, to the class that provides said data or value. The arrow is tagged with the method from the former class that requests data from the latter.

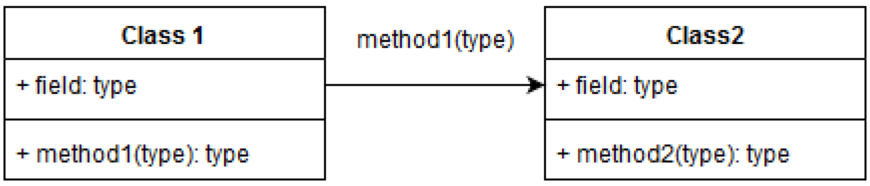


Figure 19: Relationship between clases

Find below an schematic view of all classes in the system and the relationships between them:

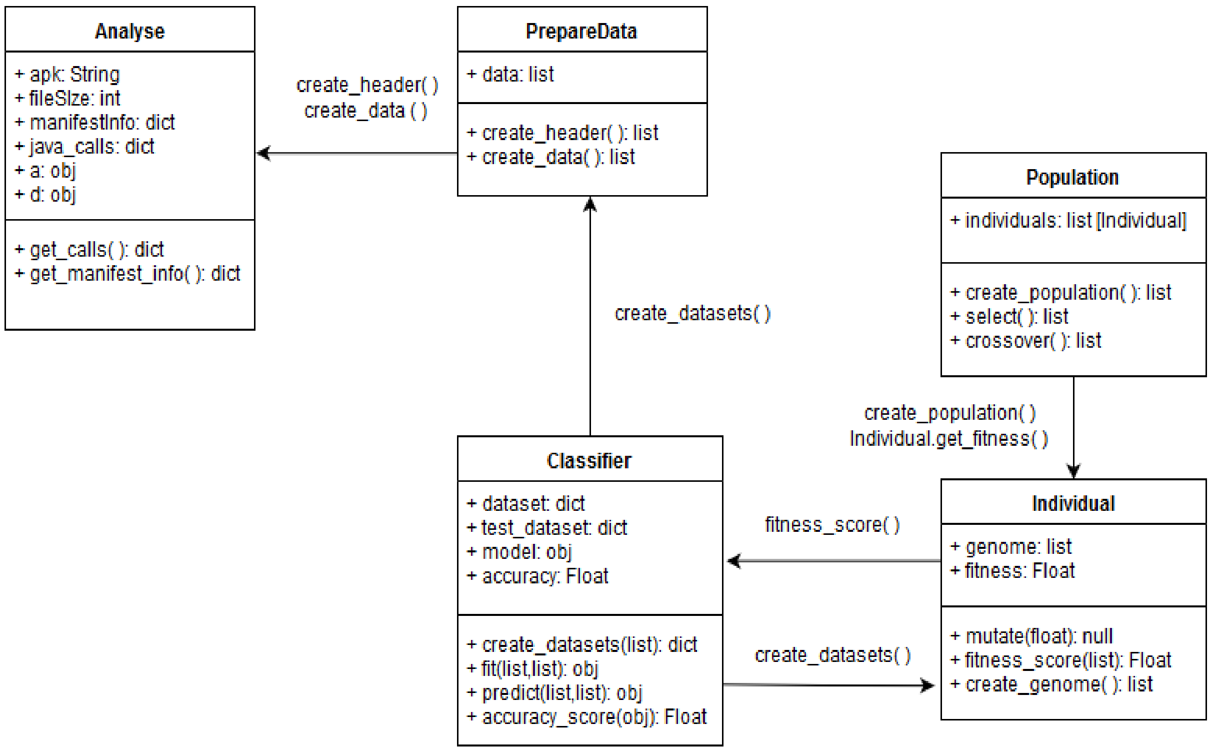


Figure 20: Class diagram

4.5. Flowchart

In this section, a flowchart is used to explain the activities in the system. The symbols used in this chart are explained in table 2:

Symbol	Meaning
●	Start
▭	Process
◇	Decision
◉	Finish
→	Direction of data flow

Table 2: Flowchart notation

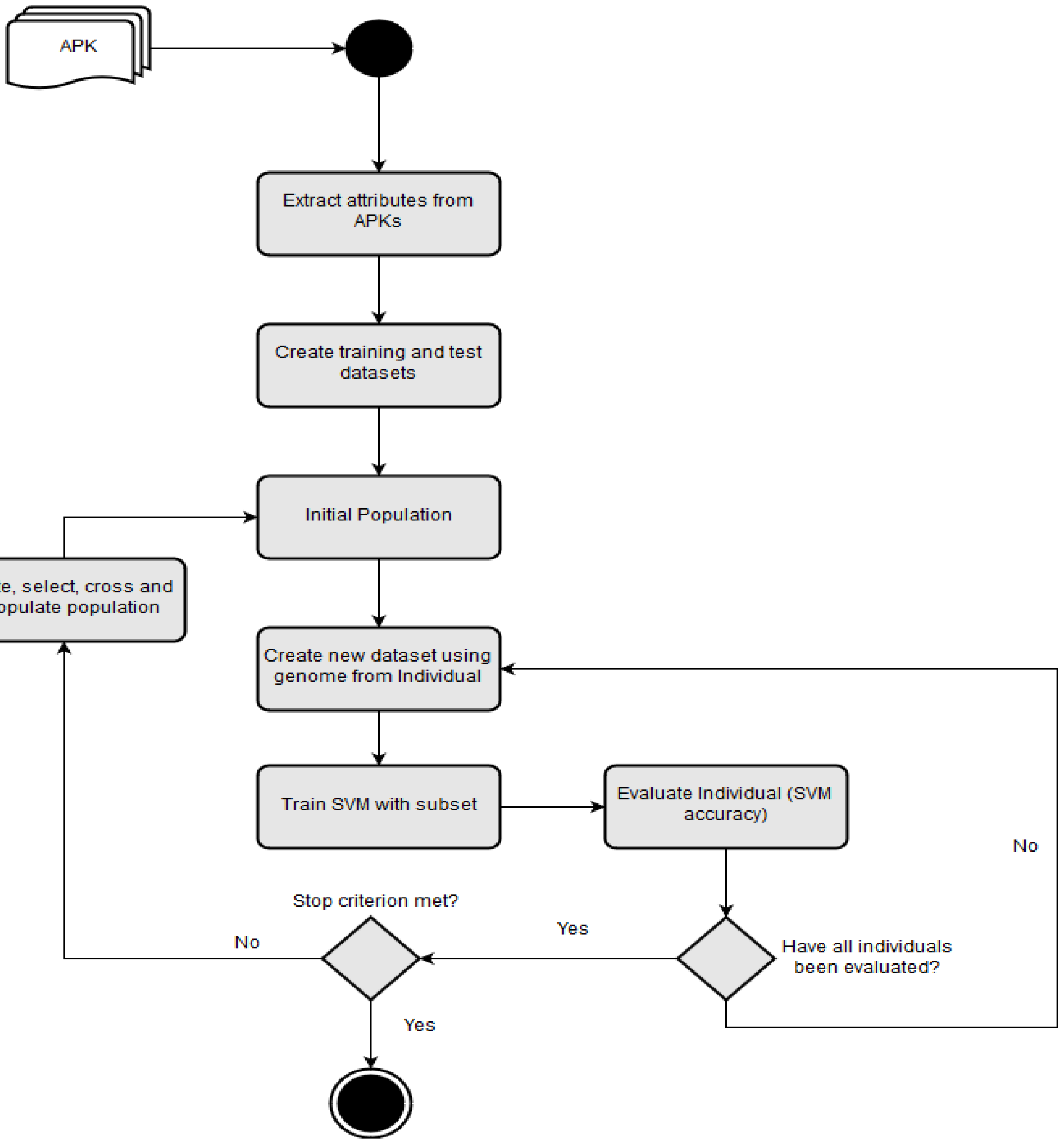

APK
Extract attributes from APKs
Create training and test datasets
Initial Population
Mutate, select, cross and repopulate population
Create new dataset using genome from Individual
Train SVM with subset
Evaluate Individual (SVM accuracy)
No
Stop criterion met?
No
Yes
Have all individuals been evaluated?
Yes

Figure 21: Flowchart

5. IMPLEMENTATION

The system was implemented in three phases: extraction of training data from sample APKs, pre-processing of data to prepare it for learning, and development of the intelligent classification system: Genetic Algorithm and integration with the SVM. For the implementation of the latter, python's library Scikit Learn was used. The two first phases and the genetic algorithm were implemented entirely in Python programming language.

In this section, the implementation of the system will be explained in five sections. First, a process overview is given, explaining the steps followed in the implementation and the functioning of the system.

In the three next sections, the implementation of each of the modules in the system is explained in depth. This includes the static analysis of the APKs, where attributes are extracted; pre-processing of the data obtained in the previous step, in which the data is prepared for learning; and finally, the detailed implementation of the intelligent malware classification system. A detailed explanation of the classes and algorithms implemented will be given for each one of them. Finally, an improvement added to the original implementation is explained.

5.1. Process overview

Find below an overview of steps implemented. This process is explained in detail in the following sections.

1. First, a series of APKs are analysed in order to extract attributes from them. A static analysis is performed, where the following attributes are extracted from the code of the APK: calls to Android libraries (number of times the application executes a certain call, for all the calls in the code), permissions, and size of the APK. This attributes are written to a file specific for each APK. The type and family of the APK is also included in the attribute file.
2. After the attributes for each APK have been extracted, they are combined in one format and pre-processed for the learning. This pre-processing includes randomizing the data to avoid bias in the learning, and splitting the data in two datasets: training (with 70% of the data) and test (with the remaining 30%).
3. The training dataset is used to train the classifier, which is formed by a genetic algorithm (GA) and a Support Vector Machine (SVM). The genetic algorithm selects a subset of attributes that are used for the classification. The SVM is then trained with the training dataset with selected attributes, and the accuracy score from the SVM (obtained with the test dataset with selected attributes) is used to evaluate the GA. The parameters for the SVM are fixed in this step.
4. Step 3 is repeated until the GA meets the stop criterion, which is a set number of cycles (also called generations).
5. When the GA stops, the best subset of attributes is chosen, and a series of experiments are carried out using different parameters for the SVM.
6. Once the best subset of attributes and the trained SVM have been created, a series of tests are performed using a new set of APK samples that weren't used in the process explained before, to prove the efficacy of the system.

5.2. Attribute extraction: APK Static analysis

The first step in the process is analysing the APK files to extract the attributes for learning via static analysis; as explained in the **STATE OF THE ART**, static analysis is performed by extracting attributes from the code of the APK, instead of capturing features from APK while it's running. The attributes are written to a CSV output file for each of the APKs. As stated before, the attributes extracted are: calls to Android libraries, permissions, size of APK.

The analysis was performed by looking at the DEX attribute of the APK (see 2.1.1), which contains the assembly code of the application. The assembly code is then read and every time there is a call to the Android library, a specific counter for the call is created. This counter is incremented each time the call is found in the code. The permissions can be obtained directly using Androguard library, which provides an interface with the permissions of the application. Lastly, the size of the application can be obtained by using a library from the OS.

These attributes were chosen because, as seen on similar work (**Static and dynamic Android malware analysis**) many researchers have obtained good results using static analysis. Furthermore, static analysis provides a less resource consuming attribute extraction technique compared to dynamic analysis. Calls, permissions and size of the application were attributes used by other researchers in similar work and could be extracted via static analysis.

5.2.1. Constructor

This method initialises the object Analyse with the information from the APK it receives as parameter. It uses the library *Androguard* to parse the application's code in order to initialise the attributes *a* and *d*, which contain the information about the APK and DEX objects. As explained in the **STATE OF THE ART**, APK is the format of Android application, and *Dex* is the compiled code of the apk, which contains all its functionality.

After these two attributes have been initialised, they are used by *get_calls( )* and *get_manifest_info( )*, as explained below, to define the other attributes of the object Analyse.

This method also obtains the size of the application, which is directly accessible using libraries from the OS.

5.2.2. Get_calls

This method returns the number of times the Android library is called by the application. It does not take into account the calls to methods within the application, only those to methods from the Android library.

The DEX (*d*) attribute contains the assembly code (in Android, called "*smali*") for the application. Get_calls iterates through this code and counts the number of times a call to the Android library appears in the code, and then returns the call and the number of times it appears.

5.2.3. Get_manifest_info

This method obtains the permissions, which are stored in the attribute APK (a). *Androguard* provides an interface that can be used to access the list of permissions directly:

```
permissions = a.get_permissions()
```

It then returns the permissions list.

After executing this program, there will be a CSV file per APK sample with the attributes that define said sample, with the following format:

```
name of call, number of times the call is made
name of call, number of times the call is made
…
permission
permission
…
size
```

Figure 22: Format of file with attributes extracted from APK

5.3. Data pre-processing

The data obtained in the previous step is processed and converted to a common format that can be used for learning; all APKs must be defined by a common set of attributes (which is defined in the header) and a class. Each APK is defined with a list with all the values for each attribute and the value for the class. The output of the class is a CSV file that contains the name of each attribute and the word "class" in the first row, and the corresponding values for each APK in the following rows. It is also randomized to avoid bias in the learning and split in training and test datasets, as explained later. It is interesting to point out that more than 44000 attributes that describe each APK were obtained, as this will affect the results obtained in the tests (as explained in section 6).

The training and test datasets are used to train the classifier model and measure its accuracy to be used as the GA's fitness, respectively. As explained in section **EVALUATION**, once the GA and SVM have been trained, and both a subset of attributes and a trained SVM are available, another set of tests is performed to measure the efficacy of the system, with a new set of APK samples that had not been used until then.

5.3.1. Create_header

The header must contain all possible Android library calls and permissions a sample can have. There are two options to achieve this: either create a header with every possible call to an Android library and permission in the Android operating system, or extract all calls and permissions present in the attributes for each APK analysed.

The first option has two main problems: first, the number of possible calls in Android will probably be very high, and as a consequence a header with an elevate amount of attributes would be created, although most of them aren't found in the samples presented. Second, it would mean incrementing the size of the training dataset file, which might slow down the learning significantly.

The second option only takes into account the calls and permissions found in the APKs previously analysed. This means that if the system is used to analyse a new application that has a call or permission that wasn't present in the APKs used to create the dataset, it will be ignored.

Although this might seem like a problem, the Genetic Algorithm will be used to filter attributes, so not all attributes will be present in the final model in any case. Also, if there is a call that is not present in any of the previously analysed APKs, it is highly unlikely that it will be relevant for the learning and therefore will probably not be present in the final subset of attributes selected for the classifier either way.

In summary, when creating the header, there is no good reason to include calls or permissions that aren't present in the APK set used in this project, thus the second option was chosen.

As explained before, the files created as the result of analysing APKs are a series of CSV files with the following format:

```
[Attribute (call || permission || size), value]
```

The `create_header( )` method works as explained in the pseudo code below:

```
loop: read all files with results of analysing apks
      loop: read all rows in file
             if element in row is not present in attribute list:
                 add element to attribute list
add “class” to attribute list
return attribute list
```

Figure 23: Pseudo code for create_header

The list of attributes is returned as a list object, with all the calls, permissions, size and the word “class”. More than **44000 attributes** were obtained.

5.3.2. Create_data

Once the header has been created, the program loops through all the result files again to write the values that each APK has for all attributes.

The value for all the Android library calls is an integer value representing the number of times that the application includes said call in its code. The value for permissions is 0 or 1, depending on whether the application asks for the specific permission or not. The value for size is an integer value. Finally, the class is a string that contains the name of the type and family of malware the APK belongs to.

The class includes both type and family information to allow different experiments in later evaluation: classification as a type of malware and in more detail, as a family.

The data is stored as a list of lists, where the first sub-list contains all attributes; and the following sub-lists contain the values for each attribute for all APKs and their class. Once added, the sub-lists are shuffled randomly to avoid bias in the learning. This list will later be written to a CSV file, where the first row will be the attributes and following rows will be the values for all APKs.

Note that Scikit Learn will later need the data in form of a Python dictionary, which might lead to question why is the data stored in the form of a list and then written as plain rows on a CSV file. This decision is based on future scalability of the solution; CSV format allows for easy integration with other tools such as *WEKA*, an open source data mining tool which can operate directly on CSV files.

The method works as follows:

```
add header to data[] # will store all values for all APKs
loop: read all files with results of analysing apks
```

```
      create list[] with size = header # will store values for each APK
      set all values of list[] to 0
      loop: read all rows in file
      # each row is a list: [attribute name, value]
             loop: read all elements in header and their index
                 if element in row == element in header
                     list[index] = row[1] #value
      append class name to list
      append list[] to data[]
random.shuffle lists in data[] #to avoid bias in the data
training_dataset = data[:0.70*len(data)] #70% for training
test_dataset = data[0.70*len(data):] #30% remaining for test
write training_dataset, test_dataset to csv
```

Figure 24: Pseudo code for create_data

The data list will then look as follows:

```
data = [
[attribute1, attribute2, attribute3,...,attributeN, class],
[value1, value2, value3,…valueN, class],..., [...]
]
```

Figure 25: Format of training data

Where there would be one list for the header and a list of values for each APK. A simplified example is given below:

```
data = [
[init(),      pow(),      random(),      ConcurrentMap.->putIfAbsent(),
ConcurrentLinkedQueue.-> <init>(), class ],
[423,1,1,2,29, spyware_tizi],
[3,1,1,3,1,1, banker_sberbank] ]
```

Figure 26: Example of training data

5.4. Intelligent malware classification

Once the dataset is ready for the learning, the process to attempt to create an intelligent classifier begins.

Since the classifier model needs to be reading the training dataset for each individual of the population that it evaluates (using only the attributes selected by the genome of the individual), and the data for learning is stored in a CSV file created in the previous step, a copy of the file is created and stored in a temporary buffer at the beginning of the process to avoid multiple OS calls to open the CSV file.

In this section, the genetic algorithm is explained in the first place, followed by the classifier model (including the SVM, which was implemented using *Scikit Learn*), and finally the integration between the two models on implementation level.

5.4.1. Genetic Algorithm

The genetic algorithm is the most complex component of the system. To allow a better understanding of the implementation, first the general definition of the algorithm (codification, fitness and operators) is given, followed by a definition of the specific implementation (including pseudo code) of all the functions of the two classes that make up the genetic algorithm: *Individual* and *Population* (which is a set of *Individual* objects). Since the functionalities of the genetic algorithm are divided in these two classes, each class implements the functionalities of the algorithm that concern either each individual or the whole population.

Definition of algorithm

A genetic algorithm codifies a population of individuals that represent solutions to a certain problem. The individual is defined by the following characteristics: first, the codification of the genome. The genome must be a binary array that codifies a solution to the problem, and it must be possible to codify all solutions in this binary array. Secondly, a fitness function; an defined evaluation function that can measure if the solution codified in the genome is good or bad.

Apart from its definition, the algorithm needs a series of genetic operators that allow a population to evolve, so that new better solutions can be found. These operators are selection, where a subset of the most fit individuals are selected; these individuals' genomes are then combined to create new individuals. This is known as crossover. Lastly, individuals can suffer mutations, which are random changes to their genomes. The mutation has a probability of happening on each generation, which is defined as part of the algorithm. Several mutation rates can be tested to ensure the best performance of the system.

Codification

Each individual is codified as a binary array with N positions where N is equal to the number of attributes obtained in the static analysis of each sample (see **Attribute extraction: APK Static analysis**). Each position in the array corresponds to a feature (which will be either a calls to an Android library, permission, or size of the APK). If the position is set to 1, the feature will be used for the learning. If, however, the position is set to 0, the feature is ignored in the learning.

An example of this codification is given below. Given the following training set with only 5 features as shown before:

```
data = [
[init(),   pow(),   random(),   putIfAbsent(),   ConcurrentLinkedQueue.->
<init>(), class ],
[423,1,1,2,29, spyware_tizi],
[3,1,1,3,1,1, banker_sberbank] ]
```

Suppose a population with two individuals as it follows:

```
            [0,0,0,1,1]                    [0,1,1,0,1]
```

The first individual defines the attributes `ConcurrentMap.->putIfAbsent()`, `ConcurrentLinkedQueue.-> <init>()` to be used for learning, whereas the second codifies the attributes: `pow(), random()`, `ConcurrentLinkedQueue.-> <init>()`.

Fitness

The fitness of each individual is defined as the accuracy percentage obtained when training the SVM with the attributes codified in the individual's genome. This is calculated using Scikit Learn's function *model.preditc( )*, in the codification of the classifier model, and will be detailed later in section **Classifier: Support Vector Machine**.

Genetic operators

Each genetic operator can be implemented in many different ways, which affect the evolution process. In this section, the implementation chosen for each operator is explained in depth.

SELECTION

20% of the population is selected to be in the mating pool. The mating pool is a list where the selected individuals are stored. The selection is done via tournaments: individuals are chosen two by two and their fitness are compared. The ones with highest fitness are selected.

CROSSOVER AND REPLACEMENT

The crossover is the process of combining two or more individuals' genomes to create new individuals. The crossover implemented is uniform crossover with a 50% chance: it takes two parent individuals and uses one gene from each to create two new individuals. For each gene in the child's genome, the parent gene is chosen randomly between the two parents. The replacement is made based on age of individuals: an old individual is removed each time a new individual is added. This is implemented by treating the population as a queue.

MUTATION
The mutation operator switches the value of one gene chosen randomly to the opposite value (0 if the gene was 1 and vice versa).

Individual

The functionalities that belong specifically to the individual are: its codification (or genome), fitness, and the mutation applied to its genome.

Create_genome
The genome of an individual is created as an array with size equal to the number of attributes and all its positions equal to 0. The size is obtained using Python's *len( )* function on the first row of the training dataset. After creating it, the method iterates through its positions and randomly changes some of them to 1:

```
size = len(training_data[0])
self.genome = [0 for x in range (size)] #all genes initially 0
loop through the genome in range (size):
      set random position to 1
      generate new random position
```

Figure 27: Pseudo code for create_genome

Fitness_score
The fitness score is calculated in the classifier model. The individual class calls the getter for the classifier class:

```
self.fitness_score = classifier.accuracy_score(individual)
```

Figure 28: Pseudo code for fitness_score

Mutate
The mutation operator is applied with a certain probability - which is specified as a parameter of the program - on each individual in the population. When the probability is met, a random gene is switched:

```
for individual in population:
     if random.random( ) < probability
     index = randint(0, size-1) #random gene
       swap the value of genome[index]
```

Figure 29: Pseudo code for mutate

Population

The population class implements the select and crossover functions, since they affect a set of individuals.

Create_population

The size of the population is specified as a parameter of the program. The method creates as many random individuals as the size specified:

```
self.individuals = []
for i in range (population_size):
    new_ind = Individual()
    new_ind.create_genome()
    new_ind.fitness_score()
    self.individuals.append(new_ind)
```

Figure 30: Pseudo code for create_population

Select

As explained before, the top 20% of the population is selected to be in the mating pool for later combination of genomes. The selection is done via tournaments: two individuals are selected and their fitness and compared. The one with the best fitness is added to the mating pool, unless it is already present in the mating pool; in that case, two new individuals are selected and compared.

```
mating_pool = []
for i in range(population_size*0.2*2): #top 20%, compared 2 by 2
    select two random distinct individuals
    compare their fitness
    if the one with best fitness is not already in mating pool:
     append the one with best fitness to mating pool
return mating_pool
```

Figure 31: pseudo code for selection

Crossover and replacement

Crossover function implements a uniform crossover with 50% chance: Two new individuals are created randomly combining the genes from two parents. The parents are chosen from the mating pool created in the previous step. For each gene on each child, one of the parent's genes in the same position is chosen randomly.

```
while mating_pool:
    first_parent = mating_pool.pop()
    second_parent = mating_pool.pop()
    first_child, second_child = Individual()
    for i in range len(first_parent.get_genome()):
        if randint(0,1) == 0
          first_child.genome[i] = first_parent.genome[i]
          second_child.genome[i] = second_parent.genome[i]
        else:
          first_child.genome[i] = second_parent.genome[i]
          second_child.genome[i] = first_parent.genome[i]
```

Figure 32: Pseudo code for crossover

The replacement of old individuals is then made by removing two individuals from the top of the population, and appending the two newly created individuals.

5.4.2. Classifier: Support Vector Machine

The classifier model was implemented using *Scikit Learn* library for Python. It uses the *training_dataset* and *test_dataset* created in the data pre-processing step to train and measure the accuracy of the model. It is called by the genetic algorithm step, which provides the genome of the Individual to calculate its fitness. For more information about the implementation using *Scikit Learn*, it is recommended to refer to the official documentation. The SVM model used is a C-SVM, which uses a parameter C to penalize the error in classification. The parameters this model receives are:

- **Gamma** value: this is related to the function used to calculate the distance between samples, which is a Gaussian function. In simple terms, a small gamma value will classify two points as belonging to the same class even if they are far apart.
- **C**: this value is used to penalize the error in classification. With higher values of C, space between classes is reduced (the margin between classes is smaller, therefore the division of classes is less clear). On the contrary, smaller values of C give priority to creating a bigger gap between different classes. If C is too small, there will probably be some misclassified samples.

The parameters of the SVM are fixed to Gamma = 0.001, and C=100 during the attribute optimization process (evolution of the algorithm). There was some later experimentation done with the SVM, as explained in section 6.3.

Create_datasets

This method filters the data in the datasets that is codified in the Individual’s genome. The training and test data are obtained from the CSVs obtained before, as explained in section **Data pre-processing**. For each individual, it filters the training and test datasets with the attributes codified in the individual.

This method works as explained in the pseudocode below:

```
store data in CSVs in buffer #aux list
dataset, test_dataset = {‘data’:[], ‘target’:[]}
for row in training_data_aux[:-1]: #last elem is target
    for elem,i in enumerate(row):
        if individual[i]==1
             dataset[‘data’].append(training_data_aux[i])
dataset[‘target’].append(training_data_aux[-1]) #last elem is target
repeat same process for test data
return dataset, test_dataset
```

Figure 33: Pseudo code for create_datasets

Fit

This function trains an SVM with the training dataset created in *create_datasets( )*. To do this, it uses Scikit Learn SVM model:

```
create SVM
model = fit SVM to training dataset
return model
```

Figure 34: Pseudo code for fit

Predict

Once the model has been trained, the model is fed the test_dataset to measure its performance:

```
prediction = model.predict(test_dataset[‘data’])
accuracy = accuracy_score(predictions, test_dataset[‘target’])
return accuracy
```

Figure 35: Pseudo code for predict

Accuracy_score

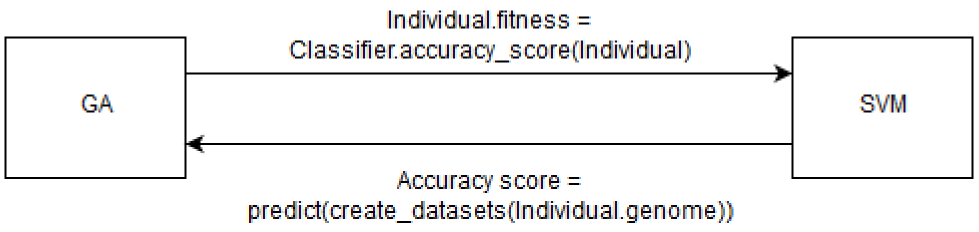
It’s the first call that the class receives. The class Individual calls this method to set the fitness of an individual. This method then calls the other three in the class:

```
training, test = create_datasets(individual)
model = fit(training)
return predict(model, test)
```

Figure 36: Pseudo code for accuracy_score

5.4.3. Integration of GA and SVM

The genetic algorithm and Support Vector Machine communicate through the fitness of individuals. The genetic algorithm requests the calculation of the fitness by the SVM, which needs the codification of the GA to create the model and measure its accuracy. The two models use each other's interface to access the desired values:


Individual.fitness =
Classifier.accuracy_score(Individual)
GA
SVM
Accuracy score =
predict(create_datasets(Individual.genome))

The SVM trains with the training dataset, filtering the attributes that are codified in the individual's genome. The accuracy is then measured by using the trained model to predict the test dataset, and is fed back to the genetic algorithm as fitness.

5.5. Improvements made to original implementation

After implementing the system explained above and testing its performance, the results were suboptimal. Each cycle of the genetic algorithm took around 1,5 minutes to run. The main cause identified was the size of training data elements; each sample was defined by more than 44000 attributes, and since each individual had a random number of attributes, evaluating each one of them implied training the model with an average of 20000 attributes per individual.

Note that this means the algorithm is searching for a solution amongst approximately 2^{44000} possible solutions. The elevated size of the search space can affect the learning significantly, and, as shown in section 6.5, it prevents the correct evolution of the algorithm.

An option that was considered was performing some kind of attribute filtering prior to the genetic algorithm. The first option that was proposed was removing strongly tied attributes, by calculating a covariance matrix and removing those attributes that had a significant statistical similarity. However, this solution wouldn't affect the size of the attribute pool, leaving it with very similar number of attributes, still around 44000.

Another option was then using Scikit Learn's feature selection libraries. These implement a series of machine learning algorithms, such as decision trees, that can determine the most significant attributes for the learning. This solution was not considered for two reasons: first, it is unclear whether it could significantly reduce the number of attributes. Second, it would be performing the task that has been made for the genetic algorithm in this work, hence defeating the original purpose of the GA. Future implementations (see 9.2) could consider using these techniques alone or in combination with others to reduce the attribute pool.

For these reasons, an improvement was made to the original definition of the genetic algorithm:

- The **codification of the genome** of an individual is changed. The new genome has a restricted number of 1's in its codification: there can be no more than 2000 1's in the binary array (which corresponds to a maximum of 2000 attributes used for learning).

This improvement could imply a bias on the learning of the algorithm, since the first 2000 positions are more likely to be in the codification of the algorithm. Moreover, this bias could be strengthen in the crossover, by combining similar individuals whose codifications concentrate the majority of 1's in the early positions of the genome. Nonetheless, it was introduced and tested. Note that 2000 attributes still implies a search space of 2^{2000} elements, which is still large, and can still present the same problems in the evolution of the algorithm.

The performance of this new implementation was proven with a series of tests that are described in section **EVALUATION**. The tests made on the original implementation are also explained in said section.

6. EVALUATION

In this section, the evaluations done to evaluate the best model are explained first, and a real-life simulation test is explained second. The real life simulation was made using new samples as explained below:

Once the final model was chosen (attributes to use in the learning and trained Support Vector Machine), a series of samples were downloaded from *ContagioDump* repository. 10 samples from each family used in the implementation were downloaded (a total of 120 samples), and the model obtained before was applied to test its performance.

6.1. Description of the experimental environment

This section explains the environment in which the tests were performed. To ensure the quality of the tests, the test set has to be sufficient and representative of the problem.

1175 samples were used for the learning of the system. These were samples evenly distributed between the following types and families of malware: Banker (Sberbank, Overlay, Overlaylocker), Ransomware (Xbot), Spyware (BeaverGangCounter, redDrop, Tizi) , Adware (Judy, Hummingbad), Exploit (Godless), Trojan (Marcher, Triada).

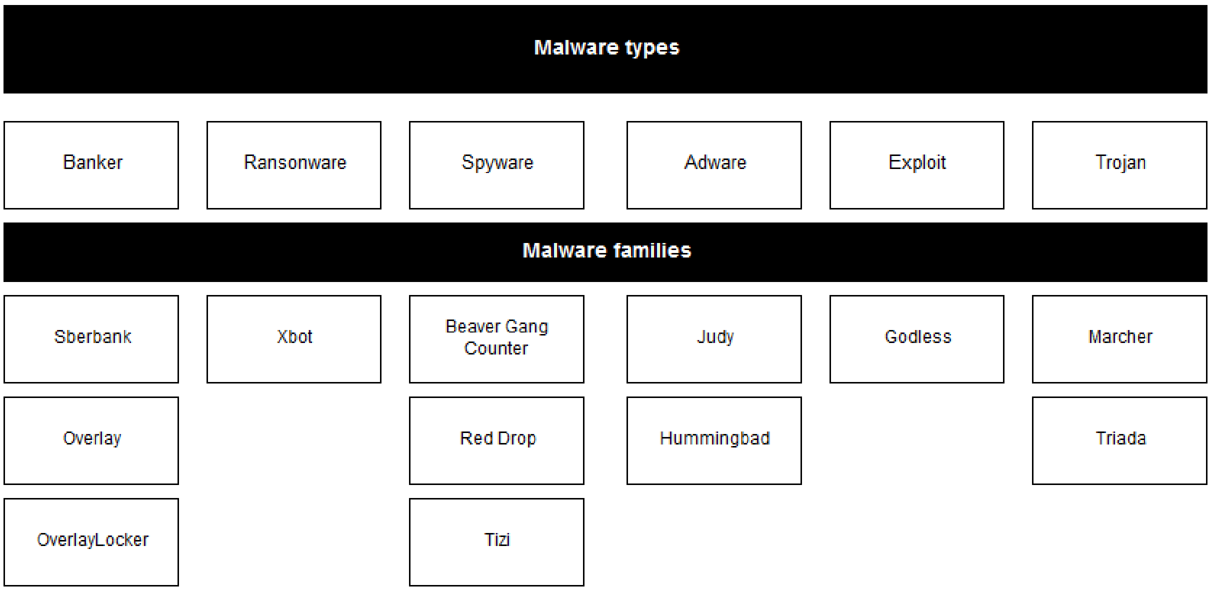


Figure 37: Malware samples used in the system

There are two test sets to take into account: first, the test set used to evaluate the performance of the genetic algorithm and SVM in different models, to choose the best subset of training attributes. This test was obtained as the 30% of the training data. The data was randomly shuffled to avoid any bias, and then 30% was saved to evaluate the model.

After these evaluations have been made, the model needs to be tested in a simulation of a real life situation. For that purpose, a new set of malicious APKs were downloaded. 10 samples for each type and family used for the learning were downloaded (a total of 120 samples). These samples were then tested with the final model (codification of best individual and best parameters of SVM).

6.2. Evaluation of the genetic algorithm

The tests performed to choose the best model were performed using the test dataset created in the data pre-processing step of the implementation (see 5.3). Each test was performed with different mutation rates and fixed values for the SVM parameters. The reason why the tests were performed this way was to obtain the best combination of attributes (best individual), to later test different values for the SVM parameters with the best individual. The size of the population (number of individuals) was set to 20.

The graph below illustrates the evolution of the population for each test in 20-30 cycles of learning (generations). The tests done to the algorithm included between 200 and 300 cycles of learning, however in most cases it converged to a value in about 15 generations, thus only 20 or 30 generations are shown in the figures below. The average running time per cycle (or generation) was 90s.

The objective of this section was to find the best combination of attributes for the classification of malicious APKs. A minimum accuracy score of 90% is required in order to consider the subset of attributes fit enough to be chosen as best subset.

- Mutation rate = 10%

The algorithm took about 27-28 generations to stabilize, and did so in an average accuracy of 79 %. The best individual had an 80.4% fitness score.

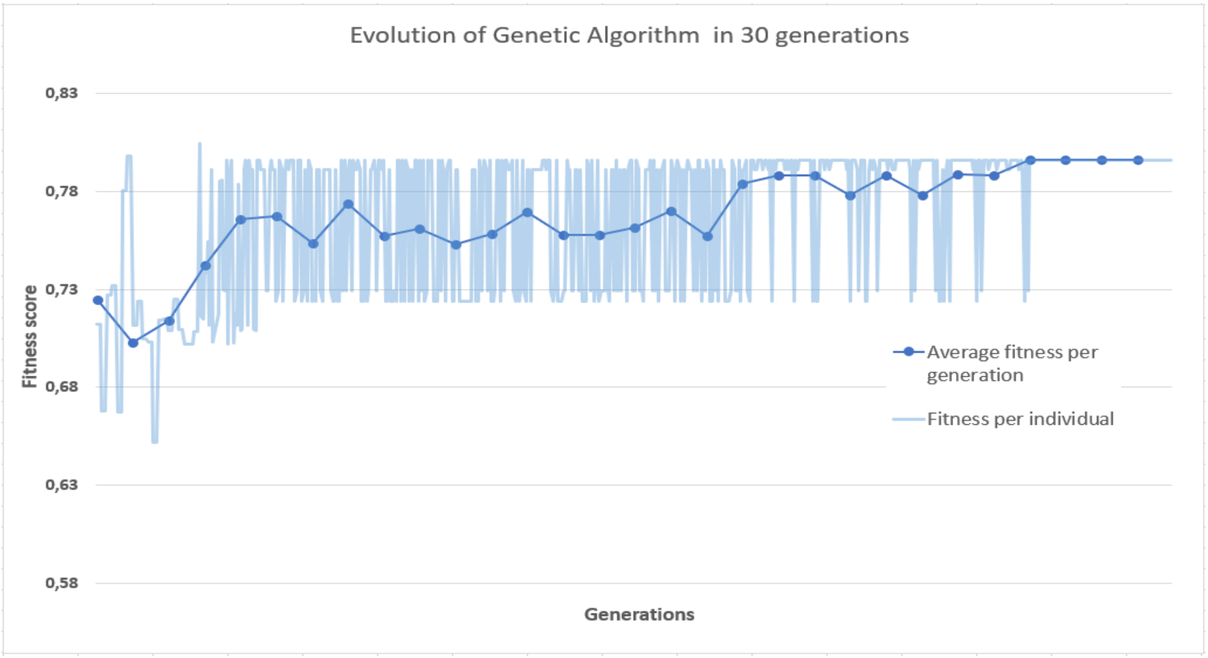


Figure 38: Evolution of genetic algorithm, 10% mutation rate

- Mutation rate = 20%

Increasing the mutation rate, the algorithm converges to an approximate 81% accuracy in 12 generations. The best individual had a total of 84.4% accuracy score.

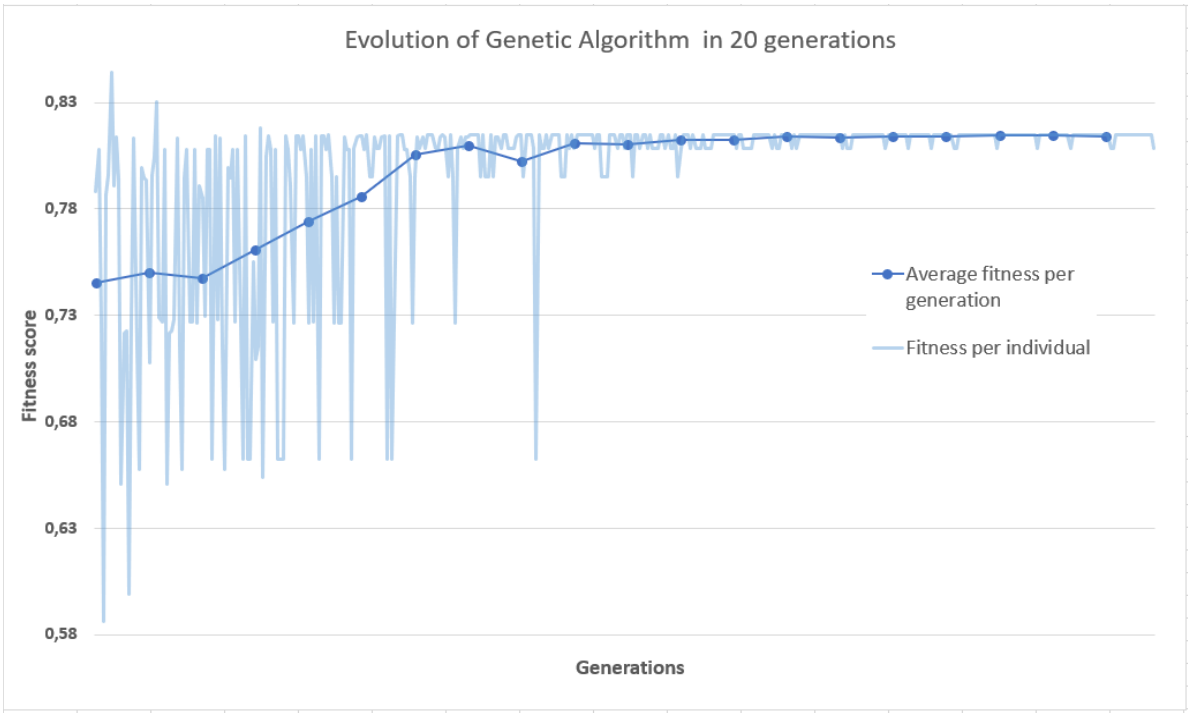


Figure 39: Evolution of genetic algorithm, 20% mutation rate

- Mutation rate = 50%

The algorithm evolves rapidly, stabilizing around 84% fitness score in about 10 generations. The best individual had an 84.4% fitness score.

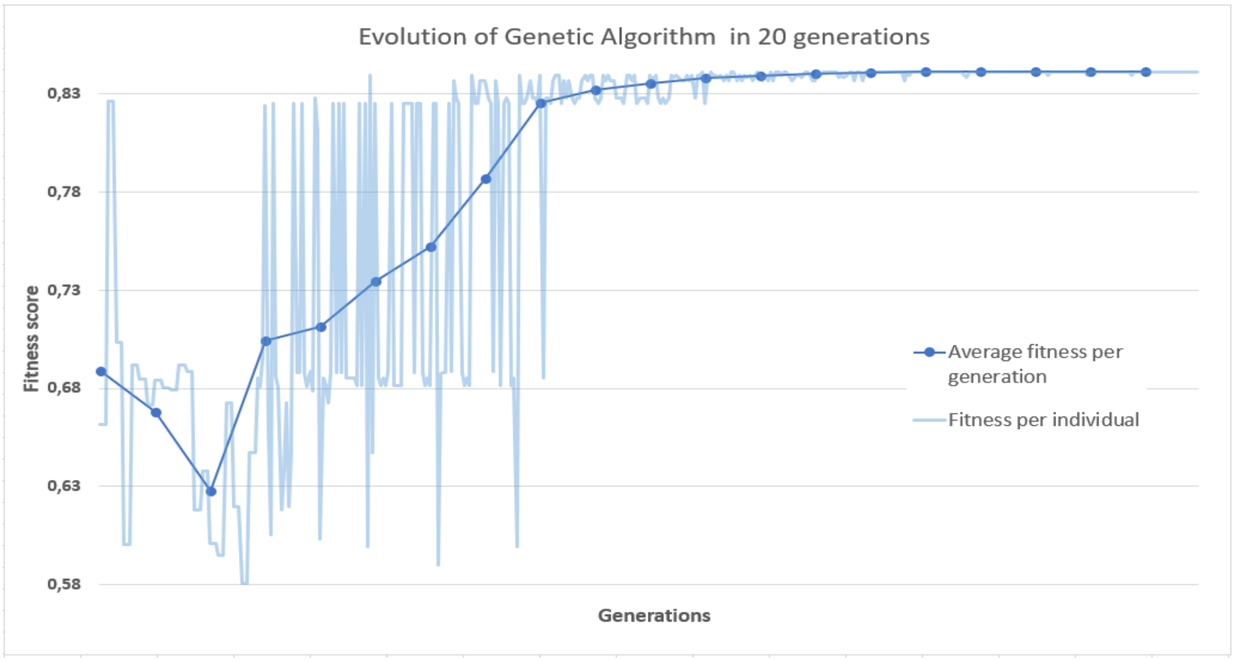


Figure 40: Evolution of genetic algorithm, 50% mutation rate

6.2.1. Tests done with improved version of original implementation

After performing the tests described before, an improvement was added to the genetic to reduce the running time of the model (described in section 5.5). This new model was tested following a similar structure to the test performed in the previous model; different mutation rates, fixed parameters for the SVM. The accuracy scores obtained were very similar, but the running time was reduced significantly, more than half of the original running time. The average time per cycle with the improved version was 42s.

- Mutation rate = 10%

The evolution of the population in this experiment was slow and hardly noticeable, since the first population had an initial average fitness of 91.3%. The algorithm converged to an approximate 93% average fitness, and the best individual had a fitness score of 93.75%.

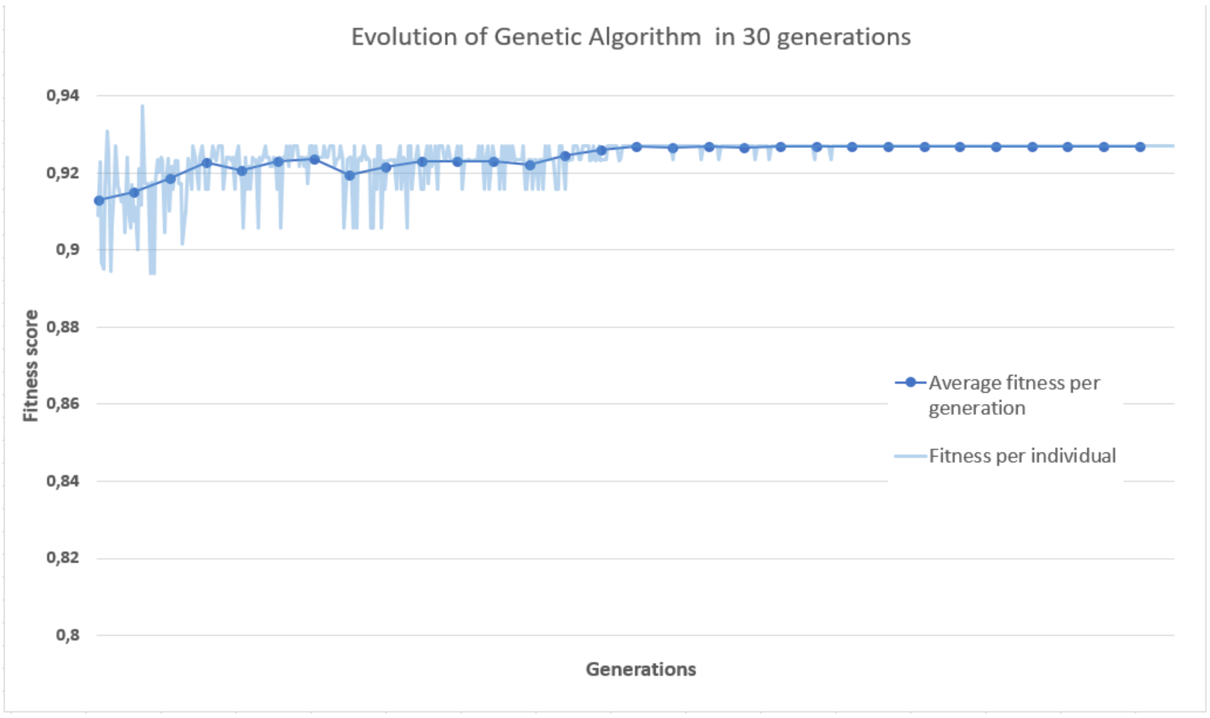


Figure 41: Evolution of improved genetic algorithm, 10% mutation rate

- Mutation rate = 20%

The algorithm converged to an approximate 77.5% in 10 generations. The first generation had an average fitness of 65%, and the best individual had a fitness score of 78.2%.

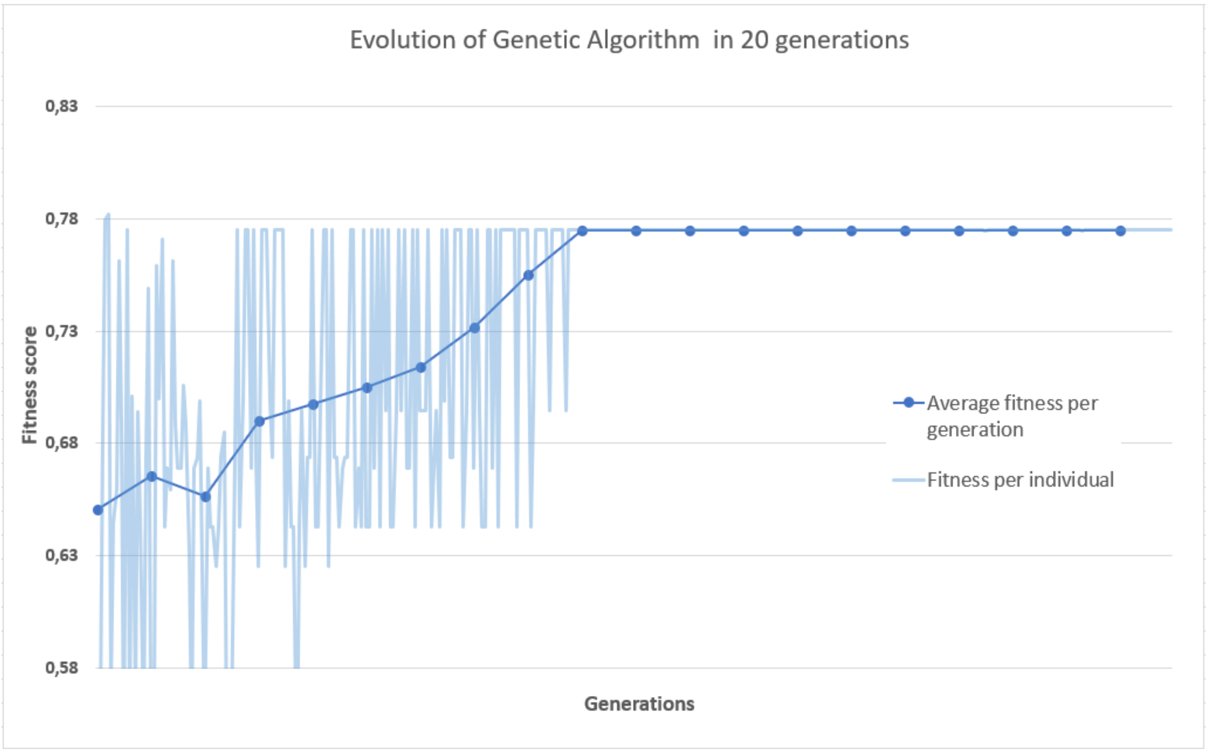


Figure 42: Evolution of improved genetic algorithm, 20% mutation rate

- Mutation rate = 50%

The best individual had a total of 94.37% fitness score. Similarly to the first experiment with this version of the genetic algorithm, the initial population had a high average fitness score (92.1%), therefore the evolution is less pronounced compared to other experiments.

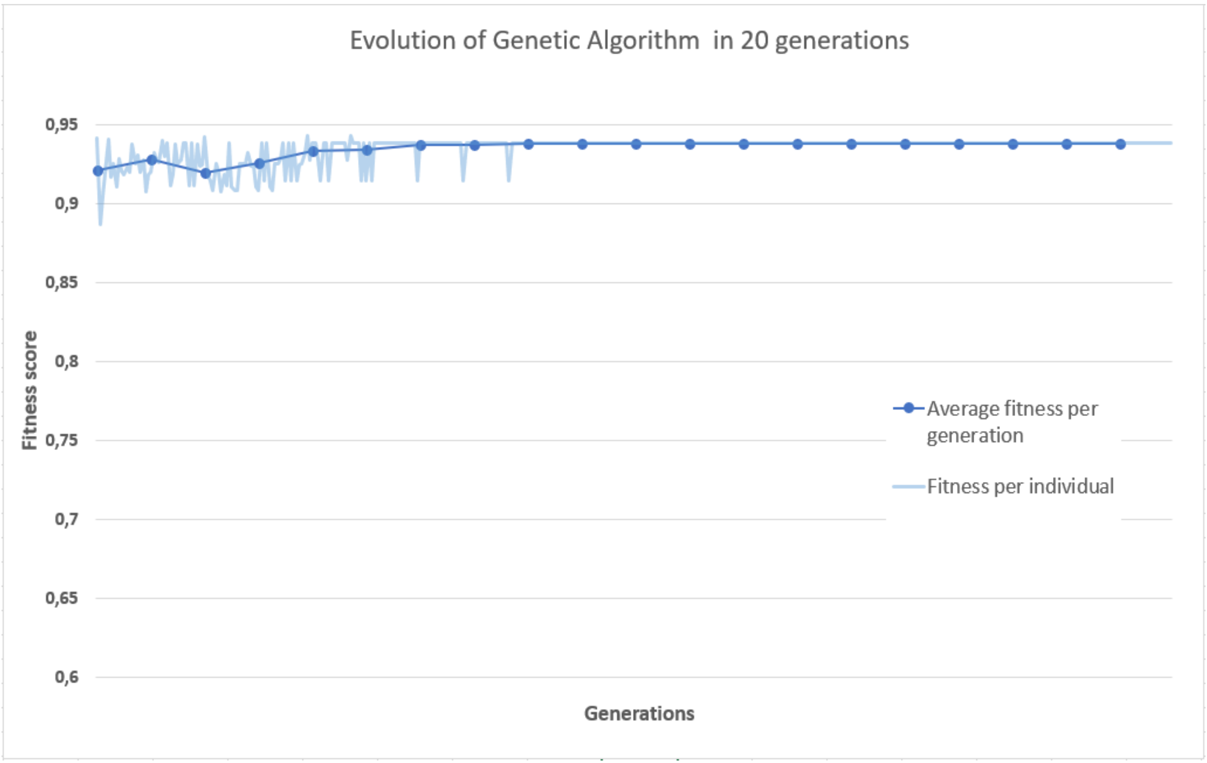


Figure 43: Evolution of improved genetic algorithm, 50% mutation rate

6.3. Evaluation of SVM

After performing the tests described above, the final model subset of attributes was chosen. This subset was obtained as the best individual from the tests performed before, which had a 94.37% accuracy.

A series of values were tested for the SVM parameters. The possible parameters for an SVM are (as explained in section 5):

- **Gamma** value: A small gamma value will classify two points as belonging to the same class even if they are far apart.
- **C**: With higher values of C, the gap between classes is smaller.

The values for these parameters in the tests performed to obtain the best combination of attributes (tests done on the genetic algorithm) were: C=100, Gamma =0.001. The experiment was repeated 5 times to obtain reliable results. The tables below illustrates the accuracy percentage obtained for each combination of values on each experiment:

		Gamma						
		0.001	**0.002**	**0.003**	**0.005**	**0.008**	**0.01**	
	100	94.37%	96.67%	96.67%	93.3%	93.3%	93.3%	
	200	96.67%	96.67%	96.67%	93.3%	93.3%	93.3%	
C	**500**	96.67%	96.67%	96.67%	93.3%	93.3%	93.3%	
	800	96.67%	96.67%	96.67%	93.3%	93.3%	93.3%	
	50	100%	100%	100%	93.3%	93.3%	93.3%	
	25	100%	100%	100%	96.67%	96.67%	96.67%	
	15	100%	100%	100%	96.67%	96.67%	96.67%	
	10	100%	100%	100%	96.67%	96.67%	96.67%	
		0.02	**0.03**	**0.05**	**0.1**	**0.2**	**0.3**	**0.5**
	100	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%
	200	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%
	500	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%
C	**800**	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%
	50	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%
	25	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%
	15	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%
	10	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%

Table 3: Evaluation of SVM, experiment 1

		Gamma						
		0.001	0.002	0.003	0.005	0.008	0.01	
C	100	91.5%	90.84%	90.84%	89.54%	88.23%	88.23%	
	200	91.5%	90.84%	90.84%	89.54%	88.23%	88.23%	
	500	91.5%	90.84%	90.19%	89.54%	88.23%	88.23%	
	800	91.5%	90.19%	90.19%	89.54%	88.23%	88.23%	
	50	91.5%	90.84%	90.84%	89.54%	88.23%	88.23%	
	25	91.5%	90.84%	90.84%	89.54%	88.23%	88.23%	
	15	91.5%	90.84%	90.84%	89.54%	88.23%	88.23%	
	10	91.5%	90.84%	90.84%	89.54%	88.23%	88.23%	
		0.02	0.03	0.05	0.1	0.2	0.3	0.5
C	100	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%
	200	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%
	500	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%
	800	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%
	50	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%
	25	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%	88.23%
	15	88.23%	88.23%	88.23%	88.23%	88.23%	89.54%	88.23%
	10	88.23%	88.23%	88.23%	88.23%	88.23%	89.54%	88.23%

Table 4: Evaluation of SVM, experiment 2

		Gamma					
		0.001	0.002	0.003	0.005	0.008	0.01
C	100	95.86%	95.86%	95.86%	95.86%	94.48%	94.48%
	200	95.86%	95.86%	95.86%	95.86%	94.48%	94.48%
	500	95.86%	95.86%	95.86%	95.86%	94.48%	94.48%
	800	95.86%	95.86%	95.86%	95.86%	94.48%	94.48%
	50	95.86%	95.86%	95.86%	95.86%	94.48%	94.48%
	25	95.86%	95.86%	95.86%	95.86%	94.48%	94.48%
	15	95.86%	95.86%	95.86%	95.86%	94.48%	94.48%

C	10	95.86%	95.86%	95.86%	95.86%	94.48%	94.48%	
		0.02	**0.03**	**0.05**	**0.1**	**0.2**	**0.3**	**0.5**
	100	94.48%	94.48%	94.48%	94.48%	94.48%	93.55%	93.55%
	200	94.48%	94.48%	94.48%	94.48%	94.48%	93.55%	93.55%
	500	94.48%	94.48%	94.48%	94.48%	94.48%	93.55%	93.55%
C	**800**	94.48%	94.48%	94.48%	94.48%	94.48%	93.55%	93.55%
	50	94.48%	94.48%	94.48%	93.55%	93.55%	93.55%	93.55%
	25	94.48%	94.48%	94.48%	93.55%	93.55%	93.55%	93.55%
	15	94.48%	94.48%	94.48%	93.55%	93.55%	93.55%	93.55%
	10	94.48%	94.48%	94.48%	93.55%	93.55%	93.55%	93.55%

Table 5: Evaluation of SVM, experiment 3

		Gamma						
		0.001	**0.002**	**0.003**	**0.005**	**0.008**	**0.01**	
	100	97.72%	97.72%	97.72%	97.94%	97.94%	97.94%	
	200	97.72%	97.72%	97.94%	97.94%	97.94%	97.94%	
C	**500**	97.72%	97.94%	97.94%	97.94%	97.94%	97.94%	
	800	97.72%	97.94%	97.94%	97.94%	97.94%	97.94%	
	50	97.72%	97.72%	97.72%	97.72%	97.94%	97.94%	
	25	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%	
	15	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%	
	10	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%	
		0.02	**0.03**	**0.05**	**0.1**	**0.2**	**0.3**	**0.5**
	100	97.94%	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%
	200	97.94%	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%
	500	97.94%	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%
C	**800**	97.94%	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%
	50	97.94%	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%
	25	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%
	15	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%

	10	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%	97.72%

Table 6: Evaluation of SVM, experiment 4

		Gamma						
		0.001	**0.002**	**0.003**	**0.005**	**0.008**	**0.01**	
	100	97.86%	97.53%	97.53%	97.69%	97.69%	97.69%	
	200	97.86%	97.53%	97.86%	97.69%	97.69%	97.69%	
C	**500**	97.86%	97.86%	97.86%	97.69%	97.69%	97.69%	
	800	98.18%	97.86%	97.86%	97.69%	97.69%	97.69%	
	50	97.86%	97.53%	97.53%	97.36%	97.69%	97.69%	
	25	97.86%	97.53%	97.53%	97.36%	97.36%	97.36%	
	15	97.86%	97.53%	97.53%	97.36%	97.36%	97.36%	
	10	97.53%	97.53%	97.53%	97.36%	97.36%	97.36%	
		0.02	**0.03**	**0.05**	**0.1**	**0.2**	**0.3**	**0.5**
	100	97.69%	97.69%	97.36%	97.36%	97.36%	97.36%	97.36%
	200	97.69%	97.69%	97.36%	97.36%	97.36%	97.36%	97.36%
	500	97.69%	97.69%	97.36%	97.36%	97.36%	97.36%	97.36%
C	**800**	97.69%	97.69%	97.36%	97.36%	97.36%	97.36%	97.36%
	50	97.69%	97.69%	97.36%	97.36%	97.36%	97.36%	97.36%
	25	97.36%	97.36%	97.36%	97.36%	97.36%	97.36%	97.36%
	15	97.36%	97.36%	97.36%	97.36%	97.36%	97.36%	97.36%
	10	97.36%	97.36%	97.36%	97.36%	97.36%	97.36%	97.36%

Table 7: Evaluation of SVM, experiment 5

Table 8 illustrates the average accuracy score for all values of C and gamma, calculated as the geometric mean of the results obtained in the 5 experiments performed before. To provide a better understanding of the table, a heat map was used to visually locate the best results. Darker blue indicates higher accuracy, whereas lower accuracy is indicated with a lighter shade of blue.

		Gamma						
		0.001	**0.002**	**0.003**	**0.005**	**0.008**	**0.01**	
	100	95,43%	95,69%	95,69%	94,81%	94,26%	94,26%	
	200	95,89%	95,69%	95,80%	94,81%	94,26%	94,26%	
C	**500**	95,89%	95,80%	95,66%	94,81%	94,26%	94,26%	
	800	95,96%	95,66%	95,66%	94,81%	94,26%	94,26%	
	50	96,54%	96,34%	96,34%	94,71%	94,26%	94,26%	
	25	96,54%	96,34%	96,34%	95,38%	94,82%	94,82%	
	15	96,54%	96,34%	96,34%	95,38%	94,82%	94,82%	
	10	96,48%	96,34%	96,34%	95,38%	94,82%	94,82%	
		0.02	**0.03**	**0.05**	**0.1**	**0.2**	**0.3**	**0.5**
	100	94,93%	94,89%	94,82%	94,82%	94,82%	94,64%	94,64%
	200	94,93%	94,89%	94,82%	94,82%	94,82%	94,64%	94,64%
	500	94,93%	94,89%	94,82%	94,82%	94,82%	94,64%	94,64%
C	**800**	94,93%	94,89%	94,82%	94,82%	94,82%	94,64%	94,64%
	50	94,93%	94,89%	94,82%	94,64%	94,64%	94,64%	94,64%
	25	94,82%	94,82%	94,82%	94,64%	94,64%	94,64%	94,64%
	15	94,82%	94,82%	94,82%	94,64%	94,64%	94,92%	94,64%
	10	94,82%	94,82%	94,82%	94,64%	94,64%	94,92%	94,64%

Table 8: Evaluation of SVM, average accuracy

6.4. Evaluation of final model

After choosing the genetic algorithm and parameters for the SVM, the final model was tested with a new set of malware samples with 10 samples belonging to each type and family used for the learning.

As seen on Table 8, there were 12 combinations for SVM parameters which gave very similar score on experimental results. From them, 3 gave slightly better results, however the difference is so small it’s not significant. To choose one of them, the definition of gamma and C were taken into account. Given a sample that has been classified wrongly, it is preferred that said sample is classified in its correct type, even if the family is wrong. For that reason, the gap between classes should be higher (smaller value of C), so families which belong to the same type are brought together, but different types are far apart. On the other hand, the results for higher values of Gamma in the experiments were worse than those with smaller values, thus the smallest Gamma from within the best 3 was chosen.

The chosen final values for Gamma and C would then be: Gamma = 0.001, C = 15. However, to obtain more significant results about the performance of the model, the tests were performed with all 12 best combinations of Gamma and C shown on Table 8.

Table 9 shows the accuracy obtained for each combination of parameters on the new set of samples. Table 11 shows the incorrectly classified instances, and the class where they were placed, to allow for a better understanding of the performance of the system.

		Gamma		
		0.001	**0.002**	**0.003**
C	**50**	92,81%	92,81%	92,81%
	25	92,81%	92,81%	92,81%
	15	92,81%	92,81%	92,81%
	10	92,81%	92,81%	92,81%

Table 9: Evaluation of final model

The model obtained was the same in all cases, thus the accuracy remains unchanged for all experiments.

Table 11 illustrates the confusion matrix, which shows the incorrectly classified instances of each class. To allow a better understanding of the table, the classes are represented with numbers as shown on Table 10:

Type	Family	Number in table
Adware	Hummingbad	1

Banker	Judy	2
	Overlay	3
	OverlayLocker	4
	Sberbank	5
Exploit	Godless	6
Ransomware	Xbot	7
Spyware	BeaverGangCounter	8
	RedDrop	9
	Tizi	10
Trojan	Marcher	11
	Triada	12

Table 10: Naming of classes in confusion matrix

		Classified as											
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
	1	**10**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	2	3	**7**	0	0	0	0	0	0	0	0	0	0
	3	1	0	**9**	0	0	0	0	0	0	0	0	0
	4	**0**	0	0	**9**	**1**	0	0	0	0	0	0	0
	5	**0**	0	0	0	**10**	0	0	0	0	0	0	0
Class	**6**	2	0	0	0	0	**8**	0	0	0	0	0	0
	7	2	0	0	0	0	0	**8**	0	0	0	0	0
	8	**0**	0	0	0	0	0	0	**10**	0	0	0	0
	9	**0**	0	0	0	0	0	0	0	**10**	0	0	0
	10	**0**	0	0	0	0	0	0	0	0	**10**	0	0
	11	**0**	0	**2**	0	0	0	0	0	0	0	**8**	0
	12	**0**	0	0	0	0	0	0	0	0	0	0	**10**

Table 11: Confusion matrix

6.5. Analysis of experimental results

After performing the experiments described in the previous sections, the results obtained are analysed to understand the performance of the system implemented.

The first element to analyse is the genetic algorithm. When looking at the results obtained from the experiments (on section 6.2), it becomes clear that the algorithm never evolves to a solution over 90% accuracy unless it has individuals with high accuracy in the initial population. Although it does evolve slightly, this means that it's preferable to choose a random combination of attributes for the training until one good individual is found than use the algorithm to filter the attributes.

This result is not surprising, as the search space of the algorithm is huge. With about 44000 attributes, there are approximately 2^{44000} possible combination of attributes to be explored by the algorithm. By changing the algorithm's codification, the search space is reduced but still too broad for the algorithm to evolve properly. Since the objective of this section was to find the best combination of attributes, and the best individual had an accuracy of 94.4%, it was considered fit enough to use as solution; however, the genetic algorithm is not enough for this attribute selection. A new codification, different attribute selection methods, or combination of genetic algorithm with other methods should be explored to obtain a better model for future research and that can be suitable for malware analysis. This question will be explored in depth in **Future work**.

The attributes obtained, however, prove to be significant for the learning. The high accuracy obtained in all experiments shows that the static analysis of the APKs is a good method to analyse Android malware, with the only downside of the high number of attributes obtained. Different methods could also be tested in future implementations, as will be discussed in **Future work**.

The SVM gives a very reliable model. Looking at the confusion matrix on the real-life test (with 120 samples equally distributed between the 12 possible families), 6 out of 12 classes have incorrectly classified instances. Out of the 6, 5 are incorrectly classified as Adware, family Hummingbad. The model seems to lean towards that class. Since the model was trained using 70% randomized data from 1175 samples, it's likely that more samples from this particular family were included in the training set, therefore slightly biasing the model. This bias is not consider significant, since the tests performed prior to the real-life experiment obtained very high accuracy with a test set that contained the remaining 30% of the randomized data from 1175 samples. Furthermore, in all cases there are no more than 2 samples classified incorrectly in this class, with the exception of the samples belonging to Adware Judy. This last exception belongs to the same type as Hummingbad, incorrectly classified instances within types are to be expected.

All in all, the model can be improved in future implementations, but it proves to give a suitable adaptive solution to the problem presented at the beginning of the project.

7. MANAGEMENT

This section explains the details about the management process related to the project, including the planning of the project in time and the costs involved in the development.

7.1. Planning

The project was planned considering two major aspects: development of the source code, including design, implementation and evaluation; and the creation of this report, including the previous research and analysis of the legal and socio-economic environment of the project, and the drafting of the document.

The project was started in December 2018, and finished in the first week of June 2019. The implementation of the code and creation of the report were planned jointly, considering that some aspects related to the report are essential prior to the development process. For this reason, the first step was an initial research on the topic and related work (**STATE OF THE ART**), and also the socio economic and legal aspects that could affect the project, to make sure the project is compliant with all regulations and consider the effects it could have from a socio economic perspective (**LEGAL AND SOCIO-ECONOMIC ENVIRONMENT**). Once this research was finished, the implementation was planned analysing and designing the system (as seen on **SYSTEM ANALYSIS** and **SYSTEM DESIGN**). After these tasks had been completed, the samples needed for the project were obtained. These samples were downloaded from the free repository *ContagioDump*. The code was then implemented (explained in section **IMPLEMENTATION**) .

The implementation of the source code is divided in three parts: genetic algorithm, SVM and integration of both. Since the SVM was developed using *Scikit Learn*, the majority of the implementation time was invested in designing and implementing the genetic algorithm.

After the code was implemented, a series of experiments to test the performance of the system were designed (**EVALUATION**). These tests were executed over three weeks, due to the high computational time needed for each execution (see 6). The tests were then analysed and a series of conclusions were extracted.

The process described above was documented in parallel to the implementation, and all the results and conclusions are shown in this document (see 6.5 and 9).

The development of the whole project is illustrated below using a Gantt chart[32].

	December				January				February				March				April				May				June
Tasks	week 1	week 2	week 3	week 4	week 1	week 2	week 3	week 4	week 1	week 2	week 3	week 4	week 1	week 2	week 3	week 4	week 1	week 2	week 3	week 4	week 1	week 2	week 3	week 4	week 1
Total time	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
State of the Art	■	■	■																						
Socio Economic Environment	■	■	■																						
Analysis and Design				■	■	■																			
Development							■	■	■	■	■	■	■	■											
Obtain samples							■																		
Genetic Algorithm								■	■	■	■														
SVM												■	■												
Integration														■											
Tests															■	■	■	■	■	■	■	■	■	■	■
Design of experiments															■	■									
Genetic Algorithm																	■	■	■						
SVM																				■	■				
Final model																						■	■		
Analysis of Results																								■	■
Project documentation	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■

Table 12: Planning of the project

7.2. Budget

The budget of the project is divided in two main sections: direct and indirect costs. Direct costs refer to all the resources needed in form of specific cost objects, such as materials, labour, licenses, etc. Indirect costs, however, can't be traced to specific cost objects: these include items such as rent, power, utilities, insurance, fees, etc.

7.2.1. Direct costs

The direct costs of the project correspond to the resources needed for the implementation. Since all the code was developed using open-source means, no software license was acquired, and there was no need for any specific hardware device, the direct costs relate to the computer used in the implementation and the labour of the developer.

Human resources

There was only one person involved in the development of the project. The project had a duration of 6 months, which equal a total of 480 hours. The salary of the developer is taken as the average salary for an entry level Python developer in North Holland (where the developer of the project was based at the time of the creation of the project) according to *Payscale*[31], which equals a total of €14,83/hour.

Estimated hours	Cost per hour	Total cost
480	€14,83	**€7188,4**

Table 13: Estimated personnel costs

Material resources

The costs for the resources and materials needed needs to be calculated including the amortization of the materials, which considers the lifespan of each device. This cost is calculated as the depreciated cost multiplied by the duration of the project.

$$depreciation = \frac{(initial\ asset\ cost - residual\ value)}{useful\ life\ of\ asset(months)}$$

$$total\ cost = \ depreciation(months) * duration\ of\ project(months)$$

Where residual value is the estimated value of the asset at the end of its useful life. For this project, the only material asset needed was a laptop. Find below the estimated cost for this asset:

Asset	Initial cost	Residual value	Useful life (months)	Duration of project(months)	Total cost
HP EliteBook 840	€ 1.293,49	€600	48	6	**€86,63**

Table 14: Estimated material costs

7.2.2. Indirect costs

Indirect costs are calculated as a 20% of the total direct costs. These cover costs such as power used to run the application, and can’t be traced to human resources or material.

The total indirect costs are **€1455**

7.2.3. Risk

The risk costs refer to the costs that the risks involved in the project add to the total cost. The risks can affect the developer, for example in case of injury or illness; they can also affect the material assets, as it’s the case of loss or breakage of the computer where the project is being developed. There are no security risks affecting this project since there is no sensitive or private data being used.

The risk costs are calculated as a 10% of the cost of the project, including direct and indirect costs.

The total risk costs are **€873.**

7.2.4. Total costs

The total cost of the project adds the expected benefit to the total amount. The expected benefit for this project equals a 20% of the total budget: **€1920,6**. The VAT is added to the total costs. The VAT is taken from the VAT general tariff in the Netherlands (where the project was developed), 21%.

Find below the total costs of the project:

Cost description	Value
Direct Costs	€7275,03
Indirect Costs	€1455
Risk Costs	€873
TOTAL	€9603,03
Expected Benefit	€1920,6
TOTAL COSTS (without VAT)	€11523,64
VAT	€2419,96
TOTAL COSTS (with VAT)	**€13943,6**

-

8. LEGAL AND SOCIO-ECONOMIC ENVIRONMENT

This section explores all the socio-economic and legal factors related to the developed project. Before the implementation of a software tool, there are certain aspects that must be taken into account relating this environment; these can affect usage of personal or sensitive data, intellectual property, possible social impact of the system, etc.

8.1. Legal

The project does not use any kind of private or sensitive data; it does also not modify any data that belongs to a user or particular privately. All samples are APKs obtained from a public repository, furthermore, the samples correspond to malicious APKs which had been published before being uploaded to the repository (since the intention of the developers of the malware was to infect users by infiltrating the apps in their devices). Python is an open source programming language, and all its libraries are available free of license. All the code used was entirely developed during the project, and no third-party code or resources were needed (except for the previously mentioned samples and Python's libraries).

For these reasons, there are no laws or regulations regarding data protection that affect this project. However, any creative work, even if available publicly, is subject to copyright by default. Thus, it is needed to analyse the intellectual property of the project and whether it is going to be made open source or not.

8.1.1. Intellectual property and open source projects

All creative work is exclusive by copyright by default when created, according to open source guide[33]. Even if the project is published, for example in a public *Github* repository, it's still subject to copyright of the author. This means that although anybody can access and see the content of the project, nobody can use, copy, distribute or modify the content of the work.

To make the project open source, there are several licenses available online. The intention of this project is to be of help in future research and provide an scalable approach to the growing problem of cyberattacks on Android devices. For these reasons, the project is made open source.

The license chosen for this project is provided by MIT[34]:

```
MIT License

Copyright (c) 2019 Sara Yuste Fernandez Alonso

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject
to the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,   FITNESS   FOR   A   PARTICULAR   PURPOSE   AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

Figure 44: MIT License for open source projects

8.2. Socio-Economic impact

Due to the popular use of smartphones and specifically Android, the project has a high social impact. The widely extended different versions of malware affect many users, and in some cases can imply a high economic loss, not only for one user, but can sometimes affect a whole enterprise or sector. The general public is sometimes not well informed about the cyberattacks and risks they face when using their smartphones. Projects like the one developed are have a big impact in future protection against cybercrime, and help reduce the impact of cyberattacks in a growing and changing landscape such as personal mobile applications.

8.2.1. Smartphones: private and sensitive information

According to Statista[35], there were 4.57 billion mobile phone users in 2018. In 2019, this number is forecast to reach 4.68 billion. As reported by a research made by Techjury[36], 47% of US smartphone users say they couldn't live without their devices , 62% of smartphone users have made a purchase on the device and there are 194 billion apps downloads in 2018 worldwide.

These statistics show the close relationship between users and mobile phones, which are an essential asset in the users' everyday lives. Users store, access or modify personal and sensitive data using their phones. An example of this are the banking apps that most banks offer, from which a user can operate on their bank accounts, transfer money, check their balance, etc. Not only banking data, but other sensitive data such as personal pictures or confidential information (for example, data about clients on a work phone) is often stored and accessed in mobile phones. Successful attacks to a mobile device allow the attacker to potentially gain access to this data, trade with the information obtained, blackmail the user, or ruin the user's o their enterprise's reputations.

8.2.2. Cybercrime

Cybercrime refers to a crime where a computer is the victim of a crime, or is used as a tool to commit a crime[37]. Cybercrime can target multiple computational devices, but due to the high popularity of smartphones, these have become an interesting target for cybercriminals over the last few years. As shown by G Data, mobile malware rose about 40% in 2018, with around 3.2 million malicious apps located by the end of the third quarter of 2018[38].

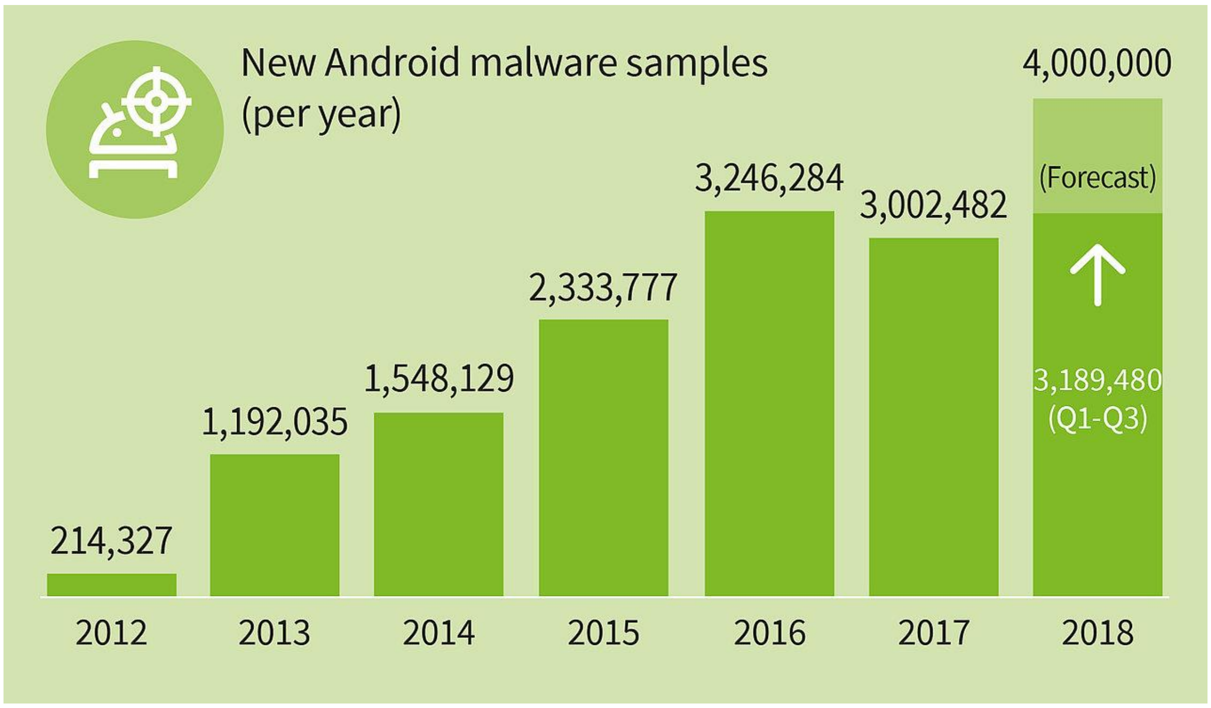


Figure 45: New Android malware samples per year 2012-2018. (GData, 2018)

The increasing number of threats to Android has been addressed by the industry, with measures such as the one taken by Google since summer 2018 (as seen on technological portal The Verge[39]), which stated a mandatory security update for at least two years for popular Android manufacturers.

Most Android users, however, are unaware of these numbers. Since most of the malicious apps use social engineering, or simply rely on the user's ignorance about these threats, they can be found in the PlayStore or any legit source the user might believe trustworthy and reliable, and can infect the device easily, being installed by the user.

The number of malicious apps grows so rapidly, it is difficult to find adaptable and fast countermeasures to keep users safe from these attacks. It becomes more clear for Android manufacturers and cyber security experts that there is a need to find a solution which provides a fast way to find the malicious apps efficiently, so they can be removed from the PlayStore or other sources before they reach the users.

9. CONCLUSIONS

After implementing the system, this section explains a series of personal and technical conclusions to summarize the content of the project.

9.1. Personal and technical conclusions

The development of this project has been a challenging task. From the research about a topic such as Android malware, from which the student had no prior formation, the understanding of malware and Android APKs, its behaviour… to the implementation of the genetic algorithm, SVM and integration of both, the development process has tested the experience of the author in programming, as well as improved the knowledge on both computational science and cybersecurity.

Looking at the objectives set at the beginning of the project, the challenge at hand was to develop a scalable, efficient way to analyse malicious Android app samples and determine the type and family they belong to. The solution developed obtained a very high accuracy in all cases, included the real-life testing, and has proven to work with 6 different malware types and 12 different malware families. Although there are many improvements that can be made to the implementation, especially regarding the recurrent problem of the number of attributes extracted from the APKs, the objectives of the project can be considered as met after the development of the system. The model obtained can be scaled to other malware applications, and adapted if necessary with new samples, and provides a reliable guide for Android cyber security experts to speed up their work when analysing malicious APKs.

Furthermore, another goal which was set at the beginning of the project was to make it available for future research, and to be accessible by other developers. The whole project has been developed with open source means, and has been open sourced to be obtainable for free, to contribute to a further development of the project.

9.2. Future work

As mentioned before, the main challenge of the system developed was to operate on such a high number of attributes extracted from the APKs. The first measure to take when working further on this project would be to look for a solution regarding this problem. The author suggests the following:

- Apply decision models prior to the genetic algorithm to decide which attributes are more relevant. For example, creating a series of decision trees and selecting the most significant attributes used to create the branches; prune the tree at a certain height and use a genetic algorithm to explore the remaining attributes.
- Combine statistical feature reduction methods with AI feature reduction. For example, the already mentioned covariance matrix combined with a decision tree; the tree would work as described before, but a genetic algorithm would not be used to explore the remaining attributes.
- Use the three methods proposed above in combination; first selection with a statistical model to remove strongly related attributes, a series of search trees to obtain the most

recurrent significant attributes, and a genetic algorithm to explore through the remaining attribute space.

- Extract attributes by different means. For example, use a dynamic analysis and extract less attributes about the behaviour of the application. Try dynamic and hybrid approaches.
- Extract different attributes. Instead of using the calls to the Android library, analyse the structure of the code inside each APK, similarly to a flow diagram. Create graphs that represent the interaction of methods within the APK, and use the graphs as attributes for the learning.

Another possible future line of work is exploring different learning algorithms in the classifier model. This project focused on the use of SVM for the classification, but further research could be done using different algorithms or techniques.

GLOSSARY OF TERMS

Ad (advertisement): A notice or announcement in a public medium promoting a product, service, or event. In the context of this works, it refers to the announcements displayed on mobile phones' apps.

Algorithm: A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

App: In computing, An application, especially as downloaded by a user to a mobile device.

Assembly code: In computing, The conversion of instructions in low-level code to machine code.

Binary array: A collection of numbers which can have the value 1 or 0.

Buffer: A temporary memory area in which data is stored while it is being processed or transferred.

CLI (Command Line Interface): A text based user interface used to view and manage computer files.

Computer vision: A field of science which aims to make computers gain understanding from images or videos.

CSV file (comma separated value file): A file format which contains values separated by commas.

Cyberattack: An attempt by hackers to damage or destroy a computer network or system.

Cybercrime: Criminal activities carried out by means of computers or the Internet.

Decipher: Convert (a text written in code, or a coded signal) into normal language.

Decision models (AI): A subdivision of AI algorithms which interpret the knowledge using a series of decisions, such as decision trees.

DEX (Dalvic Executable): A component of an APK (Android Application Package) which contains the compiled source code.

Encryption: The process of converting information or data into a code, especially to prevent unauthorized access.

Family (malware): A set of malware applications or programs which belong to the same malware type and present common features.

Feature: In AI, a piece of data that can be used to analyse a sample.

General purpose (programming language): An programming language that doesn’t operate only in a specific field or environment, but can be used to implement several different applications.

Git: An open source distributed version control system designed for code sharing.

Github: An online code repository which implements Git control system.

Hyperplane(geometry): A subspace with one dimension less to the ambient space. In a 3D dimensional space, a hyperplane is a 2D plane.

Infection (software): The action of a software application or program being infiltrated by a malicious software.

Interface (program): A point where two systems, meet and interact. Can refer to the interaction between two software components or a subject with a software component.

Malicious: Intending or intended to do harm.

Malware: Software that is specifically designed to disrupt, damage, or gain unauthorized access to a computer system.

Metamorphic malware: Malware that is rewritten with each iteration so each version of the code is different from the previous one.

Natural Language Processing: The application of computational techniques to the analysis and synthesis of natural language and speech.

Open source software: Software that can be used, copied, distributed or modified freely.

Operating system (OS): The low-level software that supports a computer's basic functions, such as scheduling tasks and controlling peripherals.

Optimization problem: The problem of finding the best solution of all possible solutions.

Parser: A program for analysing (a string or text) into logical syntactic components.

Pay-by-click: A form of paid digital marketing where advertisers pay a fee each time their ad is clicked.

Privilege (software): The authority to perform security relevant functions on a computer.

Sandbox: A virtual space in which new or untested software or coding can be run securely.

Smartphone: A mobile phone that performs many of the functions of a computer, typically having a touchscreen interface, Internet access, and an operating system capable of running downloaded apps.

Social engineering: The use of deception to manipulate individuals into divulging confidential or personal information that may be used for fraudulent purposes.

Speech recognition: The process of enabling a computer to identify and respond to the sounds produced in human speech.

Vulnerability (cybersecurity): Flaw in a computer system that can leave it open to attacks.

All definitions hereby provided according to Oxford English Dictionary[40]

BIBLIOGRAPHY

[1] Net Market Share for Mobile Operating Systems. [Online] available at: https://netmarketshare.com/operating-system-market-share (**access**: December 2018)

[2] C. Kumar Behera, D. Lalitha Bhaskari, "Different Obfuscation Techniques for Code Protection", *Procedia Computer Science*, Volume 70, Pages 757-763

[3] "History of Artificial Intelligence". Queensland Brain Institute. https://qbi.uq.edu.au/brain/intelligent-machines/history-artificial-intelligence (**access:** December 2018)

[4] C. Smith, "The History of Artificial Intelligence" , History of Computing, University of Washington, Washington, December 2006

[5] S. Ray, "History of AI", Towards Data Science, https://towardsdatascience.com/history-of-ai-484a86fc16ef, (**access:** December 2018)

[6] A.M.Turing, "On computable numbers, with and application to the Entscheidungsproblem", November 1936. [Online] available at: https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

[7] A.M.Turing, "Computing machinery and Intelligence", *Mind*, Volume 42, Pages 433-460, 1950. [Online] Available at: https://www.csee.umbc.edu/courses/471/papers/turing.pdf (**access:** December 2018)

[8] C.E. Shannon, "Programming a Computer for Playing Chess", *Philosophical Magazine*, Volume 41, No.314, March 1950.

[9] B. Poczos, "Introduction to Machine Learning", Carnegie Mellon University

[10] Rumilhart et al., "Learning representations by back-propagation errors" , *Nature*, 1986

[11] Bryson et al., "Backpropagation", 1963

[12] F.Corea, "AI Knowledge Map: How To Classify AI Technologies", Forbes, August 2018. [Online] Available at: https://www.forbes.com/sites/cognitiveworld/2018/08/22/ai-knowledge-map-how-to-classify-ai-technologies/#40b4f6f47773 (**access:** December 2018)

[13] J. Haugeland, "Artificial Intelligence The Very Idea", MIT Press, Cambridge, MA, 1985

[14] P.H. Winston, "*Artificial Intelligence at MIT, Expanding Frontiers,* MIT Press, Volume 1, 1990. [Online] Available at: https://web.media.mit.edu/~minsky/papers/SymbolicVs.Connectionist.html (**access:** December 2018)

[15] "Brief History of Machine Learning", *Erogol*, [Online] Available at: http://www.erogol.com/brief-history-machine-learning/ (**access:** December 2018)

[16] J. Sahs and L. Khan. "A machine learning approach to android malware detection." *Intelligence and Security Informatics Conference (EISIC*), 2012 European, pages 141–147, August 2012.

[17] A. Shabtai. "Malware detection on mobile devices". *Mobile Data Management (MDM), 2010 Eleventh International Conference*, pages 289–290, May 2010.

[18] S.Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. “A new android malware detection approach using bayesian classification” *Advanced Information Networking and Applications (AINA)*, 27th International Conference, pages 121–128, March 2013.

[19] Z. Xiaoyan, F. Juan, and W. Xiujuan. “Android malware detection based on permissions.” *Information and Communications Technologies*, International Conference, pages 1–5, May 2014.

[20] Te-En Wei, Ching-Hao Mao, A.B. Jeng, Hahn-Ming Lee, Horng-Tzer Wang, and Dong-Jie Wu. “Android malware detection via a latent network behavior analysis.” *Trust, Security and Privacy in Computing and Communications (Trust- Com)*, 11th International Conference on, pages 1251–1258, June 2012.

[21] Hyo-Sik Ham and Mi-Jung Choi. “Analysis of android malware detection performance using machine learning classifiers.” *ICT Convergence (ICTC*), 2013 International Conference on, pages 490–495, October 2013.

[22] K. Patel and B. Buddhadev, “Detection and Mitigation of Android Malware Through Hybrid Approach”, 2015.

[23] A. Pektaş, M.Çavdar and T.Acarman,. “Android Malware Classification by Applying Online Machine Learning.” 2016

[24] A. Sharma and S. K. Sahay, “An effective approach for classification of advanced malware with high accuracy”, Department of Computer Science and Information System, Birla Institute of Technology and Science, K. K. Birla Goa Campus, NH-17B, ByPass Road, Zuarinagar-403726, Goa, India

[25] T. Altyeb. “Classification of Android Malware Applications using Feature Selection and Classification Algorithms.” VAWKUM Transactions on Computer Sciences. 2016

[26] M. Zubair Rafique, Ping Chen, Christophe Huygens, Wouter Joosen. “Evolutionary Algorithms for Classification of Malware Families through Different Network Behaviors” KU Leuven, Leuven, Belgium, 2014

[27] A. Firdaus, Nor Badrul Anuar, Ahmad Karim, Mohd Faizal Ab Razak . “Discovering optimal features using static analysis and a genetic search based method for Android malware detection” Department of Computer System and Technology, University of Malaya, Kuala Lumpur, Malaysia

[28] Yongfeng Li, Tong Shen, Xin Sun, Xuerui Pan, and Bing Mao , “Detection, Classification and Characterization of Android Malware Using API Data Dependency”. State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China

[29] M.N. Yusoff and A. Jantan. “A Framework for Optimizing Malware Classification by Using Genetic Algorithm.” Communications in Computer and Information Science. 2011.

[30] [Online] UML diagrams

[31] [Online] Payscale, Available at: https://www.payscale.com/research/NL/Job=Data_Scientist/Salary/66c296ce/Entry-Level

[32] [Online] Gantt Diagrams

[33] “The Legal Side of Open Source”, *Open Source Guides*, [Online] , available at: https://opensource.guide/legal/ (**access:** June 2019)

[34] MIT License for open source projects. [Online] Available at: https://choosealicense.com/licenses/mit/ (**access:** June 2019)

[35] "Forecast of Mobile phone users in 2020", *Statista*. [Online] Available at: https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/ (**access:** June 2019)

[36] "Smartphone usage", *Techjury*. [Online] Available at: https://techjury.net/stats-about/smartphone-usage/ (**access:** June 2019)

[37] "Cybercrime", *Techopedia*. [Online] Available at: https://www.techopedia.com/definition/2387/cybercrime (**access:** June 2019)

[38] "Cyber attacks on Android devices on the rise", *G Data*, [Online] Available at: https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise (**access:** June 2019)

[39] J.Kastrenakes and R.Brandom, "Google mandates two years of security updates for popular phones in new Android contract", *The Verge*, October 2018, [Online] Available at: https://www.theverge.com/2018/10/24/18019356/android-security-update-mandate-google-contract (**access:** June 2019)

[40] Oxford dictionary, [Online] Available at: https://languages.oup.com/ (**access:** June 2019)