iscte

INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Malware detection based on dynamic analysis features

João Guilherme de Lourenço Vieira Duque

Master in Telecommunications and Computer Engineering

Supervisor:
Professor Luís Miguel Martins Nunes, Assistant Professor,
ISCTE-IUL

Co-Supervisor:
Professor Ana Maria Carvalho de Almeida, Assistant Professor,
ISCTE-IUL

October, 2020

Detecção de Malware baseada em recursos de análise dinâmica

João Guilherme de Lourenço Vieira Duque

Mestrado em Engenharia de Telecomunicações e Informática

Orientador:
Professor Doutor Luís Miguel Martins Nunes, Professor Auxiliar,
ISCTE-IUL

Co-Orientadora:
Professora Doutora Ana Maria Carvalho de Almeida, Professora Auxiliar,
ISCTE-IUL

Outubro, 2020

Direitos de cópia ou Copyright
©Copyright: Nome Completo do(a) candidato(a).

O Iscte - Instituto Universitário de Lisboa tem o direito, perpétuo e sem limites geográficos, de arquivar e publicitar este trabalho através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, de o divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Acknowledgements

To my coordinators, the Professors Luís Nunes and Ana de Almeida, I deeply thank all their guidance, shared knowledge and support that permitted the smooth development of this dissertation.

To the collaborators in the AppSentinel Project, Gonçalo Mendes, João Lopes, Nuno Realista and Francisco Palma, alongside Professor Carlos Serrão, I am grateful for the expertise and advice they provided. To ISCTE and Aptoide, I am profoundly grateful for this opportunity to learn.

A special thanks to my family, for giving me the unquestioning support and motivation to better myself and keep achieving whatever challenges I may face, and, most of all for believing in me. To my friends a special thanks as well for their uplifting friendship that I needed to carry on in these unprecedented times.

To all I meantioned above, my sincerest Thanks.

Resumo

O uso generalizado de dispositivos móveis e sua adaptação perfeita às necessidades de cada utilizador por meio de aplicativos úteis (Apps) tornam-os um alvo principal para que criadores de malware obtenham acesso a dados confidenciais do usuário, como detalhes bancários, ou para reter dados e bloquear o acesso do utilizador. Estas apps são distribuídas em mercados que alojam milhões, e portanto, têm as suas próprias formas de detecção automatizada de malware, a fim de dissuadir os desenvolvedores de malware e manter sua loja de apps (e reputação) confiável, mas ainda existem várias apps capazes de ignorar esses detectores e permanecerem disponíveis no mercado para qualquer utilizador fazer o download. As estratégias atuais de detecção de malware dependem principalmente do uso de recursos extraídos estaticamente, dinamicamente ou de uma conjunção de ambos, e de torná-los adequados para aplicações de aprendizagem automática, a fim de dimensionar a detecção para cobrir o número de apps que são enviadas ao mercado. Neste artigo, o foco principal é o estudo da eficácia dos métodos automáticos de detecção de malware e as suas capacidades de acompanhar a popularidade de novo malware, bem como as suas tendências em constante mudança. Analisando o desempenho de algoritmos de ML treinados, com dados do mundo real, em diferentes períodos e escalas de tempo com recursos extraídos estaticamente, dinamicamente e com feedback do utilizador, é possível identificar a configuração ideal para maximizar a detecção de malware.

Palavras-Chave: Detecção de Malware; Análise Dinâmica; Aprendizagem Automática; Android; ETL.

Abstract

The widespread usage of mobile devices and their seamless adaptation to each users' needs by the means of useful applications (Apps), makes them a prime target for malware developers to get access to sensitive user data, such as banking details, or to hold data hostage and block user access. These apps are distributed in marketplaces that host millions and therefore have their own forms of automated malware detection in place in order to deter malware developers and keep their app store (and reputation) trustworthy, but there are still a number of apps that are able to bypass these detectors and remain available in the marketplace for any user to download. Current malware detection strategies rely mostly on using features extracted statically, dynamically or a conjunction of both, and making them suitable for machine learning applications, in order to scale detection to cover the number of apps that are submited to the marketplace. In this article, the main focus is the study of the effectiveness of these automated malware detection methods and their ability to keep up with the proliferation of new malware and its ever-shifting trends. By analising the performance of ML algorithms trained, with real world data, on diferent time periods and time scales with features extracted statically, dynamically and from user-feedback, we are able to identify the optimal setup to maximise malware detection.

Keywords: Malware Detection; Dynamic Analysis; Machine Learning; Android; ETL.

Funding

This work is part of the AppSentinel project, co-funded by Lisboa2020/Portugal2020/EU in the context of the Portuguese Sistema de Incentivos à I&DT - Projetos em Copromoção (project 33953). The authors also would like to acknowledge the FCT Project UIDB/MULTI/04466/2020 (ISTAR-IUL) and UIBD/EEA/50008/2020 (Instituto de Telecomunicações). The authors also appreciate Aptoide's collaboration for providing support and the user feedback and static code analysis data that was used in this study.

Index

Acknowledgements i
Resumo ii
Abstract iii
Funding iv
Index v
Table Index vii
Figure Index viii
Glossary of Terms, Abbreviations and Acronyms ix
Chapter 1 - Introduction 1
1.1. Theme Framework 1
1.2. Motivation and theme relevance 2
1.3. Research questions and objectives 3
1.4. Methodological approach 5
1.5. Structure and organization of the dissertation 6
Chapter 2 - Literature Review 8
2.1. Machine Learning for Malware Detection 8
2.1.1. Large Scale Automated Detection 8
2.1.2. Regarding Data Collection 8
2.2. User Feedback Analysis 9
2.3. Static Analysis 10
2.4. Dynamic Analysis 11
2.5. Hybrid Analysis 12
2.6. Closing Remarks 12
Chapter 3 - Methodology 17
3.1. Datasets 19
3.1. 1. User Feedback Dataset 19
3.1. 2. Static Analysis Dataset 22
3.1. 3. Dynamic Analysis Dataset 25
3.2. Exploratory Data Analysis 31
3.2.1. Principal Component Analysis 31
3.2.2. T-SNE Analysis 33
3.3. Data Preparation 35
3.3.1. Standard Scaler 36
3.3.2. Normalizer 36
3.3.3. Power Transformer (Yeo-Johnson) 37

3.3.3. Quantile Transformer 37
3.4. Detection Models 37
3.4.1. Extreme Gradient Boosting 37
3.4.2. Random Forest 38
3.4.3. Support Vector Machines 38
3.4.4. K-Nearest Neighbor 38
3.3.5. Naïve Bayes Classifier 39
3.5. Model Application 39
3.6. Model Evaluation 40
Chapter 4 - Malware Detection Model Testing 42
4.1 User Feedback Complete Dataset 43
4.2 Static Analysis Complete Dataset 46
4.3 Dynamic Analysis Complete Dataset 49
4.4 Comparison of Results 52
4.5 Ratio Analysis 53
Chapter 5 - Performance Decay Analysis 55
5.1. User Feedback 56
5.2. Static Analysis 58
5.3. Dynamic Analysis 60
Chapter 6 - Conclusions and Recommendations 62
6.1 Main Conclusions 63
6.2 Study Limitations 65
6.3 Future Research Proposals 66
Bibliography 67

Table Index

Table 1 – Literature Review Summary 14
Table 2 – User Feedback Dataset sample distribution 20
Table 3 – User Feedback Dataset Feature description 20
Table 4 – User Feedback Dataset Feature statistics 21
Table 5 – User Feedback Dataset Feature correlation matrix 22
Table 6 – Static Analysis Dataset sample distribution 23
Table 7 – Static Analysis Dataset Feature description 23
Table 8 – Static Analysis Dataset Feature statistics 23
Table 9 – Static Analysis Dataset Feature correlation matrix 24
Table 10 – Dynamic Analysis Dataset sample distribution... 25
Table 11 – Dynamic Analysis Dataset Feature description... 26
Table 12 – Dynamic Analysis Dataset Feature statistics.. 28
Table 13 – Confusion Matrix Example 40
Table 14 – Model performance metrics for the full User Feedback dataset.................. 44
Table 15 – Comparison of top model performance for User Feedback models............ 45
Table 16 – Model performance metrics for the full Static Analysis dataset 47
Table 17 - Comparison of top model performance for Static Analysis models 48
Table 18 – Model performance metrics for the full Dynamic Analysis dataset............ 50
Table 19 - Comparison of top model performance for Static Analysis models 51
Table 20 – XGBoost performance on the different Datasets with “realistic” ratio........ 54
Table 21 – Example of model training and testing framework 56
Table 22 – User Feedback Performance Decay Analysis 57
Table 23 – Static Analysis Performance Decay Analysis 59
Table 24 – Dynamic Analysis Performance Decay Analysis.. 61

Figure Index

Figure 1 - Phases of the CRISP-DM reference model (Chapman, et al. 2000)................ 5
Figure 2 - Prpoposed Framework Layout.. 18
Figure 3 - Dynamic Analysis Dataset Feature correlation heatmap.............................. 30
Figure 4 – User Feedback datasest first 3 PCA components... 32
Figure 5 – Static Analysis datasest first 3 PCA components 32
Figure 6 – Dynamic Analysis datasest first 3 PCA components................................... 33
Figure 7 – User Feedback t-SNE 2D and 3D comparison... 34
Figure 8 – Static Analysis t-SNE 2D and 3D comparison 34
Figure 9 – Dynamic Analysis t-SNE 2D and 3D comparison....................................... 35
Figure 10 – XGBoost User Feedback feature importance... 45
Figure 11 - XGBoost Static Analysis feature importance 49
Figure 12 - XGBoost Dynamic Analysis feature importance 52
Figure 13 – Visual Representation of the User Feedback Performance Decay.............. 58
Figure 14 – Visual Representation of the Static Analysis Performance Decay 60
Figure 15 – Visual Representation of the Dynaic Analysis Performance Decay........... 61

Glossary of Terms, Abbreviations and Acronyms

ADB – Android Debug Bridge

AUC – Area Under the Curve

App – (Mobile) Application

BNB – Bernoulli Naïve Bayes

CRISP-DM – Cross-Industry Standard Process for Datamining

CSV – Comma-Separated Values

GBM – Gradient Boosting Machine

GNB – Gaussian Naïve Bayes

GUI – Graphical User Interface

IDE – Integrated Development Environment

JSON – JavaScript Object Notation

KNN – K-Nearest Neighbors

MD5 – Message Digest (Series) 5

ML – Machine Learning

NLP – Natural Language Processing

NORM – Normalizer Transformation

OS – Operating System

P-R – Precision Recall curve

PCA – Principal Component Analysis

PHA – Potentially Harmful Application

PowerYJ – Power Transform (Yeo-Johnson)

Quant – Quantile Transform

RF – Random Forest

RNN – Recurrent Neural Network

ROC – Receiver Operating Characteristic

SD – Standard Deviation

SSL – Secure Socket Layer

STD – Standardizer Transformation

SVM – Support Vector Machines

TF-IDF – Term Frequency-Inverse Document Frequency

t-SNE – t-Distributed Stochastic Neighbor Embedding

XGBoost – eXtreme Gradient Boosting

Chapter 1 – Introduction

1.1. Theme Framework

Since the introduction of smartphones to the mainstream global audience, their surge in popularity, although remarkable, is unsurprising due to their integration of highly personal and powerful attributes. The unification of portability, high computational power, an ever-increasing access to the Internet and accessibility make these personal devices an almost mandatory tool in our modern society, connecting everything and everyone.

The pervasive nature of these devices and their extended variants, such as tablets and other mobile platforms, gives an incentive for developers to create apps that allow for a wide array of uses, from social networks to mobile games and provide an exceedingly costumizable experience, adapted to each and every users' needs if they so wish. These apps are available on online app stores, to which any app developer can submit their own creations and make them accessible to anyone, as long as they pass the app stores' own publishing requirements.

Since most apps end up dealing with large amounts of personal data, like private information, photos, and even physical location through gps, they become easy targets for developers with malicious intents. By creating apps that are seemingly innocent on the surface, these developers can exploit user given permissions to perform a myriad of harmful actions for their own benefit. This can come in the form of banking details or leveraging personal information to reinforce another attack vector.

With Android being the most used mobile platform, its app stores are of course the most targeted. Given the sheer number of malware apps published, the main concern becomes how to filter through all these apps. Since manually scanning all of this digital content is nearly impossible and current malware detection methods can still be improved, it becomes paramount then, to research which kinds of malware detection methods are most effective and what makes them so, in order to protect users from being exposed to these kinds of malware and as such create a more user-safe environment for the global mobile community.

1.2. Motivation and theme relevance

Smartphones, tablets, and other mobile platforms have long been integrated into our daily lives, due to the factors mentioned above. These personal computing devices, surpassing 1 billion units sold in 2014 (Statista 2020), have fueled the development of complex mobile malware. More than seven million new malware samples have been accumulated by McAfee in 2018 alone (McAfee 2019).

With Android taking the place of the most used mobile Operating System (OS), with approximately 76% of the global market share as of November 2019 (StatCounter 2019), due to its open-source approach and a free of charge Integrated Development Environment (IDE), it gives developers an easier point of entry to its platform than its main competitor, the iOS (StatCounter 2019), which has especially rigorous approval policies and requires developers to use proprietary hardware and software in order to develop and publish iOS apps, placing a higher barrier to entry on those who wish to develop iOS apps. The Android platform also allows its users the ability to install apps from Google unverified sources, that may be accessible through the Internet as well as third-party app stores.

Joining the Android's platform ease of use with the fact that these apps handle substancial volumes of personal and sensitive assets (e.g., financial or messaging apps) portrays the mobile platform as an alluring target for malware developers. In 2013 a report showed that attackers can earn up to 12,000 USD per month with mobile malware (The Register 2013). The increase of mobile malware can be associated with the development of new technologies providing new access points for profitable exploitations (Spreitzenbarth and Freiling 2012, Nigam 2015). In addition, an increase in black markets that profit by selling system vulnerabilities, malware source code and malware development tools, has contributed to a bigger incentive for profit driven malware (InformationWeek 2014). Due to the risks of malware developers bypassing safeguarding mechanisms, further improvements must be made to existing methods.

To protect its users, the main Android App Stores have continuously developed malware detection methods to filter through submitted apps and block those deemed malicious. They achieve this by using either static or dynamic malware detection methods, or in some cases, a conjunction of both, to scan the apps' intent and behaviours to ascertain if it should be classified as malware or not. Unfortunately, each of these methods suffer from some form of exploit and can be bypassed with sufficient knowledge.

For example static analysis detectors, which rely on the analysis of the application's code without running it, mean that they are vulnerable to code obfuscation techniques that remove or limit code access, such as string encryption or renaming of methods and variables (Moser, Kruegel and Kirda 2007). And similar to it, the dynamic analysis, which focuses on examining the applications' behaviour during runtime, remains vulnerable to being hindered or bypassed with native code (e.g., non-Java Code compiled to run with an Android Central Processing Unit (CPU)) or reflection (e.g., modification of interfaces, classes and methods during execution) (Xu, Hassen and Ross 2012), and in some cases, malicious applications which can detect emulated environments and restrain their harmful processes accordingly (Xu, Hassen and Ross 2012), in order to circumvent detection. Even by combining both methods, hybrid analysis still fails to address these issues completely. Furthermore, as malware keeps evolving to constantly find new exploits and attack vectors, many of these malicious applications will keep avoiding whichever detection strategies mobile market operators put in place.

1.3. Research questions and objectives

In this dissertation the aim is to study the effectiveness of several Machine Learning (ML) approaches in large scale mobile malware detection in the Android environment and their ability to keep up with the proliferation of new malware and its ever evolving trends.

This work is part of the AppSentinel project, which proposes to develop a cloud-based technology for Android app stores to proactively prevent malware circulation through app behavioural pattern analysis. This project also intends to test applications regarding good practices in secure mobile software development, which then can lead to educational feedback to app developers. Finally, the research and development of a supervised ML system to efficiently detect malicious applications is planned. This last part is where this dissertation is set, continuing the research and development of the supervised ML system previously tested in (Lopes 2020) where malware classifiers were trained on features obtained through static analysis. To facilitate the integration of this work in the context of this project, several contributions were made, namely in the writing of technical reports and two research papers (Duque, et al. 2020, Duque, et al. 2020) were developed for submission regarding the work described in Section 4.1 and Chapter 5, respectively.

To achieve the best possible approach of feature and algorithm selection and develop models that can distinguish malware from benign apps, several methods of data analysis, data preprocessing and ML techniques, are tested, to produce optimal classifiers trained to differentiate Android malware apps from benign ones and assess their performance in this task. This objective is divided into several research questions to better address the overall proof of concept as follows:

- How can user feedback be used for malware detection systems?
- Which user generated data is more relevant to perform malware detection using ML methods?
- Can Android app analysis tools produce relevant information to train malware detection systems?
- What should be the best approach to retrain the system to maintain high performance over time?

The first two questions are relevant to understand if there is knowledge to be extracted from the feedback users generated in an app store environment that can be used to enhance a functioning malware detection system, and to what extent is this knowledge helpful. Understanding which features to use can also be beneficial to better understand the relationship between feedback and malware.

The third research question focuses on the usage of Android app analysis tools and leveraging its results to aid malware detection. Converting the outputs these tools provide into numerical data to be statistically analysed, processed and fed into the machine learning classifiers, can provide useful insights to create more efficient models.

To answer the fourth and final question, the proposed system must be subject to testing with data from the months that follow the training data and its performance observed. This way, the feasibility of the system to adapt to new malware trends can be examined and optimized, or, if it fails to do so, what can be done to allow such adaptation.

In order to obtain a solution for these research questions 1,864 experiments were made, resulting in 126,400 algorithms trained and approximately 936 hours or 39 days of computing power.

1.4. Methodological approach

This dissertation follows the Cross-Industry Standard Process for Datamining (CRISP-DM) (Chapman, et al. 2000) methodology in order to build the classification models, . This open standard provides a robust and well-proven structured approach to designing a data mining project, from data collection and preparation, to model development and implementation. Given its powerful flexibility, practicality, and its usefulness, it is utilized in various fields and industries. As it can be seen in Figure 1, CRISP-DM defines the six core procedures that serve as guidelines to develop data mining projects: business understanding, data understanding, data preparation, modelling, evaluation and finally, deployment. This section outlines these procedures in depth, with the exception of the deployment procedure which is beyond the scope of this research.

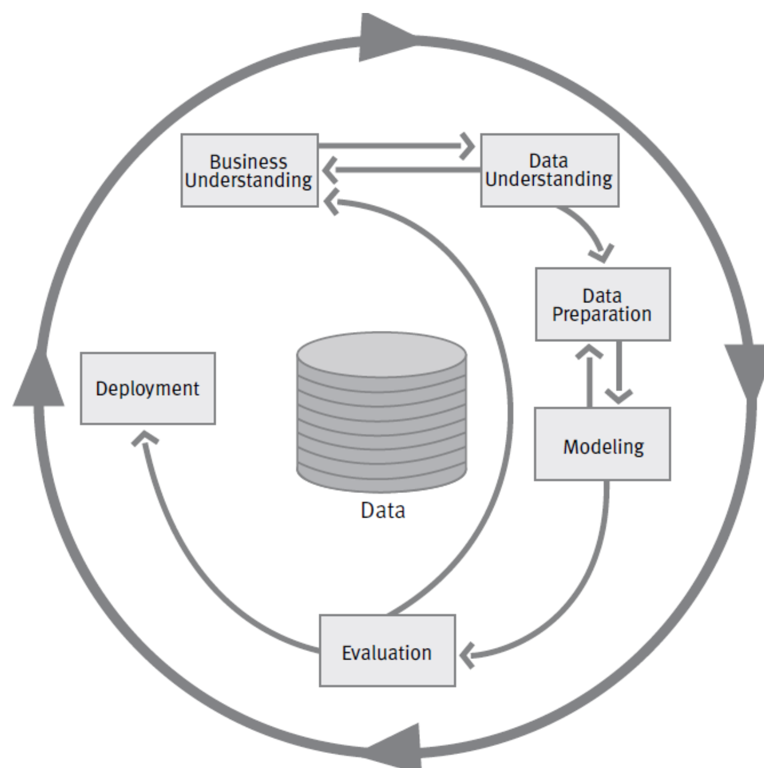

Figure 1 - Phases of the CRISP-DM reference model (Chapman, et al. 2000)

The first step in data mining projects is Business Understanding, where the focus is on leveraging the business perspective to better understand the project requirements and objectives to develop an initial plan to meet the desired objectives. Following this, the Data Understanding phase starts by collecting the data necessary to the project and

proceeds by utilizing statistical and visual data analysis techniques on the data collected to scrutinize and discover some insights that can form assumptions to better guide the succeeding steps. The Data Preparation phase encompasses the necessary procedures, such as feature selection, transformation, and data cleaning, used to build the data set that will be fed into the machine learning models. The Modeling phase is comprised of the selection and application of the chosen modelling techniques, as well as their respective parameter calibration. Finally, in the Evaluation stage the primary goal is to evaluate and review the results obtained so far and the steps taken to reach them, considering the objectives that where set before.

1.5.Structure and organization of the dissertation

This study is composed of the following 4 additional chapters:

- In **Chapter 2**, an overview of the currently defined state of the art for automated malware detection methods is presented, along with a more detailed analysis on the usage of features generated by user feedback, static code analysis and dynamic code analysis respectively.

- In **Chapter 3**, the methodology for this study is outlined, with a comprehensive analysis on the datasets utilised, data analysis, data preparation and data processing alongside an overview of the algorithms used, their implementation and the metrics by which they are evaluated.

- In **Chapter 4**, the analysis of the obtained results for the full datasets is presented according to the methodology that was deemed appropriate, joined by a broad comparison of their respective results. A small experiment analysing the impact of malware to goodware ratio is also presented.

- In **Chapter 5**, the study of the model performance decay is presented, by training the best performing model, from the previous chapter in each category, on different time frames and testing it against the following monthly periods.

- Finally, in **Chapter 6**, conclusions are drawn from this study, as well as some recommendations to overcome its limitations, alongside some possible proposals to improve its application in future works.

Chapter 2 – Literature Review

In this chapter a detailed overview of the malware detection systems currently in use is given, alongside some of the tools and processes employed, which are more impactful and useful to this study.

2.1. Machine Learning for Malware Detection

2.1.1. Large Scale Automated Detection

Large scale mobile malware detection methods currently rely on a mix of automated detection tools and manual malware detection methods. The former usually depends on anomaly-based detection schemes to detect unwanted behaviours (Google 2018) based on known patterns, while the latter usually relies on the usage of code analysis tools like Androguard (Desnos 2012), Droidbox (Lantz 2011) and Kirin (Enck, Ongtang and McDaniel 2009), among many others, to reveal malicious patterns or ascertain the level of risk an app might pose. Although manual detection methods are used to detect novel malware variations and distinguish between real malicious apps and poorly developed ones (greyware), they are rather resource intensive and, in most cases, still demand the intervention of a security analyst (Enck and McDaniel 2010). Therefore, to maintain a scalable malware detection framework, App stores have the need to continuosly develop and fine tune these automated methods, so that they can adapt to the ever-emerging malware trends.

2.1.2. Regarding Data Collection

To effectively train ML algorithms, large amounts of data are usually required. Most of the works on the subject resort to application repositories (Arp, et al. 2014, Chakradeo, et al. 2013, Deo, et al. 2016) , or build the datasets indiscriminately with the help of store scrapers and Antivirus products (Roy, et al. 2015, Singh, Walenstein and Lakhotia 2012, Allix, et al. 2014). Although app repositories usually provide labeled datasets of known benign and malicious apps, alongside somewhat large and complex feature vectors for each app, they are built upon previously detected applications that are sometimes several years old or are not updated to take into account new malware that had previously evaded

detection. While the market downloaded apps do not suffer as much from the old data predicament, they cannot be reliably labeled with Antivirus products since most known malware has already been filtered out by the app store and/or the Antivirus products haven't yet caught up with the most recent malware trends. Nonetheless, these are the options available to build malware datasets.

ML approaches rely heavily on the quality of the input data, therefore, making the best use of the available data is essential. However, many obstacles arise when composing a train/test dataset that can maximise algorithm performance while avoiding biases and other pitfalls. As mentioned before, a classifier that learns from a dataset that contains dated apps can lead to inferior results when applied in an up to date realistic scenario.

Another issue that comes up frequently is the malware to goodware ratio. Most works either try to mimic a realistic less than 1% of malicious apps (Google 2018) in their data set (Sahs and Khan 2012, Peiravian and Zhu 2013) or try to maintain an equal ratio of malware to goodware (1:1) (Roy, et al. 2015, Deo, et al. 2016, Chen, et al. 2016) . Both approaches have diferent pitfalls. The former, while allowing training and testing the models in a more realistic environment can lead to a goodware classification bias, due to the highly imbalanced dataset. Meanwhile, the latter can avoid this issue by balancing both classes, however, this can lead to a misrepresentation in a real-world application. Lastly, the data itself can have noise built in due to the nature of malware, as there are malicious apps that have managed to evade detection (e.g., adware) as well as benign apps that have security flaws and might be considered potentially harmful apps (PHA).

2.2. User Feedback Analysis

Very little has been done with user feedback regarding malware detection. For example, WHYPER (Pandita, et al. 2013), focused on processing app market metadata, such as application descriptions, to examine whether the description provided any indication as to why the application needed certain permissions. Nonetheless, parallels can be drawn from other uses in the customer feedback analysis domain such as online reviews for hotels (Antonio, et al. 2018), restaurants (Kiritchenko, et al. 2014), and e-commerce providers (Kiritchenko, et al. 2014). With app store users being able to post their feedback regarding their downloaded apps, via ratings, comments, and other sorts

of flags, this can potentially be used to help malware detection methods by providing more features to be analysed.

Although structured information, like ratings and flags, can easily be added as features to be used by machine learning algorithms, unstructured information like comments need to be processed first. This can be achieved through Natural Language Processing (NLP) techniques like sentiment analysis (Eshleman and Yang 2014, Forte and Brazdil 2016), and opinion mining (Petz, et al. 2013). However, literature on non-social media complaint analysis is considerably scarce, mainly due to the fact that such data is typically not publicly available (Filgueiras, et al. 2019). In (Ordenes, et al. 2014), a framework is proposed to analyse customer experience feedback, using a linguistics-based model, by identifying activities resources and context, to automatically distinguish compliments from complaints.

Traditional approaches to text categorization employ feature-based sparse models, using bag-of-words and Term Frequency-Inverse Document Frequency (TF-IDF) encoding (Filgueiras, et al. 2019). More recent techniques, such as word embeddings (Mikolov, et al. 2013) and recurrent neural networks (RNN) (Elman 1990), have also been used in complaint classification.

2.3. Static Analysis

Static analysis is a detection method which consists of examining a program's code without its execution. In the case of the Android environment this analysis also takes into consideration other components that go beyond the code itself, most notably the AndroidManifest file, which allows for a more thorough examination. Theoretically, this method can unveil every possible execution path, however, it suffers from several drawbacks (Tam, et al. 2017). The major drawback is the vulnerability to obfuscation techniques, that remove or limit code access (Moser, Kruegel and Kirda 2007), such as string encryption or renaming methods and variables.

Other drawbacks include the injection of non-Java code, network activity, and the modification of objects at runtime which are outside the scope of static analysis as they are only visible during execution (Tam, et al. 2017). Alongside these vulnerabilities is also the fact that free alternative code compilers mean that signature-based methods are incompatible with android. Therefore, most android static analysis either focuses on the

android package (APK) bytecode, such as DroidMOSS (Zhou, et al. 2012), that uses fuzzy hashing to leverage small fingerprints from the extracted instruction sequences therefore localizing altered code, or its APK components, such as the AndroidManifest.xml, which states permissions, package name, version, referenced libraries and app components, like Droidmat (Wu, et al. 2012) and PUMA (Sanz, et al. 2013) that leverage machine learning algorithms to classify apps by their permissions. And finally, the APK classes.dex file, which contains all Android classes compiled into a dex file format, compatible with the Dalvik virtual machine.

2.4. Dynamic Analysis

In contrast to static analysis, dynamic analysis executes a program and observes the results (Tam, et al. 2017). Its main downside is the limited code coverage, since only one path can be followed each time. However, this can be mitigated by exploiting multiple execution paths (Brumley, et al. 2007, Chipounov, Kuznetsov and Candea 2011, Moser, Kruegel and Kirda 2007). Given that android apps are designed for user interaction, user behaviours need to be emulated via the interface, received intents or with automatic event injectors (Azim and Neamtiu 2013, Machiry, Tahiliani and Naik 2013, Mahmood, Mirzaei and Malek 2014). For example, in order to stimulate applications, DynoDroid (Machiry, Tahiliani and Naik 2013) was developed to simulate real user interactions from collected user data, such as screen tapping, long pressing and dragging, to find bugs in Android apps.

Since the app is running during analysis, many features can be gathered from different architectural layers (e.g., hardware, kernel, app, or OS) to examine its behaviour. However, since malware is running as well, it can tamper with the analysis or even supress its malicious behaviour if it detects an emulated environment. While in-the-box analysis gathers data on the same privilege level as the malware, meaning that it can access memory structures and high OS-level data easily, it is vulnerable to being attacked or bypassed, with native code (e.g., non-Java Code compiled to run with an Android Central Processing Unit (CPU)) or reflection (e.g., modifying methods, classes and interfaces during runtime) (Xu, Hassen and Ross 2012). Meanwhile, out-of-the-box analysis, like DroidScope (Yan and Yin 2012) and CopperDroid (Tam, Khan, et al. 2015), manage to emulate android through a VM, and so, are able to provide complete control and oversight

of the Android environment. However, malware can counter emulation by detecting false, non-real environments and alter its behaviour in order to evade analysis.

2.5. Hybrid Analysis

By combining static and dynamic analysis, hybrid methods can increase robustness, monitor edited apps, increase code coverage, and find vulnerabilities (Tam, et al. 2017). By implementing both methods sequentially, certain drawbacks can be limited. For example, SmartDroid (Zheng, et al. 2012), EvoDroid (Mahmood, Mirzaei and Malek 2014) and in (Spreitzenbarth, Freiling and Echtler, et al. 2013), the authors managed to increase code coverage by using static analysis to find all possible activity paths in order to guide dynamic analysis through them. Other detectors, like (Bläsing, et al. 2010), use static analysis to estimate the app's risk before dynamically logging its system calls with kernel-level sandboxing.

Alternatively, by collecting features through static and dynamic analysis, machine learning algorithms can be trained to detect malware with a large enough dataset. In (Wang, Qiu and Zhao 2018), several machine learning classifiers were trained on a dataset composed of a binary feature vector for each app, where features were extracted using various forms: statically through the apps permission system and API calls, with reverse-engineering tools Baksmali (JesusFreke 2009) and Androguard (Desnos 2012); dynamically, through virtualization, with an automated test tool called monkey (Android Developers 2016); and through malicious behaviour monitoring, using DroidBox (Lantz 2011). Similarly, in (Liu, et al. 2016), machine learning algorithms are also trained on a binary feature vector created for each app. The main difference being the employment of the Android Debug Bridge (ADB) to execute apps on the device while connected to a computer, instead of executing them on a virtualized environment.

2.6. Closing Remarks

Given the vast research done on this subject, the works presented in this section were selected as an overall representation of the most notable features and drawbacks in each category. This work does not intend to address the detailed intricacies of the various analysis methods but rather the use of their respective results to train ML algorithms in the most effective manner to better classify malicious applications, and afterwards test

their effectiveness in retaining classification performance over time as new data becomes available. Table 1 presents a systematization of the noteworthy works presented above.

Table 1 – Literature Review Summary

Study	Analysis	Detector	Sample Source	Sample Size	Malware Ratio	Method	Summary
(Arp, et al. 2014)	X	X	Google Play and MalGenome	129,013	4%	Static	The DREBIN framework leverages features obtained from static code analysis and the manifest file to train a SVM classifier which detects 93.9% of the malware samples with a false positive rate of 1%.
(Chakradeo , et al. 2013)		X	Google Play, Contagio and MalGenome	15,620	5%	Static	The MAST framework utilises Multiple Correspondence Analysis (MCA) to rank applications by their potential to exhibit malicious behaviour.
(Deo, et al. 2016)		X	Marvin, McAfee, MalGenome and Drebin	124,190	50%	Static	This framework proposes the use of Venn-Abers predictors for assessing the quality of binary classification tasks to identify antiquated models.
(Roy, et al. 2015)	X		Google Play, VirusShare and Arbor Networks	1,019,000	Several	Static	This study tested several differing experimentations regarding the use of ML classifiers to detect malware based on static analysis.
(Singh, Walenstein and Lakhotia 2012)	X		Unnamed AV Company	4,173	100%	Static	This study tries to track malware concept drift through similarity of byte 2-grams and mnemonic 2-grams.
(Allix, et al. 2014)		X	Google Play, Appchina, 1Mobile	206,237	30%	Static	This study demonstrates the relevance of historic coherence in the selection of datasets.
(Sahs and Khan 2012)		X	Unnamed Source	2,172	4%	Static	This framework demonstrates the usage of machine learning-based malware detection systems for the Android operating system. By using Permissions and Control Flow Graphs as features to train a SVM model.
(Peiravian and Zhu 2013)		X	Unnamed Source	1,860	32%	Static	In this study the authors used APKs Permissions and API calls to train three different machine learning classifiers: SVM, Decision Trees and Bagging. And compared results on the algorithms trained with different feature combinations.
(Chen, et al. 2016)		X	MalGenome, Mobile-Sandbox	6,000	50%	Static	This study used syntax-based and semantics-based features to train several classifiers, reporting na improved robustness to classifier performance when semantics-based features are incorporated in training as compared to syntax-based features.
(Pandita, et al. 2013)	X		Google Play	581	Unknown	NLP	WHYPER focuses on persmissions for a given app and inspects whether or not the app's description provides details as to why the app needs the permission.
(Wu, et al. 2012)		X	Google Play and Contagio	1,738	13%	Static	DroidMat studies the impact of different feature compositions on the model performance of two classifiers, KNN and NaïveBayes.
(Sanz, et al. 2013)		X	Google Play and VirusTotal	606	40%	Static	The PUMA framework focuses on the usage of permissions to train several different machine learning classifiers.

Table 1 (Continued) – Literature Review Summary

Study	Analysis	Detector	Sample Source	Sample Size	Malware Ratio	Method	Summary
(Brumley, et al. 2007)	X		Unnamed Source	-	-	Dynamic	The BitScope framework prosposes several techniques to extract behavioural information through an emulated environment.
(Chipounov, Kuznetsov and Candea 2011)	X		-	-	-	Dynamic	The S2E platform performs *in-vivo multipath analysis* of systems through the combination of virtualization, dynamic binary translation and symbolic execution to perform a behaviour analysis.
(Moser, Kruegel and Kirda 2007)	X		Unnamed AV Company	308	100%	Dynamic	This study presents a system that explores multiple execution paths by letting the program fully execute each path and reverting to checkpoints placed along the way.
(Azim and Neamtiu 2013)	X		Google Play	25	-	Dynamic	The A3E framework presents two novel approaches to app exploration, Targeted Exploration and Depth first that focus on the events triggered during GUI exploration.
(Machiry, Tahiliani and Naik 2013)	X		Google Play	50	-	Dynamic	The Dynodroid framework presents a practical system for generating relevant inputs on mobile apps with a novel "observe-select-execute" approach, significantly automating task testing.
(Xu, Hassen and Ross 2012)	X		Lisvid	3,189	-	Dynamic	The Aurasium framework is able to detect attempts by multiple applications to collaborate and implement a malicious logic on critical resources.
(Yan and Yin 2012)	X		Unnamed Source	2	100%	Dynamic	DroidScope is a dynamic binary instrumentation tool that rebils two levels of semantic information: OS and Java. API tracing, native instruction tracing, Dalvik instruction tracing and taint tracking are already core components.
(Tam, Khan, et al. 2015)	X		Contagio, MalGenome and McAfee	2,986	100%	Dynamic	The CopperDroid framework uses VMI-based dynamic system call-centric analysis to describe the application behavior

Table 1 (Continued) – Literature Review Summary

Study	Analysis	Detector	Sample Source	Sample Size	Malware Ratio	Method	Summary
(Zheng, et al. 2012)	X	X	Unnamed Source	7	100%	Hybrid	SmartDroid is a framework that combines static and dynamic analysis to automatically reveal UI-based trigger conditions. It uses static analysis to build Call Graphs in order to guide the dynamic analysis towards the sensitive APIs.
(Mahmood, Mirzaei and Malek 2014)	X	X	F-Droid	110	-	Hybrid	Evodroid presents a novel evolutionary testing technique that preserves and promotes the genetic makeup of individuals in the automated testing search process.
(Spreitzenbarth, Freiling and Echtler, et al. 2013)	X	X	Google Play, VirusTotal and other Unnamed sources	183,500	>1%	Hybrid	This study proposes the usage of static and dynamic analysis to detect malicious behaviour. Most notably logging all performed actions including those stemming from native API calls.
(Bläsing, et al. 2010)	X	X	Google Play	151	>1%	Hybrid	The AASandbox framework uses both static and dynamic analysis to automatically detect suspicious applications. Static analysis scans the package for malicious patterns without installing it, while the dynamic analysis implementation is placed in kernel space and hijacks system calls for further analysis.
(Liu, et al. 2016)	X	X	Wandoujia and MalGenome	1,000	50%	Hybrid	This study proposes a hybrid scheme that submits applications through both analysis methods and builds several classifiers with the features extracted from both methods.

Chapter 3 – Methodology

The focus of this chapter is on describing the framework designed to test the proof of concept of malware detection using different types of data to train the algorithms. Figure 2 shows the framework layout. The main objective is to show that it is possible to detect malware apps using features extracted from user feedback data, static code analysis, and dynamic code analysis.

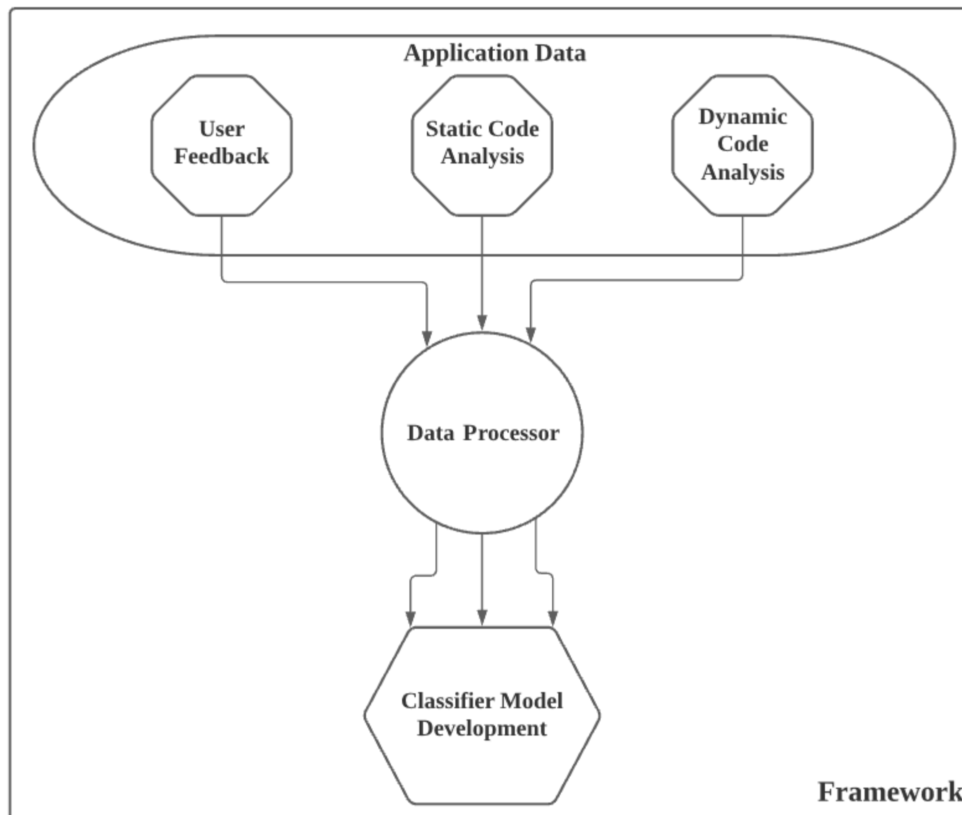

Figure 2 - Prpposed Framework Layout

This framework is then divided into those three sections respectively, using the same pipeline for each. This framework was developed in the continuation of the AppSentinel project, in (Lopes 2020) a prototypical pipeline was already tested for its potential use regarding the usage of features obtained through static code analysis to train machine learning classifiers, obtaining similar results to those in Chapter 4.2.

The Data Processor segment is composed of the various transformative and preparatory techniques applied onto the various datasets, before being fed into the Classifier Model Development segment. Here the various machine learning classifiers are trained on the previously

prepared data and the best performing ones are select for comparison. These segments are detailed in Chapter 3.

Another study followed in order to test the ability of the models generated by this framework to adapt to temporal changes in the malware patterns, by training them with different sizes of historically coherent datasets and testing them on the months that followed those same datasets. This way an analysis was made on the effectiveness of these models to adapt in an unforeseen environment with data from "future" sets relative to those used in training and validation. This procedure and its results are detailed in Chapter 5.

3.1. Datasets

Here the complete datasets for each of the categories are examined. Their feature composition further inspected in order to maintain the most significant ones and remove the ones that are less contributive to the overall system performance. This is achieved by means of correlation analysis between the features. This helps ensure that the models are not being trained on redundant and unneeded information, therefore, reducing the overall time required to train them as well as increasing correct detection rate.

As mentioned before, in (Lopes 2020) the possibility of using features obtained through static analysis to train machine learning classifiers to detect malicious applications had already been tested, however, in order to test the reproducibility and replicability of his methods and validate his findings, it was decided to repeat the same methods using a similar dataset to train the models. The results in Chapter 4.2 show similar findings to those in (Lopes 2020), proving that his findings contribute to the scientific knowledge in this field.

3.1.1. User Feedback Dataset

The first dataset used was the user feedback data. This historically coherent dataset originated from Aptoide's repository and is comprised of 2332 applications and their respective feedback, from the beginning of October 2019 to the end of January 2020. Table 2 displays a more detailed breakdown.

Table 2 – User Feedback Dataset sample distribution

	2019			2020	Average
	October	November	December	January	
Number of samples	586	141	450	1155	583
Malware samples	49	64	79	396	147
Goodware samples	537	77	371	759	436
Malware Ratio	8%	45%	18%	34%	26%

This dataset was constructed from apps submitted to the Aptoide's app store during the mentioned time period, and each app was labelled with a target classification of Trusted or Critical, indicating the classification of Goodware or Malware respectively, which was given by Aptoide's internal security audit.

Besides the target feature, this dataset included the following other features detailed in Table 3.

Table 3 – User Feedback Dataset Feature description

Feature	Description
MD5	Application MD5 Checksum
Package	Android Package which the application belongs to
Date	Date when the application was analysed and classified as Goodware or Malware
1 Star Rating	Number of 1 Star Ratings
2 Star Rating	Number of 2 Star Ratings
3 Star Rating	Number of 3 Star Ratings
4 Star Rating	Number of 4 Star Ratings
5 Star Rating	Number of 5 Star Ratings
Good Flag	Number of "Good" Flags users gave
Virus Flag	Number of "Virus" Flags users gave
Fake Flag	Number of "Fake" Flags users gave
License Flag	Number of "Needs License" Flags users gave
Comments	All the comments users gave to the application
Classification	Target application label - Trusted or Critical

This dataset has the distinctive characteristic of allowing to observe the feedback users gave to malicious apps before these were detected and swiftly removed from the app store. With this twist, the machine learning classifiers have the possibility of detecting certain patterns that might allow for malware classification. Because nonnumerical features like MD5 and Package serve only as

identifiers, they were removed. The date feature was only useful into dividing the apps into monthly groups, so it was removed as well. Finally, due to being out of the scope of this study, the comments were removed as well.

Table 4 shows the statistical distribution of this data set. A few things are worth noting. Firstly, that each application has at least one value for each feature, demonstrated by the count value being equal to the number of apps. Secondly, the mean represents the mean value for each feature across the whole dataset. Globally, there was an average of 1046 ratings and 9 flags per application. Thirdly, the standard deviation represents the value dispersion or by how much does each feature vary from the mean. Fourthly, the percentiles represent the number of observations that can be found under each percentage, with 50% being the median. Finally, the minimum and maximum values represent their respective equivalents in each feature, for example the 5 Star Rating has a maximum value of 353,411 which means that the maximum number of 5 Star Ratings an application received was 353,411.

Table 4 – User Feedback Dataset Feature statistics

Features Statistics										
Feature	Star Ratings					Flags				Target
	1	2	3	4	5	Good	Virus	Fake	License	Classification
Count	2332	2332	2332	2332	2332	2332	2332	2332	2332	2332
Mean	554.5	151.2	1,121.3	609.1	5,282.7	8.7	3.9	3.8	11.7	0.25
Standard Deviation	4,404.8	1,069.2	8,088	3,399.5	37,987.3	56.4	19.3	22.8	140.7	0.43
Minimum	0	0	0	0	0	0	0	0	0	0
25%	0	0	0	0	1	0	1	0	0	0
50%	2	0	2	1	10	1	1	0	0	0
75%	38	10	63	40.2	361	4.2	2	2	1	1
Max	37,049	9,859	74,968	29,994	353,411	1,332	618	638	4125	1

Furthermore, we can see a detailed statistical feature distribution which reveals a high degree of deviation from the mean alongside varying maximum values. These are explained by the fact that a small percentage of apps reach high levels of popularity, therefore, receive more user feedback, wether it be favourable or not. A certain level of correlation can be observed between all ratings, the value ranges for each of the feature groups are similar. To further examine this Table 5 presents a correlation matrix.

Table 5 – User Feedback Dataset Feature correlation matrix

Correlation Matrix										
Features		Star Ratings					Flags			
		1	2	3	4	5	Good	Virus	Fake	License
Star Ratings	1	1.000	0.952	0.850	0.813	0.882	0.165	0.085	0.021	0.114
	2	0.952	1.000	0.860	0.908	0.895	0.130	0.062	-0.005	0.056
	3	0.850	0.860	1.000	0.664	0.961	0.170	0.068	0.000	0.073
	4	0.813	0.908	0.664	1.000	0.755	0.059	0.040	-0.018	0.051
	5	0.882	0.895	0.961	0.755	1.000	0.165	0.070	-0.006	0.063
Flags	Good	0.165	0.130	0.170	0.059	0.165	1.000	0.622	0.588	0.536
	Virus	0.085	0.062	0.068	0.040	0.070	0.622	1.000	0.653	0.409
	Fake	0.021	-0.005	0.000	-0.018	-0.006	0.588	0.653	1.000	0.404
	License	0.114	0.056	0.073	0.051	0.063	0.536	0.409	0.404	1.000

Table 5 shows that the numeric ratings have a high degree of correlation between themselves. The same can be said to a lesser extent about the symbolic flags. Although this presents itself as redundant information, their relation to the opposing feature group shows very low levels of correlation, therefore, allowing the models to interpret this as relevant information. This will be demonstrated in the exploratory data analysis sub chapter.

3.1.2. Static Analysis Dataset

The second dataset utilized contained the static code analysis results of 131,429 applications by analysing their AndroidManifest.xml and DEX files with the Androguard (Desnos 2012) tool, which contain essential information about the app to the Android build tools and the Android OS. Among many attributes, the manifest file declares the following: the app's package name and md5 checksum as identifiers; app components, which include all activities, services, broadcast receivers and content providers, device configurations it can handle and intent filters; the permissions that the app needs; the hardware and software features the app requires.

This dataset contains apps from the beginning of October 2019 to the end of March 2020. Table 6 displays a more detailed breakdown. Much like the previous dataset the applications were labelled with a target classification of Trusted or Critical, indicating the Goodware or Malware

classification respectively, which was given by Aptoide's internal security audit. Besides this target feature the dataset also included the following other features detailed in Table 7.

Table 6 – Static Analysis Dataset sample distribution

	2019			2020			Average
	October	November	December	January	February	March	
Number of samples	21,934	24,158	32,757	13,923	16,170	22,487	21,905
Malware samples	9,889	10,524	14,918	5,550	5,558	8,409	9,141
Goodware samples	12,045	13,634	17,839	8,373	10,612	14,078	12,764
Malware Ratio	45%	44%	46%	40%	34%	37%	41%

This dataset allows for a statistical analysis of the app's resources usage and requirements. Because the MD5 checksum serves only as an identifier, this feature was removed. The rest of the features represent the numerical representation of each occurring feature to use as model inputs.

Table 7 – Static Analysis Dataset Feature description

Feature	Description
MD5	Application MD5 Checksum
Time	Time it took to analyse the app's files
Size	Size of the application in bytes
Permissions	Number of permissions the app needs in order to run
Activities	Number of activity components the app can execute
Services	Number of Services. Long operations usually run in the background
Receivers	Number of android-name attributes of all receivers
Opcodes	Number of Dalvik specific opcodes
Res. Strings	Number of additional resource files
Smali Strings	Number of smali (non-Java code) strings
API Package	Number of API classes the app needs to run
System Commands	Number of System commands the app executes
Intents	Number of intent messages to activate activities, services, or receivers
Classification	Target application label - Trusted or Critical

Table 8 shows a detailed statistical distribution of each feature from this data set. This set contained 34,172 malware labelled applications, which represented approximately 26% of the total dataset. Similarly to the previous dataset, the maximum values of every feature far exceeded the mean values, usually by one or two orders of magnitude. This value disparity can cause certain algorithms to learn incorrect weights due to big value differences, this will be addressed in section 3.3.

Table 8 – Static Analysis Dataset Feature statistics

SD – Standard Deviation, Min – Minimum, Max - Maximum

Features Statistics

Feature	Permissions	Activities	Services	Receivers	Opcodes	Resource Strings	Smali Strings	API Packages	Sys cmd	Intents
Count	131429	131429	131429	131429	131429	131429	131429	131429	131429	131429
Mean	15	33.2	10.3	7.1	563374	546.3	30181	148515	135.8	16.7
SD	17.2	67.9	18.3	9.5	265373	1047.6	30042	109331	156.6	46.4
Min	0	0	0	0	0	0	0	0	0	0
25%	6	6	2	1	398063	70	12328	77538	41	3
50%	11	15	6	4	609000	168	21135	129317	78	8
75%	18	37	13	9	756225	552	41301	199498	186	18
Max	326	6198	488	148	3614313	65510	738219	2265619	4677	5031

Table 9 shows the correlation matrix for the static analysis dataset, to better understand the relationship between each feature and their correlations. In it, it is shown that this set of features presents low correlations values with the exceptions of certain functions like services, activities, and receivers, as well as code execution like smali Strings, API packages, and system commands. This is possibly due to the function group they belong to, leading to similar usage. So far none of these features present themselves as redundant information, and therefore, are used as input to train and validate the models.

Table 9 – Static Analysis Dataset Feature correlation matrix

Correlation Matrix

	Permissions	Activities	Services	Receivers	Opcodes	Resource Strings	Smali Strings	API Packages	System cmd	Intents
Permissions	1.00	0.50	0.69	0.64	0.24	0.40	0.42	0.40	0.35	0.45
Activities	0.50	1.00	0.62	0.60	0.19	0.44	0.43	0.42	0.41	0.29
Services	0.69	0.62	1.00	0.79	0.24	0.37	0.42	0.40	0.36	0.49
Receivers	0.64	0.60	0.79	1.00	0.31	0.48	0.52	0.48	0.46	0.47
Opcodes	0.24	0.19	0.24	0.31	1.00	0.29	0.53	0.61	0.47	0.16
Res Strings	0.40	0.44	0.37	0.48	0.29	1.00	0.63	0.60	0.51	0.38
Smali Strings	0.42	0.43	0.42	0.52	0.53	0.63	1.00	0.89	0.87	0.38
API Packages	0.40	0.42	0.40	0.48	0.61	0.60	0.89	1.00	0.83	0.36
System cmd	0.35	0.41	0.36	0.46	0.47	0.51	0.87	0.83	1.00	0.29
Intents	0.45	0.29	0.49	0.47	0.16	0.38	0.38	0.36	0.29	1.00

3.1.3. Dynamic Analysis Dataset

The third and final dataset utilized contained the dynamic code analysis results of 4866 applications by analysing their runtime behaviour and network usage with the Droidbox (Lantz 2011) and CuckooDroid (Revivo and Caspi 2014) tools alongside static code analysis results from Androguard (Desnos 2012) previously described. The dynamic analysis aspect of this dataset allows for a better understanding of malware behaviour by analysing the network traffic generated and received, file and camera accessed and even cryptographic operations through the Android API. These behaviour components were examined during the app's runtime through the usage of an emulator and extracted to the host machine with the Android Debug Bridge (ADB).

This dataset contains apps from the beginning of February 2020 to the end of July 2020, encompassing a total of 6 months of app report data. Table 10 displays a more detailed breakdown. Like the previous two datasets the applications were labelled as being detected malware or not, but unlike the previous two, this dataset was semi-randomly collected from the Koodous (Koodous 2018) platform during that time frame, where the apps are analysed with the tools mentioned and expert malware analysts review the analysis results and vote to decide wether the applications are malware or not. Table 11 displays each feature and their function.

Table 10 – Dynamic Analysis Dataset sample distribution

	2020						Average
	February	March	April	May	June	July	
Number of samples	347	972	1,093	1,100	900	453	811
Malware samples	176	544	519	550	450	228	411
Goodware samples	171	428	574	550	450	225	400
Malware Ratio	51%	56%	47%	50%	50%	50%	51%

Table 11 – Dynamic Analysis Dataset Feature description

Tool	Feature	Description
Androguard	Displayed Version	Android version code
	Target SDK Version	Recommended Android version
	Min SDK Version	Minimum Android version required
	Providers	Number of content providers that manage access the central data repository
	New Permissions	Number of new permissions the app requested in order to run
	Filters	Number of intent filters that decide the type of intents the components would like to receive
	Activities	Number of activity components the app can execute
	Receivers	Number of android-name attributes of all receivers
	Services	Number of Services. Long operations usually run in the background
	Permissions	Number of new permissions the app requests in order to run
	URLs	Number of URL links in the application
	Ads	Number of Advertisement components
	Installed Apps	Number of searches for installed applications
	Serial No	Serial number
	MCC	Number of Mobile Country Code methods, used to identify network operators
	SMS	Number of SMS methods used
	Phonecall	Number of Phonecall methods used
	Crypto	Number of cryptographic operatios used (such as encryption or key generation)
	SSL	Number of SSL connections
	Camera	Number of Camera methods used
	Dynamic Broadcast Receiver	Number of event triggers
	IMEI	Number of Get device ID Key events
	Sensor	Number of Sensor events
	Run Binary	Number of binaries used
CukooDroid	Socket	Number of sockets used
	HTTP	Number of HTTP connections established
	Hosts	Number of Host adresses
	DNS	Number of DNS requests
	Domains	Number of domains accessed

Table 11 (Continued) – Dynamic Analysis Dataset Feature description

Tool	Feature	Description
DroidBox	Domains	Number of domains accessed
	Files Written	Number of files written
	Files Read	Number of files read
	Crypto	Number of Crypto methods used
	Service Start	Number of services started
	Libraries	Number of libraries used
	Dexclass	Number of processes for dexclass
	Send Net	Number of network operations sent (HTTP, POST, GET, etc..)
	Receive Net	Number of network operations received (HTTP, POST, GET, etc..)
Target	Classification	Target application label - Malware Detected or Not Detected

This extensive feature composition further allows the detection of more complex patterns. Due to the complex issue of analysing string-based features, like domains accessed, libraries used, and files read, most of these were turned into string or word counts to facilitate a more statistical analysis approach, this process is further detailed in section 3.3. Other features were removed due to serving only as app identifiers, such as SHA256 and MD5 checksums. Lastly others were dropped because they did not contain any information at all due to being empty for most apps.

Table 9 shows the featurewise statistical analysis of this dataset. Here a detailed statistical feature distribution can be observed. Similarly to the previous datasets' statistical distributions, this one also shows very low standard deviations joined by low maximum values. This is due to the fact that each application doesn't need to use all features available in the Android OS, such as the camera, sending text messages or making phonecalls. One particularity is the Android displayed version which contains some odd but still valid versions. Figure 3 shows the correlation between the aforementioned features, due to the elevated number of features a heatmap represented an easier visual medium to represent their correlation matrix. These features show for the most part low levels of correlation between themselves, with a few exceptions, most notably between features related to network associated actions, such as HTTP connections and Host addresses, and associations between cryptographic methods and Secure Socket Layer (SSL) connections. Although some of these features reach very high levels of correlation, they were maintained due to their varying levels of correlation with the other features.

Table 12 – Dynamic Analysis Dataset Feature statistics

SD – Standard Deviation

Androguard Features Statistics								
Feature	Displayed Version	Target SDK Version	Min SDK Version	Providers	New Permissions	Filters	Activities	Receivers
Count	4,866	4,866	4,866	4,866	4,866	4,866	4,866	4,866
Mean	51,153	22.24	13.39	1.03	0.17	11.74	12.62	5.87
SD	2,913,813	6.99	6.37	2.2	0.59	10	15.43	3.87
Min	0	0	0	0	0	0	0	0
25%	1.1	22	8	0	0	3	8	2
50%	1.2	22	15	0	0	16	10	8
75%	2.1	28	17	1	0	16	13	8
Max	201,809,190	30	29	22	10	135	380	40

Table 12 (Continued) – Dynamic Analysis Dataset Feature statistics

Androguard Features Statistics								
Feature	Services	Permissions	URLs	Ads	Installed Apps	Serial No.	MCC	SMS
Count	4866	4866	4866	4866	4866	4866	4866	4866
Mean	4.88	13.39	91.5	0.86	0.19	0.04	2.16	0.19
SD	4.16	9.32	216.54	1.35	0.53	0.23	1.24	0.71
Min	0	0	0	0	0	0	0	0
25%	3	8	43	0	0	0	1	0
50%	5	13	47	0	0	0	3	0
75%	5	15	66	3	0	0	3	0
Max	92	76	1667	6	3	3	7	11

Table 12 (Continued) – Dynamic Analysis Dataset Feature statistics

Androguard Features Statistics								
Feature	Phonecall	Crypto	SSL	Camera	Dynamic Broadcast Receiver	IMEI	Sensor	Run Binary
Count	4866	4866	4866	4866	4866	4866	4866	4866
Mean	0.07	2.68	2.72	0.46	2.76	2.6	0.03	2.6
SD	0.33	0.99	0.86	1.19	0.85	1.05	0.23	1.09
Min	0	0	0	0	0	0	0	0
25%	0	3	3	0	3	3	0	3
50%	0	3	3	0	3	3	0	3
75%	0	3	3	0	3	3	0	3
Max	4	9	6	10	7	8	3	11

Table 12 (Continued) – Dynamic Analysis Dataset Feature statistics

CuckooDroid Features Statistics					
Feature	Socket	HTTP	Hosts	DNS	Domains
Count	4866	4866	4866	4866	4866
Mean	2.77	0.05	0.08	0.74	0.08
SD	0.78	0.7	0.47	4.98	0.48
Min	0	0	0	0	0
25%	3	0	0	0	0
50%	3	0	0	0	0
75%	3	0	0	0	0
Max	6	21	11	116	9

Table 12 (Continued) – Dynamic Analysis Dataset Feature statistics

Droidbox Feature Statistics									
Feature	Domains	Files Written	Files Read	Crypto	Service Start	Libraries	Dexclass	Send Net	Receive Net
Count	4866	4866	4866	4866	4866	4866	4866	4866	4866
Mean	0.05	1	1.34	1.85	0.01	0.02	0.3	0.05	0.1
SD	0.38	4.27	3.83	12.86	0.23	0.32	0.52	0.68	1.58
Min	0	0	0	0	0	0	0	0	0
25%	0	0	0	0	0	0	0	0	0
50%	0	0	0	0	0	0	0	0	0
75%	0	1	3	0	0	0	1	0	0
Max	11	44	91	130	10	12	6	20	50

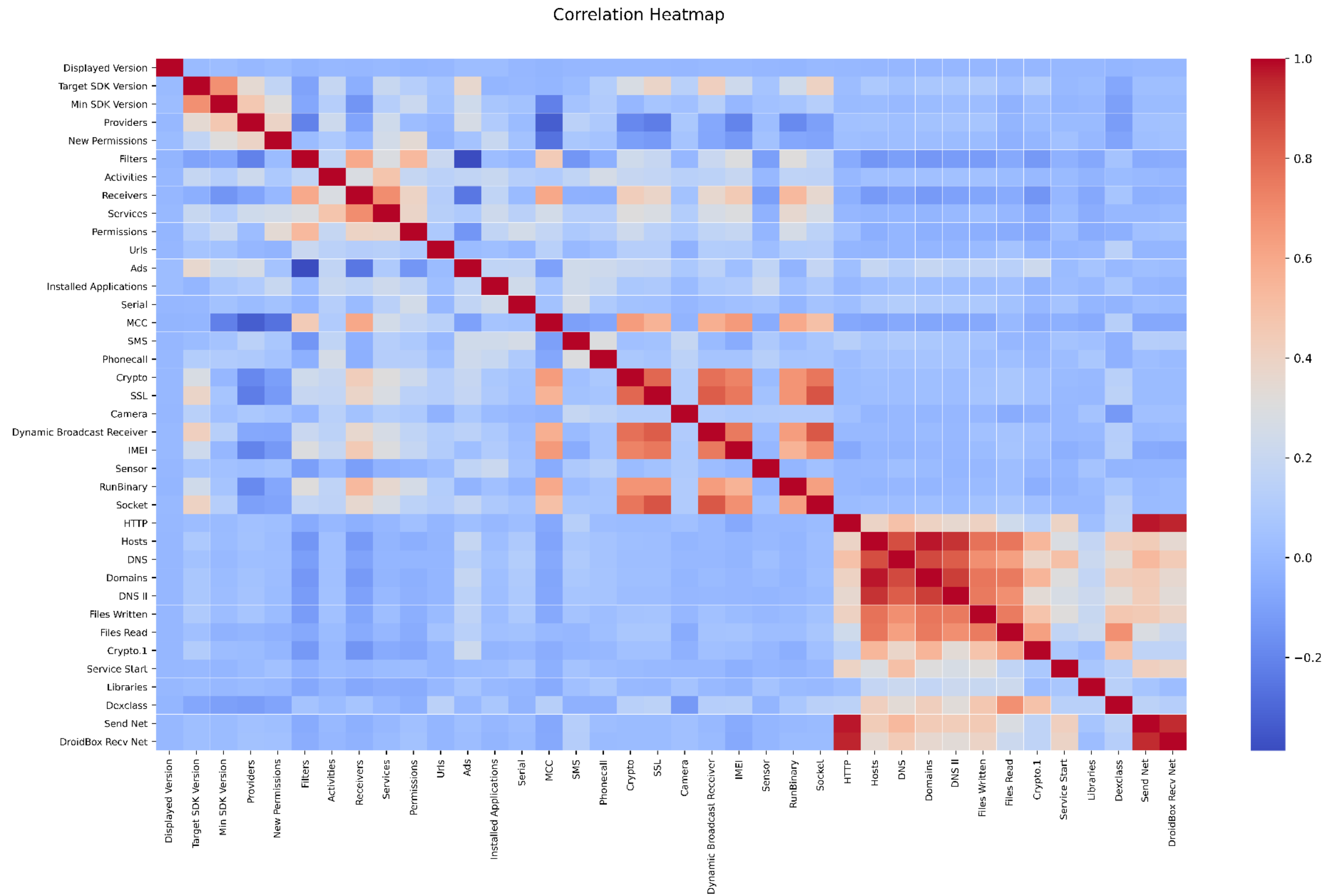

Figure 3 - Dynamic Analysis Dataset Feature correlation heatmap

3.2. Exploratory Data Analysis

This section focuses on describing the exploratory data analysis procedures that help uncover useful patterns in the data samples. According to (Tukey 1962) these methods are primarily of a visual nature in order to make analysis easier, more precise or more accurate. This section is divided into two parts which represent two different dimensionality reduction approaches: Principal Component Analysis (PCA); and T-Distributed Stochastic Neighbor Embedding (t-SNE) analysis. For these approaches, the programming language Python (Rossum 1995) was used in conjunction with Matplotlib (Hunter 2007) and Scikit-learn (Buitinck, et al. 2013) libraries.

3.2.1. Principal Component Analysis

Due to the large nature of the datasets used, a commonly applied technique to interpret them is Principal Component Analysis (PCA) (Wold, Esbensen and Geladi 1987). According to (Jolliffe and Cadima 2016), this technique is one way to perform dimensionality reduction on such datasets, by trying to reduce information loss and clarifying readability. This is achieved by creating new variables from the ones available that are uncorrelated and iteratively maximizing variance.

The following images were accomplished with Python's (Rossum 1995) Scikit-learn (Buitinck, et al. 2013) library. Figures 4 to 6 show the first three PCA components of the user feedback, static analysis, and dynamic analysis, respectively, which are show in three dimensions for improved readability. These figures show small, isolated clusters of both malware and goodware, slightly separated from the main cluster. The transparency of each circle indicates the frequency of apps that fall into that group. This further outlines the ability of malicious applications to mimic goodware.

In the user feedback cases, the first 3 components managed to explain approximately 76% of the total variance. To be able to explain the total variance in this sample, the PCA would need 8 components, which is just one less feature than the the available ones. For the other two datasets, their respective PCA showed that the first 3 components managed to explain approximately 100% of the total variance.

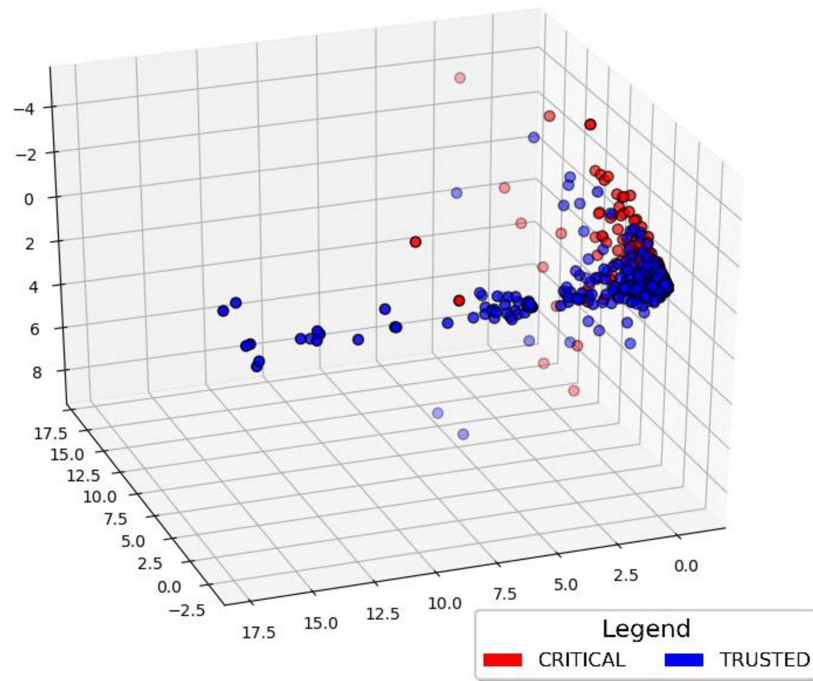

Figure 4 – User Feedback datasest first 3 PCA components

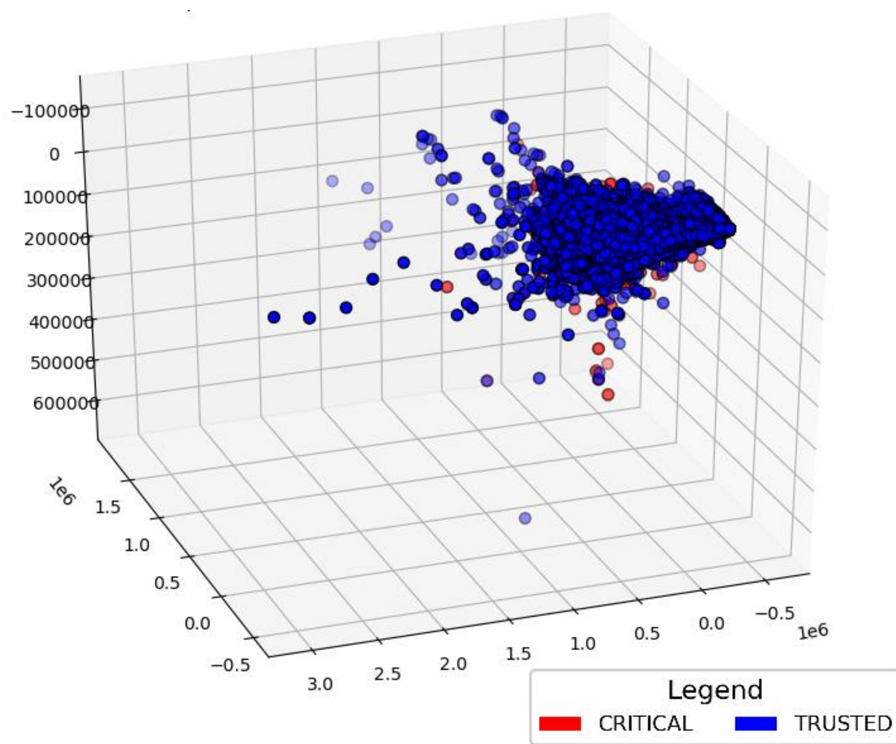

Figure 5 – Static Analysis datasest first 3 PCA components

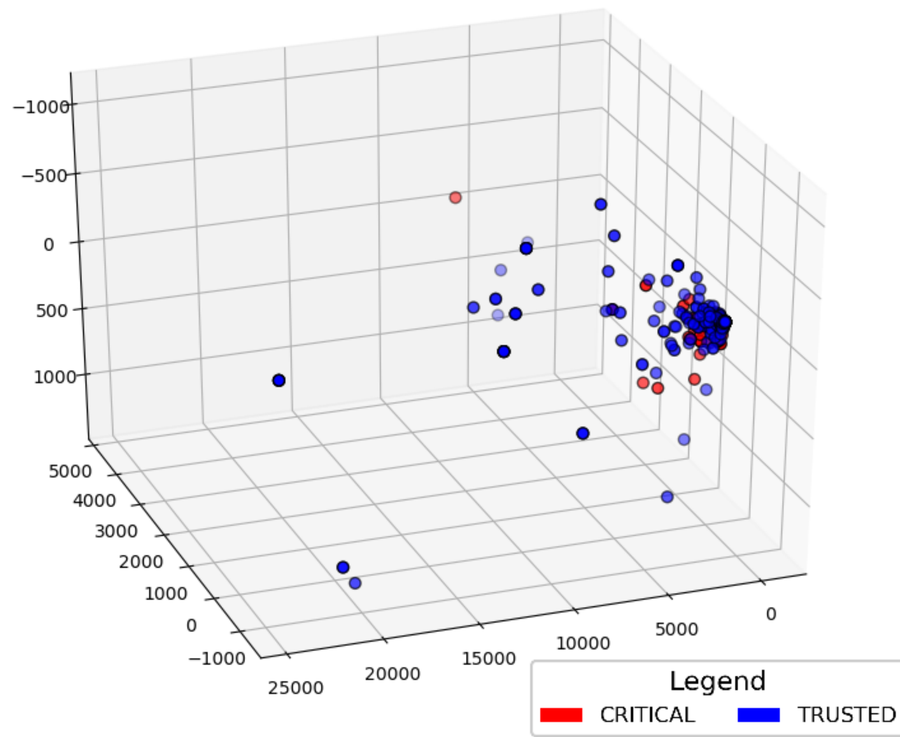

Figure 6 – Dynamic Analysis datasest first 3 PCA components

3.2.2. T-SNE Analysis

The large complexity of this issue, due to the elevated feature space, required a different approach more suited for this task of dimensionality reduction. T-Distributed Stochastic Neighbor Embedding (t-SNE) (Maaten and Hinton 2008) is a more appropriate approach for large and complex datasets due to its proficiency in embedding high-dimensional data for visualization in low-dimensional spaces. This algorithm accomplishes this by forming a probability distribution over object pairs so that similar objects are given a higher probability while contrasting object pairs are given a lower one. Then it defines these probabilities in a low-dimensional plot by minimizing the divergence between the two distributions relative to their positions on the plot.

This approach is known for forming visual clusters from the original data wich are strongly dependant on the algorithm parameterization. These can appear from non-clustered data as well and present misleading results. To avoid this pitfall, over 200 parameter combinations were tested for each dataset, most notably between perplexity and learning rate, avoiding the false clusters by finding the outputs that confirm convergence patterns.

Figures 7 to 9 present the exemplary results of the algorithm applied to the three separate datasets. Here it is shown both two- and three-dimensional perspectives for the three respective datasets. Contrary to the PCA results, these outputs indicate a clear presence of easily identifiable malware clusters, most notably in the representation from the dynamic analysis dataset. However, many malicious samples continue to present themselves alongside goodware, especially in the static analysis dataset. This clear definition of some clusters of malware make it expectable that classification models are able to differentiate clearly at least part of the malware.

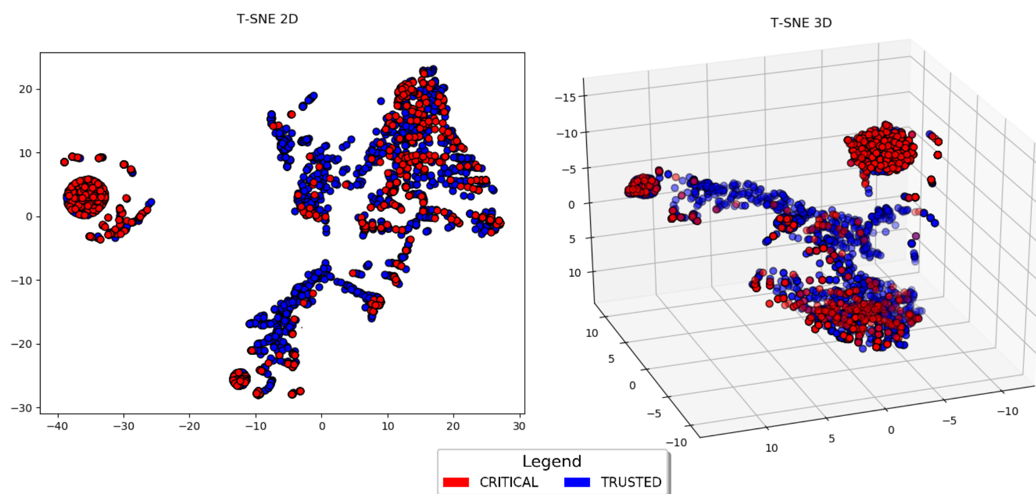

Figure 7 – User Feedback t-SNE 2D and 3D comparison

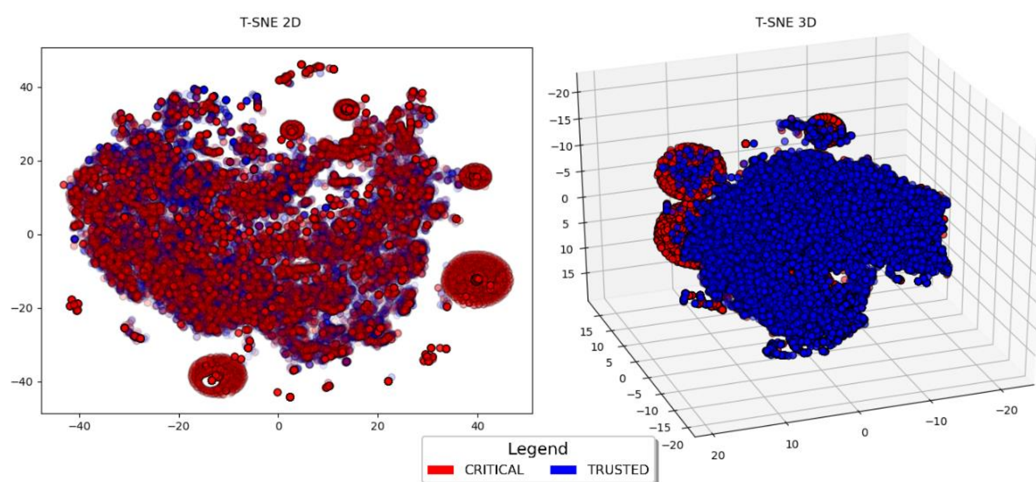

Figure 8 – Static Analysis t-SNE 2D and 3D comparison

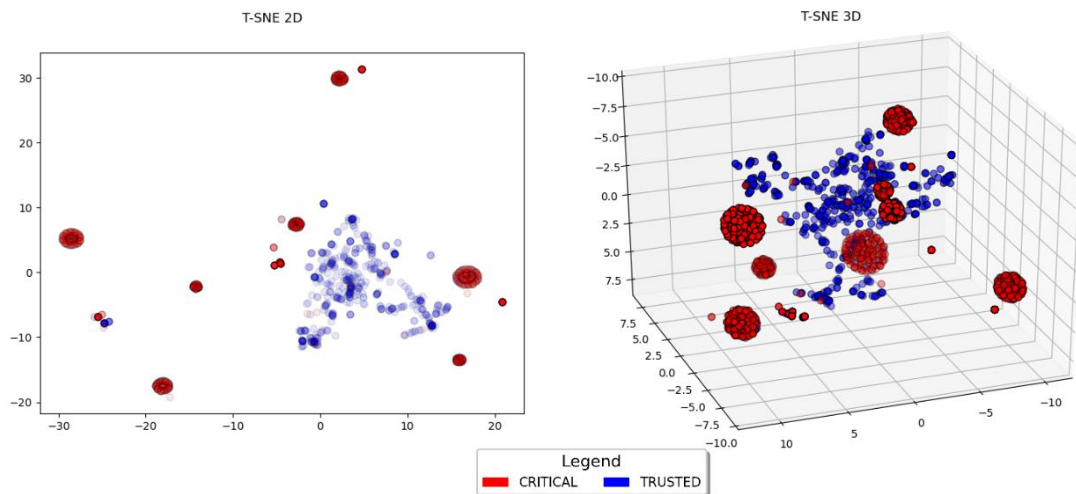

Figure 9 – Dynamic Analysis t-SNE 2D and 3D comparison

3.3. Data Preparation

To facilitate model application and to improve its performance, the quality of the data that is fed into them is of crucial importance. This section outlines the methods used to prepare and process data before applying the machine learning algorithms. To perform these types of data transformation several tools were used, namely Microsoft Excel and Python programming language (Rossum 1995) in conjunction with the Pandas (McKinney 2010), Numpy (Walt, Colbert and Varoquaux 2011), and Scikit-learn (Buitinck, et al. 2013) libraries.

Since the datasets used are reports produced from analysis tools, their outputs are already structured and their features well defined. Other than the Koodous reports, for the dynamic analysis dataset, which had to be converted from detailed JSON files to a single CSV file, the other datasets were already in Microsoft Excel sheets which made conversion to CSV a non-issue. As such, invalid data as well as data errors were not found.

To improve model preformance of the user feedback dataset some outliers were removed. Due to the elevated popularity of some apps and their features presenting values in orders of magnitude higher than the rest, the top 0.5% of the goodware applications was removed, therefore, a total of 2318 remained from the original 2332.

In order to develop balanced models and reduce their bias to the majority of the sample population, the user feedback and static analysis datsets were balanced to approximately 1:1 malware to goodware ratio. This was achieved through a random undersample of the majority class, which in both cases was the goodware class. Although the current literature does not indicate in this case which malware to goodware ratio provides better results when applied to a real-world scenario, this procedure was made to circumvent a high goodware bias from the models during training.

To better utilise the Koodous analysis reports, many features had to be transformed to be served as input for the models to learn on. Owing to the text based naure of some features, for example, the specific files read were converted into the total number of files read for each app instead of creating a single unique feature for each specific file which would create a very sparse matrix. The same was done to the rest of the text-based features, like cryptographic processes, domains accessed, HTTP connections and so on. Other features that did not present any content were removed as well as unique identifiers.

Following these primary data preparation methods several data preprocessing techniques were chosen to transform the datasets. These techniques were chosen to create several separately processed datasets before inputing them into the machine learning algorithms. This was done to avoid model biases, and in some cases improve time efficiency, by scaling the features to the same scales and therefore contributing equally to the model fitting. Alongside these, the unprocessed datasets were also used in model fitting to serve as performance control. They are as follows:

3.3.1. Standard Scaler

The Standard Scaler (STD) preprocessing method standardizes each feature by removing the mean and scaling to unit variance, essentially setting the mean value to 0 and a standard deviation of 1. With multivariate datasets this is achieved feature-wise, standardizing each feature according to their values, independent from each other.

3.3.2. Normalizer

The Normalizer (NORM) preprocessing method rescales the vector for each sample to have unit norm, independently of the distribution of the samples, effectively scaling each row of the dataset to unit norm, without removing the mean.

3.3.3. Power Transformer (Yeo-Johnson)

The Power Transformer (PowerYJ) preprocessing method, utilizing the Yeo-Johnson transform (Yeo and Johnson 2000) variant, develops a monotonic transformation of data using power functions, turning the data more Gaussian-like, while also allowing for zero and negative values, where the optimal parameter for stabilizing variance and minimizing skewness is estimated through maximum likelihood.

3.3.3. Quantile Transformer

The Quantile Transformer (Quant) preprocessing method transforms the features to follow a normal distribution using quantiles information. This transformation tends to spread out to the most frequent values while also reducing the impact of outliers, making it a robust preprocessing scheme. This transformation is applied to each feature independently, firstly by mapping the original values to an uniform distribution using an estimate of the cumulative distribution function, and secondly, by mapping the obtained values to the desired output distribution using the associated quantile function.

3.4. Detection Models

This section presents the algorithms chosen to construct the malware detection models. Algorithms from different classifier families were selected to fit the optimization pipeline, by systematically training these under differently preprocessed datasets and parameterizations and comparing their results the most effective algorithm is chosen for each of the three types of analysis. Here a simple description of each of the algorithms introduced. The following algorithms were used through the Python programming language (Rossum 1995) in conjunction with the Scikit-learn (Buitinck, et al. 2013) and XGBoost (Chen and Guestrin 2016) libraries.

3.4.1. Extreme Gradient Boosting

The Extreme Gradient Boosting (XGBoost) (Chen and Guestrin 2016) algorithm is an optimized distributed gradient boosting algorithm designed with a focus on efficiency and flexibility. An ensemble method which is a variant of the Gradient Boosting Machine (GMB), providing a parallel tree boosting method. It utilizes a depth-first approach to tree

pruning, meaning it uses the max depth parameter as specified and starts pruning trees backward. This feature alongside cache awareness and out-of-core computing, among many other optimizations, create an enhanced version of the GBM framework therefore, reducing training time and improving prediction power.

3.4.2. Random Forest

Another ensemble method, the Random Forest (RF) classifier (Breiman 2001) is a meta estimator that generates and fits a forest of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. Each of the individual trees is considered a weak learner that is constructed on a subset of the original dataset. To make the final prediction, the algorithm uses the average prediction of over all the trees. The number of trees defined, and their depth, allows for a better control over variance and bias.

3.4.3. Support Vector Machines

The Support Vector Machines (SVM) (Vapnik, Golowich and Smola 1997) are a set of supervised machine learning methods used for regression, classification, and outlier detection. They work by finding the hyperplane in the N-dimensional space, where N is the number of features, that precisely separates the datapoints. These hyperplanes are essentially decision boundaries that separate the datapoints through the usage of support vectors that try to maximise the margin between the data points and the hyperplane. Due to the high dimensionality and clustered properties of the datasets used, the Radial Basis Function (RBF) kernel was the one used in this study. The RBF kernel is a function whose value is dependant on the distance between the data points.

3.4.4. K-Nearest Neighbor

The K-Nearest Neighbor (KNN) (Altman 1992) is a non-parametric instance-based learning method used for regression and classification. The principle behind this method is to find the K predefined number of training samples closest in distance to the new point. The classification is computed from a majority vote of these nearest neighbors of each

point and the weight of these votes can be defined as dependant on the distance of the neighbors, the nearest neighbors therefore having a more impactful vote than the others.

3.3.5. Naïve Bayes Classifier

Two variants of the Naïve Bayes classifier were used, the Gaussian Naïve Bayes (GNB) and the Bernoulli Naïve Bayes (BNB) (Manning, Schütze and Raghavan 2008), these methods are a set of supervised learning algorithms based on applying Bayes' theorem with the assumption of conditional independence between every pair of features given the value of the class variable. The difference between these two variants is in the distribution of the data, the former assumes a Gaussian data distribution while in the latter the data assumes multivariate Bernoulli destributions.

3.5. Model Application

To develop the optimized model for each of the datasets, a simple approach was taken that enabled to get the most out of each algorithm. The previously introduced algorithms were applied on the three different datasets, user feedback, static and dynamic analysis, and each of their differently preprocessed variants, alongside the original dataset without any of the preprocessed methods applied to serve as a performance comparison baseline. Each of the datasets were divided into 80% train and validation data, with a corresponding 20% data holdout to use as test data to measure the performance of the final model fit on the training dataset. The training data was split using a 10-fold stratified cross validation which works by randomly partitioning the sample data into k-sized subsamples. This variation perserves the percentages of samples from each class and as such is considered a better scheme when compared to regular cross validation (Kohavi 1995).

To automate the hyperparameter tunning of all the learning algorithms, a random grid search strategy was implemented, with 100 parameter combinations for each algorithm. This was to ensure that the best parameterization for each algorithm and dataset combination was achieved without exhausting all of the parameter combinations through a normal grid search or through a manual search, which would be impractical due to the time constraints and computational power available. This chosen method has been shown to be more efficient than the other two in (Bergstra and Bengio 2012).

3.6. Model Evaluation

To compare the performance between models, several metrics were used. These following evaluation metrics were chosen to classify and compare models in different categories. To better understand some of the following metrics a brief introduction of the confusion matrix concept is needed. Table 13 depicts an example of a confusion matrix.

Table 13 – Confusion Matrix Example

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="2">Predicted Class</td></tr>
<tr><td>Positive</td><td>Negative</td></tr>
<tr><td rowspan="2">Actual Class</td><td>Positive</td><td>True Positives (TP)</td><td>False Negatives (FN)</td></tr>
<tr><td>Negative</td><td>False Positives (FP)</td><td>True Negatives (TN)</td></tr>
</table>

This presents the relationship between each class and what the model predicted. True Positives (TP) and True Negatives (TN) are the malware and goodware samples respectively that the model classified correctly. While False Positives (FP) and False Negatives (FN) are the incorrectly classified goodware and malware respectively, goodware being classified as malware and vice-versa. From these basic measures the overall evaluation of the system is given by:

- Precision: $P = \frac{TP}{TP+FP}$ (positive predicted value)
- Recall: $R = \frac{TP}{TP+FN}$ (sensitivity, hit rate or true positive rate)
- Accuracy: $A = \frac{TP+TN}{TP+TN+FP+FN}$
- F1 score: $F_1 = 2 * \frac{P*R}{P+R}$

- False-positive rate: $FPR = \frac{FP}{FP+TN}$ (Apps that are goodware, but are predicted as malware)

- False-negative rate: $FNR = \frac{FN}{FN+TP}$ (Apps that are malware, but are predicted as goodware)

Besides these metrics, the Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) curve and the AUC of the Precision-Recal (P-R) were also used as metrics to compare model performance. These two metrics calculate the ability of a binary classifier system as its discrimination threshold between classes is shifted. In the case of ROC curve, the TPR is ploted against the FPR at different thresholds while the P-R curve shows the tradeoff between precision and recall for different thresholds. The AUC of these respective curves represents the measure of separability, how well the model is capable of distinguishing between classes, in this case, between malware and goodware.

Finally, the training time of the models is also taken into consideration. This is calculated as the sum of the time taken to train and test each of the hyperparameter configurations before outputting the one that has the best overall performance metric values.

Chapter 4 – Malware Detection Model Testing

This chapter focuses on the testing of the different detection models previously developed and trained on the different dataset categories. Each detection model is tested and compared on the various evaluation metrics formerly introduced for each of the differently preprocessed datasets. The algorithm that displayed the best overall performance is then selected for the performance decay study in the chapter that follows. The models developed in this section are built using the full balanced datasets of each category with a malware to goodware ratio of 1:1.

4.1 User Feedback Complete Dataset

Table 14 presents the results for all the classification models developed for each of the differently preprocessed versions of the original user feedback dataset, alongside the unpreprocessed dataset as a reference point. Most models show similar results performance-wise with some variations, and few exceptions. The ensemble algorithms, XGBoost and Random Forest, provided the best results overall with all of the dataset variants, with XGBoost prevailing on top by reaching an F_1 Score of 0.79 with the PowerYJ transform and AUC/ROC and AUC/P-R scores of 0.873 and 0.841 respectively with the STD transformed dataset.

A notable mention to the PowerYJ transform which allowed the other algorithms to reach their best results overall, with the exception of NBBernoull, most notably the distance based algorithms, SVM and KNN, which in this case rival the scores of the Random Forest algorithm, with 0.75 F_1 Score in both models and 0.818 and 0.830 AUC/ROC from the SVM and KNN models respectively. The Naïve Bayes based algorithms showed weakest performance overall due to their need of specific data distributions. Nonetheless, the NBGaus achieved similar results compared to the other models with the Quant transformation and the NBBernoulli without any kind of preprocessing method and even achieving the lowest FNR of 19.5%. Although in raw numbers there are lower ones these are irrelevant due to the high FPR bias, which in some cases reaches 100%, rendering these models negligible.

Table 14 – Model performance metrics for the full User Feedback dataset

Algorithm	Preprocessing Method	F_1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R	Total Time
XGBoost	NoPrep	0.770	0.775	0.136	0.314	0.867	0.835	1h 40m 15.83s
XGBoost	STD	0.770	0.775	0.144	0.305	0.873	0.841	1h 27m 51.84s
XGBoost	NORM	0.750	0.754	0.203	0.288	0.842	0.775	0h 56m 20.58s
XGBoost	PowerYJ	0.790	0.788	0.144	0.280	0.863	0.808	0h 36m 36.93s
XGBoost	Quant	0.770	0.771	0.136	0.322	0.869	0.839	0h 23m 17.26s
RF	NoPrep	0.720	0.725	0.237	0.314	0.808	0.789	1h 8m 46.66s
RF	STD	0.710	0.712	0.212	0.364	0.804	0.789	3h 49m 0.99s
RF	NORM	0.730	0.733	0.314	0.220	0.810	0.771	3h 26m 28.16s
RF	PowerYJ	0.710	0.712	0.229	0.347	0.806	0.790	3h 11m 43.36s
RF	Quant	0.720	0.720	0.237	0.322	0.806	0.789	3h 5m 55.01s
SVM	NoPrep	0.720	0.716	0.314	0.254	0.787	0.743	1h 13m 59.12s
SVM	STD	0.600	0.614	0.542	0.229	0.721	0.716	0h 33m 22.75s
SVM	NORM	0.700	0.699	0.246	0.356	0.790	0.771	0h 25m 55.93s
SVM	PowerYJ	0.750	0.750	0.186	0.314	0.818	0.794	0h 29m 25.29s
SVM	Quant	0.710	0.712	0.186	0.390	0.769	0.714	0h 32m 6.58s
KNN	NoPrep	0.710	0.716	0.212	0.356	0.774	0.767	0h 1m 0.63s
KNN	STD	0.700	0.699	0.254	0.347	0.771	0.744	0h 1m 9.68s
KNN	NORM	0.700	0.703	0.246	0.347	0.796	0.777	0h 1m 11.03s
KNN	PowerYJ	0.750	0.754	0.212	0.280	0.830	0.807	0h 1m 0.59s
KNN	Quant	0.720	0.725	0.237	0.314	0.796	0.647	0h 1m 3.39s
NBGaus	NoPrep	0.350	0.508	0.983	0.000	0.772	0.761	0h 0m 4.28s
NBGaus	STD	0.380	0.521	0.958	0.000	0.548	0.551	0h 0m 4.24s
NBGaus	NORM	0.680	0.686	0.246	0.381	0.761	0.732	0h 0m 4.37s
NBGaus	PowerYJ	0.690	0.699	0.424	0.178	0.780	0.774	0h 0m 4.2s
NBGaus	Quant	0.710	0.708	0.288	0.297	0.773	0.769	0h 0m 4.46s
NBBernoulli	NoPrep	0.720	0.720	0.364	0.195	0.754	0.761	0h 0m 9.35s
NBBernoulli	STD	0.380	0.508	0.941	0.042	0.510	0.740	0h 0m 9.48s
NBBernoulli	NORM	0.330	0.500	0.000	1.000	0.500	0.750	0h 0m 9.16s
NBBernoulli	PowerYJ	0.630	0.653	0.585	0.110	0.684	0.649	0h 0m 9.46s
NBBernoulli	Quant	0.540	0.597	0.746	0.059	0.622	0.631	0h 0m 8.81s

Table 15 highlights the difference of the top preprocessed datasets for each of the models used, presenting in addition the mean and standard deviation (SD) results of the cross validation. The low standard deviation values for every model indicate little variance between each fold from the cross-validation procedure, suggesting that the algorithms could maintain similar levels of performance on analogous datasets.

Table 15 – Comparison of top model performance for User Feedback models
SD – Standard Deviation

Model	Preprocess Method	Measure	F1	Accuracy	FPR	FNR	AUC ROC	AUC P-R
XGBoost	PowerYJ	Mean	0.790	0.788	0.144	0.280	0.863	0.808
		SD	0.054	0.029	0.022	0.032	0.036	0.027
RF	Norm	Mean	0.740	0.733	0.314	0.220	0.810	0.771
		SD	0.040	0.038	0.048	0.033	0.048	0.042
SVM	PowerYJ	Mean	0.750	0.750	0.186	0.314	0.818	0.794
		SD	0.049	0.023	0.022	0.051	0.052	0.048
KNN	PowerYJ	Mean	0.750	0.754	0.212	0.280	0.830	0.807
		SD	0.033	0.031	0.046	0.037	0.055	0.049
NBGaus	Quant	Mean	0.710	0.708	0.288	0.297	0.773	0.769
		SD	0.040	0.038	0.044	0.032	0.046	0.055
NBBernoulli	NoPrep	Mean	0.720	0.720	0.364	0.195	0.754	0.716
		SD	0.039	0.048	0.049	0.038	0.047	0.050

A particular feature of the XGBoost (Chen and Guestrin 2016) framework is that it allows to calculate the importance of each feature used in the model training and by how much it influences it. Figure 10 shows the importance of each feature sorted by their gain in the developed highest performing XGBoost model, in this case the one applied on the PowerYJ transformed dataset. In this case the gain metric represents the average gain across all splits the feature is used in.

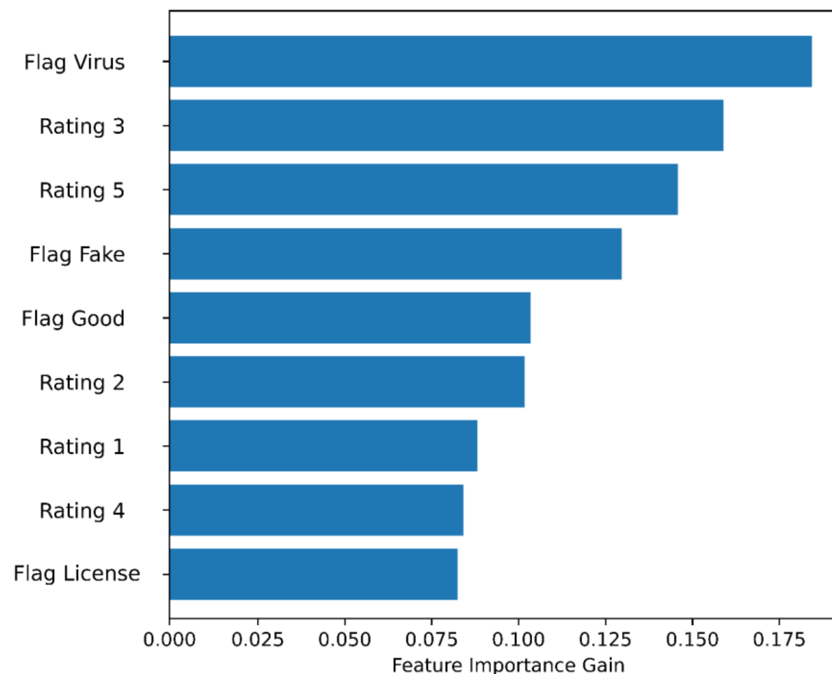

Figure 10 – XGBoost User Feedback feature importance

Here we can see that the Flag Virus is considered the most important feature, this means that users can be expected to give this flag to more suspicious apps. Following this, the 3- and 5-star ratings were ranked in second and third place respectively, possibly due to the fact that popular apps present in the marketplace are less likely to show malicious behaviours, given that these were the most used ratings on average. In last place of importance came the Flag License, most likely due to its use being more prevalent when apps are still gaining popularity and are still not widely used.

4.2 Static Analysis Complete Dataset

Following this, Table 16 shows the model test results for the Static Analysis dataset and all its differently preprocessed variants, joined by the unprocessed dataset to serve as point of reference. In this case the Naïve Bayes algorithms failed to meet any requirements, showing poor performance overall. However, the other algorithms demonstrated more viable malware detection when compared to the previous use case. The XGBoost models, demonstrated superior performance yet again, this time with the Quantile transformed dataset, reaching an F_1 Score of 0.86 and both AUCs above the 90% threshold, with 0.913 and 0.926 for the AUC/ROC and AUC/P-R respectively, demonstrating the capability of classifying malware with this type of analysis only, also reaching a low FPR of 7.8%. The distance-based algorithms, SVM and KNN, also showed improved overall performance in this use case, reaching an F_1 Score of 0.84 and 0.83, respectively. The major downside with the SVM models is their very high training time when compared to the rest, even reaching an astounding 40 hours of train time for all of the parameter combinations in the unprocessed dataset. Lastly, the RF models demonstrated little improvement comparatively to the previous use case due to the very high FNR bias reaching 43.2% in the worst case.

Table 16 – Model performance metrics for the full Static Analysis dataset

Algorithm	Preprocessing Method	F_1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R	Total Time
XGBoost	NoPrep	0.840	0.838	0.082	0.242	0.902	0.918	2h 15m 48.77s
XGBoost	STD	0.850	0.849	0.106	0.196	0.903	0.915	6h 6m 40.3s
XGBoost	NORM	0.850	0.849	0.059	0.243	0.905	0.917	4h 34m 3.19s
XGBoost	PowerYJ	0.840	0.838	0.129	0.194	0.898	0.915	2h 47m 36.95s
XGBoost	Quant	0.860	0.856	0.078	0.210	0.913	0.926	3h 32m 54.33s
RF	NoPrep	0.740	0.745	0.089	0.420	0.788	0.833	4h 27m 57.17s
RF	STD	0.740	0.749	0.070	0.432	0.790	0.834	6h 39m 22.4s
RF	NORM	0.750	0.755	0.061	0.428	0.795	0.842	9h 54m 0.56s
RF	PowerYJ	0.730	0.742	0.096	0.420	0.789	0.834	6h 7m 47.52s
RF	Quant	0.740	0.746	0.086	0.422	0.788	0.833	6h 12m 18.68s
SVM	NoPrep	0.790	0.798	0.026	0.378	0.827	0.881	40h 29m 53.36s
SVM	STD	0.810	0.809	0.073	0.309	0.855	0.865	9h 18m 32.12s
SVM	NORM	0.640	0.646	0.482	0.225	0.679	0.620	11h 23m 29.28s
SVM	PowerYJ	0.830	0.833	0.103	0.230	0.881	0.872	12h 31m 44.12s
SVM	Quant	0.840	0.837	0.099	0.227	0.883	0.874	11h 39m 26.04s
KNN	NoPrep	0.810	0.814	0.114	0.259	0.862	0.879	0h 40m 39.1s
KNN	STD	0.830	0.830	0.145	0.196	0.891	0.911	2h 57m 52.16s
KNN	NORM	0.800	0.805	0.118	0.272	0.857	0.886	1h 6m 19.25s
KNN	PowerYJ	0.830	0.832	0.129	0.208	0.893	0.913	2h 41m 39.87s
KNN	Quant	0.830	0.833	0.134	0.200	0.892	0.909	0h 18m 49.74s
NBGaus	NoPrep	0.510	0.562	0.761	0.115	0.655	0.579	0h 1m 2.45s
NBGaus	STD	0.510	0.562	0.761	0.115	0.655	0.579	0h 1m 2.01s
NBGaus	NORM	0.480	0.548	0.818	0.087	0.705	0.686	0h 1m 1.88s
NBGaus	PowerYJ	0.630	0.635	0.476	0.255	0.666	0.619	0h 1m 1.94s
NBGaus	Quant	0.650	0.652	0.422	0.275	0.669	0.629	0h 1m 1.92s
NBBernoulli	NoPrep	0.450	0.534	0.072	0.859	0.580	0.587	0h 1m 4.48s
NBBernoulli	STD	0.560	0.583	0.663	0.171	0.611	0.739	0h 1m 4.21s
NBBernoulli	NORM	0.330	0.500	0.000	1.000	0.500	0.750	0h 1m 3.52s
NBBernoulli	PowerYJ	0.570	0.592	0.618	0.197	0.631	0.736	0h 1m 4.33s
NBBernoulli	Quant	0.580	0.596	0.618	0.191	0.626	0.723	0h 1m 4.28s

Table 17 displays the performance metrics of the top performing model for each algorithm for comparison. It is worth noting that in this case the Quant transform was favored by most algorithms apart from RF and KNN. Here the variation between folds for each metric averages around a decimal point, demonstrating stable performance between cross-validation folds.

Table 17 - Comparison of top model performance for Static Analysis models
SD – Standard Deviation

Model	Preprocess Method	Measure	F_1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R
XGBoost	Quant	Mean	0.860	0.856	0.078	0.210	0.913	0.926
		SD	0.013	0.011	0.006	0.010	0.007	0.008
RF	Norm	Mean	0.750	0.755	0.061	0.428	0.795	0.842
		SD	0.010	0.012	0.011	0.029	0.006	0.007
SVM	Quant	Mean	0.840	0.837	0.099	0.227	0.883	0.874
		SD	0.011	0.014	0.016	0.020	0.005	0.006
KNN	PowerYJ	Mean	0.830	0.832	0.129	0.208	0.893	0.913
		SD	0.014	0.010	0.018	0.027	0.009	0.011
NBGaus	Quant	Mean	0.650	0.652	0.422	0.275	0.669	0.629
		SD	0.037	0.038	0.044	0.019	0.007	0.009
NBBernoulli	Quant	Mean	0.580	0.596	0.618	0.191	0.626	0.723
		SD	0.045	0.048	0.051	0.022	0.010	0.007

To be able to understand the impact of each feature of the dataset used in this use case, the same analysis was performed as in the previous section, using the top performing model to extract its feature importance values, in this case the XGBoost model trained on the Quant transformed Static Analysis dataset. In Figure 11 the features from the Static Analysis dataset are ordered in terms of importance by the same gain metric. In this case all the features display very similar values. The notable instances are the Resource Strings, where the number of additional resource files an app has can indicate its intent, and the Dalvik specific opcodes, which appear to be less relevant than the other features.

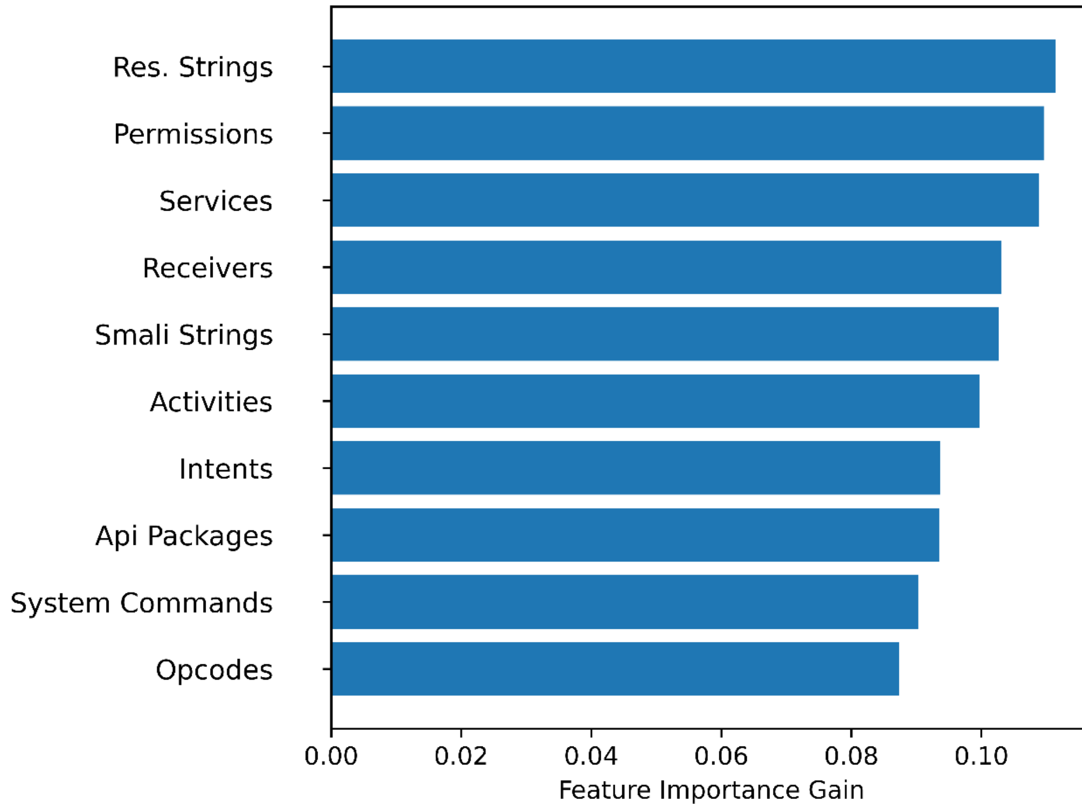

Figure 11 - XGBoost Static Analysis feature importance

4.3 Dynamic Analysis Complete Dataset

Finally, Table 18 introduces the model performance metrics when applied to the dynamic analysis dataset, and all its preprocessed variants. This last use case demonstrated the relevance of dynamic analysis features. Although the Naïve Bayes algorithms displayed the lowest performance values overall, they still exhibited similar classifying capabilities when compared to the other algorithms. The remaining algorithms were all capable of surpassing the 90% thresholds for Accuracy and F_1 Scores. The XGBoost models remained unchallenged in their overall performance, although by a much smaller margin, being surpassed by the RF and KNN models in the time taken to train.

Table 18 – Model performance metrics for the full Dynamic Analysis dataset

Algorithm	Preprocessing Method	F_1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R	Total Time
XGBoost	NoPrep	0.940	0.938	0.106	0.018	0.973	0.958	0h 21m 55.92s
XGBoost	STD	0.940	0.939	0.106	0.016	0.972	0.959	0h 20m 10.24s
XGBoost	NORM	0.930	0.934	0.100	0.032	0.968	0.955	0h 24m 15.52s
XGBoost	PowerYJ	0.940	0.939	0.102	0.020	0.967	0.952	0h 24m 38.35s
XGBoost	Quant	0.940	0.940	0.104	0.016	0.973	0.961	0h 33m 48.64s
RF	NoPrep	0.930	0.928	0.104	0.040	0.965	0.955	0h 11m 9.21s
RF	STD	0.920	0.917	0.117	0.050	0.970	0.961	0h 10m 26.33s
RF	NORM	0.930	0.928	0.106	0.037	0.967	0.958	0h 23m 47.17s
RF	PowerYJ	0.920	0.923	0.102	0.052	0.970	0.962	0h 9m 38.94s
RF	Quant	0.920	0.919	0.100	0.062	0.970	0.960	0h 9m 41.97s
SVM	NoPrep	0.920	0.922	0.098	0.058	0.948	0.924	2h 35m 41.42s
SVM	STD	0.940	0.935	0.100	0.029	0.962	0.951	1h 29m 21.98s
SVM	NORM	0.920	0.923	0.129	0.025	0.948	0.924	0h 52m 5.31s
SVM	PowerYJ	0.930	0.928	0.102	0.042	0.953	0.935	0h 58m 24.56s
SVM	Quant	0.920	0.916	0.098	0.071	0.961	0.939	0h 17m 18.83s
KNN	NoPrep	0.930	0.929	0.115	0.027	0.961	0.952	0h 2m 38.01s
KNN	STD	0.930	0.929	0.113	0.029	0.959	0.944	0h 4m 3.69s
KNN	NORM	0.930	0.926	0.104	0.044	0.950	0.926	0h 1m 54.51s
KNN	PowerYJ	0.940	0.935	0.104	0.025	0.968	0.956	0h 4m 5.44s
KNN	Quant	0.930	0.931	0.108	0.029	0.960	0.943	0h 3m 9.72s
NBGaus	NoPrep	0.810	0.820	0.200	0.060	0.900	0.910	0h 0m 2.69s
NBGaus	STD	0.810	0.809	0.315	0.067	0.930	0.921	0h 0m 2.69s
NBGaus	NORM	0.850	0.849	0.210	0.092	0.907	0.874	0h 0m 1.61s
NBGaus	PowerYJ	0.900	0.897	0.123	0.083	0.915	0.901	0h 0m 1.64s
NBGaus	Quant	0.750	0.757	0.444	0.042	0.923	0.901	0h 0m 1.47s
NBBernoulli	NoPrep	0.880	0.880	0.183	0.056	0.914	0.896	0h 0m 3.69s
NBBernoulli	STD	0.780	0.780	0.367	0.073	0.870	0.853	0h 0m 2.38s
NBBernoulli	NORM	0.330	0.500	0.000	1.000	0.500	0.750	0h 0m 2.12s
NBBernoulli	PowerYJ	0.870	0.868	0.204	0.060	0.906	0.874	0h 0m 3.41s
NBBernoulli	Quant	0.830	0.830	0.290	0.050	0.888	0.857	0h 0m 2.26s

Table 19 shows the top performing models for each of the algorithms side by side for an easier comparison, as well as the mean and standard deviation values from the cross-validation method.

Table 19 - Comparison of top model performance for Dynamic Analysis models
SD – Standard Deviation

Model	Preprocess Method	Measure	F_1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R
XGBoost	Quant	Mean	0.940	0.940	0.104	0.016	0.973	0.961
		SD	0.011	0.010	0.016	0.007	0.006	0.005
RF	Norm	Mean	0.930	0.928	0.106	0.037	0.967	0.958
		SD	0.012	0.010	0.020	0.011	0.090	0.008
SVM	STD	Mean	0.940	0.935	0.100	0.029	0.962	0.951
		SD	0.023	0.034	0.025	0.013	0.040	0.019
KNN	PowerYJ	Mean	0.830	0.832	0.129	0.208	0.893	0.913
		SD	0.009	0.011	0.032	0.023	0.010	0.009
NBGaus	PowerYJ	Mean	0.900	0.897	0.123	0.083	0.915	0.901
		SD	0.025	0.031	0.055	0.025	0.040	0.042
NBBernoulli	NoPrep	Mean	0.880	0.880	0.183	0.056	0.914	0.896
		SD	0.036	0.040	0.049	0.026	0.047	0.039

In the same fashion as the previous cases the feature importance analysis was made. Figure 12 displays the importance of each feature in this dataset ordered by their average gain across all splits that were used in the training of the highest performing model, in this case the XGBoost model trained on the Quant transformed dataset. This XGBoost model considered the number of content providers that manage access to the central data repository the most relevant feature followed by the number of receivers, filters and DNS requests, indicating that these are the most relevant to identify the presence of malicious behaviour.

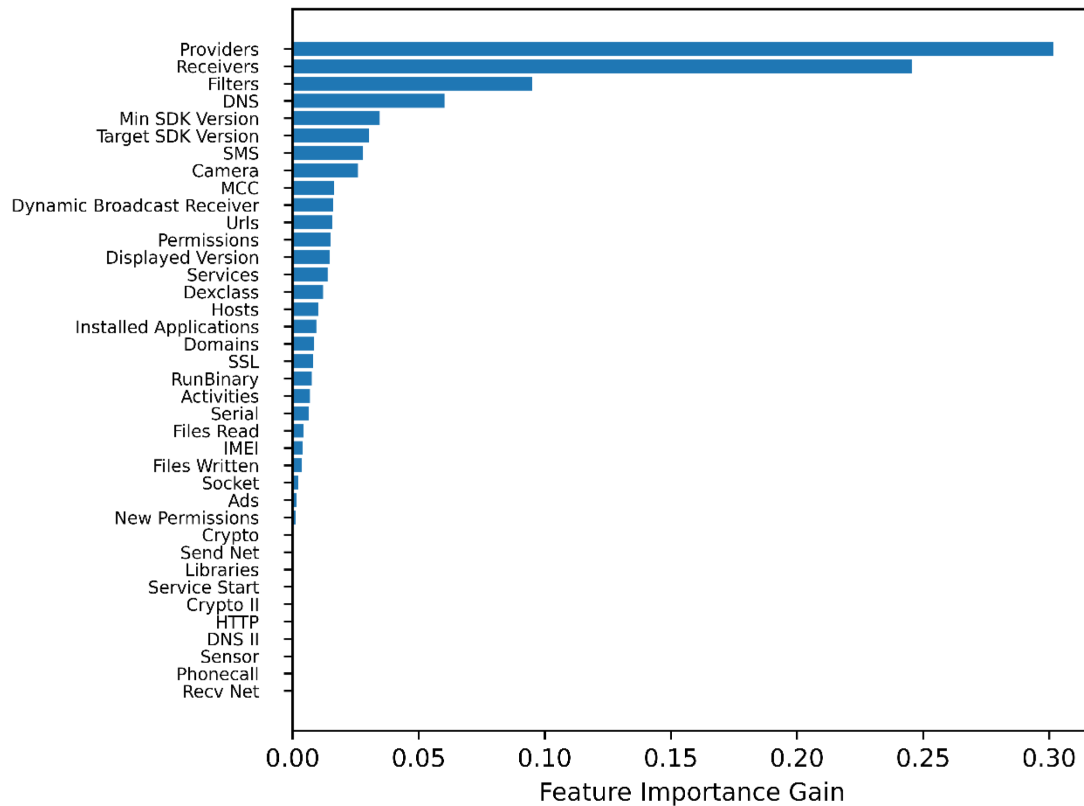

Figure 12 - XGBoost Dynamic Analysis feature importance

Although the XGBoost model did consider some features like the number of phonecalls and libraries used to have zero information gain, it does not mean that other models did not extract information out of these. Therefore, it was decided to maintain the features for training further models.

4.4 Comparison of Results

By analysing the results obtained from the models built on each of the dataset categories, the XGBoost algorithm demonstrated the best overall capabilities in classifying malware. Its results were mostly consistent in each of the preprocessed variants in each category. It is worth noting its proficiency as a classifier even when trained on the original dataset without any of the preprocessing methods applied beforehand.

The user feedback models showed the lowest results. This can be due to the fact that users tend to generate feedback according to how they feel about the applications, relying

more on their emotions rather than their objective experience with each app. This is shown by the previously demonstrated high correlation values between the features in this dataset, most notably the rating values, which indicate the low variance of each rating value across the dataset, meaning that even highly popular apps retained overall the same percentage of low ratings than unpopular ones.

In the static analysis use case, the models displayed mostly the same relations with each other performance-wise, but with slight improvements overall. The XGBoost algorithm continued to be superior, being the only to produce models with AUCs above the 90% threshold, further displaying its ability as a classifier. The FNR remained above 20% in most cases, which means that observing each apps' code alone is not enough to enable satisfactory discrimination between malware and goodware.

In the dynamic analysis case, every classifier was able to properly discriminate malware from goodware, apart from the Naïve Bayes algorithms by comparison. The FPRs remained largely around the 10% mark while FNRs remained below the 5% mark. This is a good indicator of the ability for these classifiers to distinguish malware from goodware with a complex set of features obtained from analysing application behaviour.

Lastly, the training times cannot be compared between use cases due to the large difference in application samples in each of the categories.

4.5 Ratio Analysis

To demonstrate the impact of class imbalance on the performance of the classifier models, a small experiment was devised. In this instance, the top performing algorithm (XGBoost) underwent the same pipeline, but with the small change of using an approximation of a "realistic" malware to goodware ratio in the training and testing datasets. In (Google 2018) they indicate that less than 1% of all applications published in their app store are PHAs, however no indication is made on the amount of malicious applications that are submitted to the marketplace. Due too this undisclosed amount, it was assumed that 10% of the apps in the train and testing dataset were malware, as "realistic" measure. Table 20 displays the performance results of this experiment.

Table 20 – XGBoost performance on the different Datasets with "realistic" ratio

User Feedback Dataset							
Algorithm	Preprocess Method	F_1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R
XGBoost	NoPrep	0.870	0.895	0.020	0.943	0.835	0.359
XGBoost	STD	0.870	0.908	0.006	0.943	0.849	0.363
XGBoost	NORM	0.900	0.919	0.009	0.800	0.879	0.424
XGBoost	PowerYJ	0.880	0.900	0.020	0.886	0.854	0.399
XGBoost	Quant	0.860	0.908	0.000	1.000	0.862	0.471
Static Analsysis Dataset							
Algorithm	Preprocess Method	F_1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R
XGBoost	NoPrep	0.930	0.940	0.012	0.540	0.850	0.603
XGBoost	STD	0.940	0.941	0.014	0.507	0.864	0.631
XGBoost	NORM	0.940	0.941	0.015	0.499	0.850	0.613
XGBoost	PowerYJ	0.940	0.941	0.018	0.475	0.847	0.614
XGBoost	Quant	0.930	0.940	0.013	0.528	0.845	0.605
Dynamic Analsysis Dataset							
Algorithm	Preprocess Method	F_1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R
XGBoost	NoPrep	0.930	0.936	0.029	0.408	0.968	0.659
XGBoost	STD	0.950	0.947	0.029	0.286	0.969	0.742
XGBoost	NORM	0.930	0.934	0.025	0.469	0.964	0.641
XGBoost	PowerYJ	0.940	0.938	0.029	0.388	0.951	0.570
XGBoost	Quant	0.940	0.942	0.023	0.408	0.971	0.696

At first glance the usage of a more "realistic" malware to goodware ratio shows higher values for the F_1 Score, Accuracy, FPR and AUC ROC metrics, indicating better performance when compared to the previous results. However, a crucial detail renders these models as poor classifiers. With the FNR showing unusual high values, this exposes the goodware classification bias, where the models, due to the higher number of samples from the goodware class tend to more frequently classify new testing samples as such, in one case even reaching 100% false negatives, where this particular model classified every test sample as goodware. This poor performance is further reflected in the AUC P-R metric which averages around the 50 to 60% mark.

This small experiment also reinforces the notion that machine learning classifiers need to be evaluated with several different metrics taken into account. Instead of relying only on general performance metrics like the F_1 Score and AUC ROC, other metrics should also be analysed and considered depending on the problem at hand.

Chapter 5 – Performance Decay Analysis

This chapter's focus is on the model's ability to retain their performance over the course of the following months after they are developed. Due to the performance presented from the XGBoost algorithm in the previous chapter, it was decided to use this algorithm exclusively for the purposes of this part of the study. The same framework was used with the only difference in selecting the specific time frames for training and testing separately. The XGBoost models were trained and tested on the differently transformed data and developed using the same random grid search hyperparameter system with 100 tested combinations.

To effectively study the model performance decay for each of the data categories, these were broken down into monthly datasets. These monthly datasets were then used to train the models on different time frames and tested on the months that followed, for example, training the models with data from October to November and then testing them separately on December, January, and so on. The number of sequential monthly combinations were made according to the data available as well. For example, if the total dataset was comprised of 6 months, the incremental monthly training combinations were tested separately in the remaining months, as exemplified in Table 21.

Table 21 – Example of model training and testing framework

October	November	December	January	February	March
Training	Testing	Testing	Testing	Testing	Testing
Training	Training	Testing	Testing	Testing	Testing
Training	Training	Training	Testing	Testing	Testing
Training	Training	Training	Training	Testing	Testing
Training	Training	Training	Training	Training	Testing

[blue] - Training [green] - Testing

5.1. User Feedback

Like in the previous chapters the first analysis category to be submitted to this analysis was the user feedback. On account of having the shortest time frame of data available, from October 2019 to January 2020, long-term performance decay analysis becomes

somewhat limited. Nonetheless, several XGboost algorithms were trained and validated in each of the sequential monthly combinations and tested on the remaining ones.

Table 22 displays the detailed reports of each model train-test combination. Here it becomes evident the inability of a model trained on the limited time frame of one month. Although the XGBoost model trained only on the data from the month of October has a comparative performance to its equivalent in the previous chapter (0.79 F_1 Score) when tested on the following month of November, it abruptly declines when applied to the data from December and January, reaching an ineffective F_1 Score of 0.54 alongside a FNR of 57.7%.

When trained on more data, from October to November, the model becomes more resilient to changes. Instead of declining in performance, it improved in all metrics apart from the FPR. Notwithstanding the absence of additional monthly time frames, the model trained on three months worth of data demonstrated that supplementary data improves performance.

In Figure 13 it is shown a visual representation of Table 22, indicating the possibility that the month of November provided an easier time in testing the model to discriminate malware from goodware, and when used for training, auxiliate in its weight learning procedure, clearly demonstrating that for this case more data stabilizes model performance when tested in future data.

Table 22 – User Feedback Performance Decay Analysis

Training Months	Testing	F_1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R
October	Nov	0.790	0.793	0.293	0.121	0.909	0.896
	Dec	0.670	0.678	0.472	0.169	0.749	0.723
	Jan	0.540	0.545	0.333	0.577	0.594	0.501
October to November	Dec	0.710	0.713	0.278	0.296	0.748	0.728
	Jan	0.730	0.729	0.367	0.174	0.792	0.774
October to December	Jan	0.740	0.738	0.277	0.247	0.796	0.748

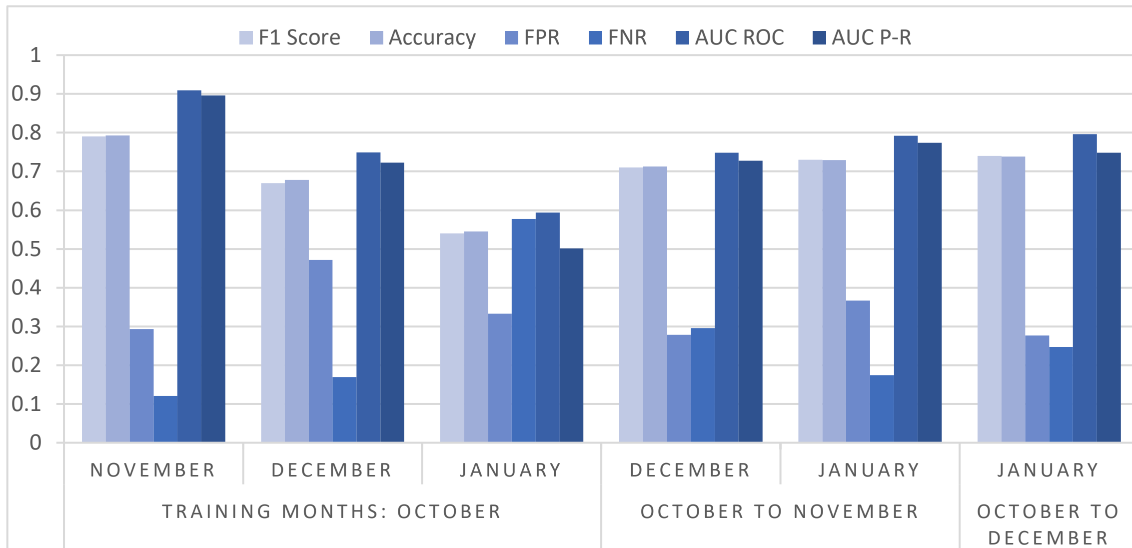

Figure 13 – Visual Representation of the User Feedback Performance Decay

5.2. Static Analysis

The static analysis data allows for a better overview of performance decay thanks to the wider time frame of six months, from October 2019 to March 2020, as well as the largest number of data samples of the three. Replicating the aforementioned procedure, the XGBoost model was trained on increasingly wider monthly time frames and tested on the remaining ones.

Table 23 presents the evaluation metrics for each of the XGBoost train-test combinations. These results again confirm the low malware detection capabilities of models trained on short time frames of data, even with each month averaging approximately 22,000 thousand samples. Although the model trained on the first month of the dataset displays high performance when tested against the following months, the overall variability of its performance joined by low performance metrics in some case reveals its inefficacy as a robust model. However, the smaller complexity of the data shows FPR and FNR in a low range of 22% and 15% respectively, when compared to the models that follow.

The need for a wider time frame is further acknowledged by the improvement in overall performance shown by the models trained on ever increasing time periods. With Accuracy values climbing from an average of 0.67, with the model trained on a period of

three months, to 0.759 when trained on six months of data. However, because of the increased data noise, the models gained a high malware classification bias demonstrated by the contrasting 5% (on average) FPR to the 50% (on average) FNR.

Finally, in similarity to the previous case, Figure 14 shows a visual representation of Table 23, assisting the hypotheses that models trained on wider time frames lead to more stability in overall model performance.

Table 23 – Static Analysis Performance Decay Analysis

Training Months	Testing	F1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R
October	Nov	0.780	0.778	0.289	0.154	0.738	0.639
	Dec	0.820	0.818	0.224	0.141	0.895	0.858
	Jan	0.680	0.683	0.269	0.364	0.668	0.594
	Feb	0.820	0.817	0.220	0.145	0.839	0.747
	Mar	0.610	0.622	0.222	0.534	0.773	0.726
October to November	Dec	0.630	0.661	0.042	0.636	0.730	0.773
	Jan	0.670	0.694	0.051	0.561	0.768	0.801
	Feb	0.620	0.653	0.038	0.657	0.729	0.769
	Mar	0.660	0.684	0.040	0.591	0.761	0.800
October to December	Jan	0.710	0.725	0.089	0.462	0.790	0.821
	Feb	0.670	0.686	0.076	0.552	0.764	0.797
	Mar	0.700	0.713	0.067	0.507	0.787	0.819
October to January	Feb	0.710	0.726	0.072	0.475	0.802	0.829
	Mar	0.700	0.715	0.080	0.490	0.794	0.830
October to February	Mar	0.750	0.759	0.087	0.395	0.830	0.862

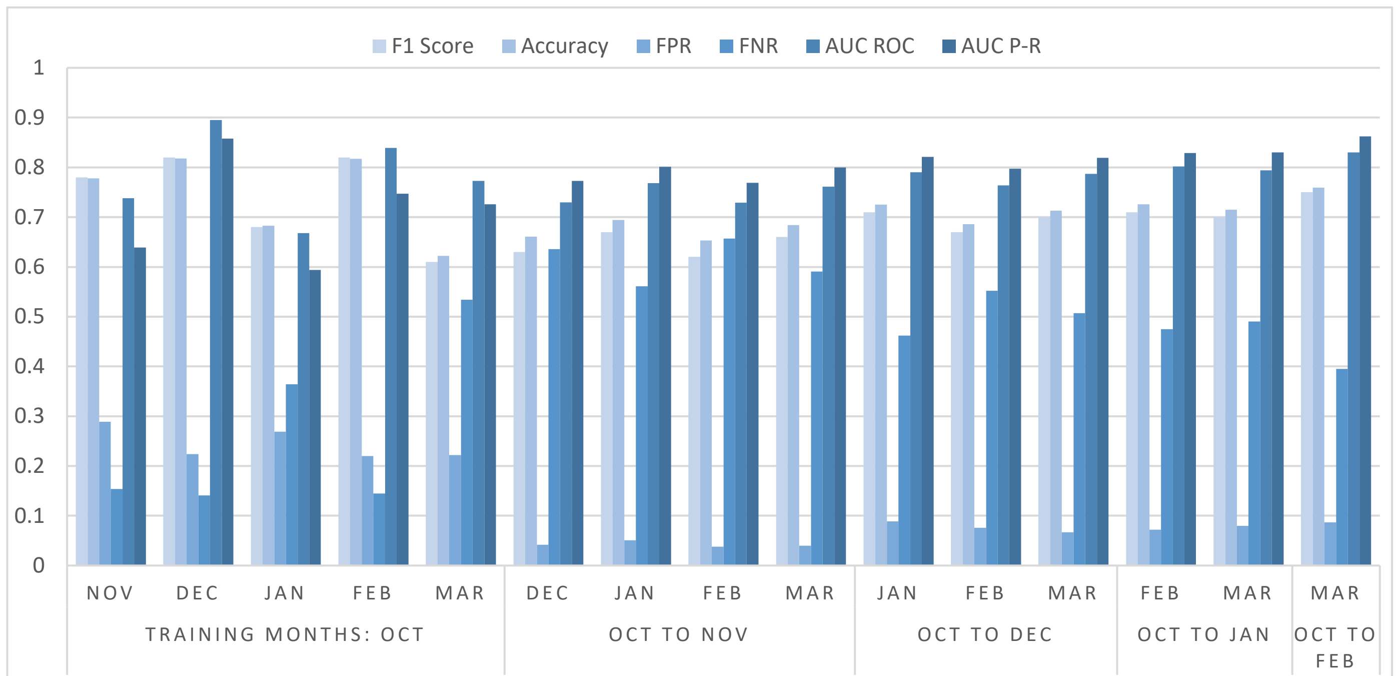

Figure 14 – Visual Representation of the Static Analysis Performance Decay

5.3. Dynamic Analysis

Finally, the static analysis dataset was the last one to undergo the performance decay analysis. Having an equivalent time window to the Static Analysis dataset of six months, from February 2020 to July 2020, allows the analysis of the performance decay of the models trained with this data to match that of the previous case. Although the average number of data samples per month is approximately 800, paling in comparison the the previous 22,000, the large feature composition allows for more complex patterns to be discovered by the algorithms that allow for an easier discrimination of malware from goodware, as seen in the previous chapter.

Table 24 exhibits the performance evaluation metrics for each of the XGBoost train-test set combinations once demonstrating the need for a wider time frame of training data, with Figure 15 providing an equivalent visual representation. Although with this type of feature rich data the model did not suffer as much of a performance drop when trained with fewer months, the high variability in the results is still present. The same increase in overall performance follows the increase in training time frame. The performance

variability also decreases up to the 3 months mark, stabilizing on the same value thresholds from there on.

Table 24 – Dynamic Analysis Performance Decay Analysis

Training Months	Testing	F1 Score	Accuracy	FPR	FNR	AUC ROC	AUC P-R
February	March	0.880	0.880	0.087	0.147	0.952	0.940
	April	0.860	0.861	0.251	0.014	0.880	0.812
	May	0.710	0.721	0.127	0.432	0.885	0.855
	June	0.930	0.928	0.027	0.117	0.980	0.980
	July	0.760	0.773	0.022	0.429	0.956	0.828
February to March	April	0.860	0.863	0.234	0.029	0.916	0.858
	May	0.850	0.855	0.149	0.142	0.891	0.853
	June	0.930	0.932	0.022	0.115	0.966	0.970
	July	0.820	0.822	0.009	0.345	0.966	0.972
February to April	May	0.930	0.927	0.027	0.119	0.980	0.981
	June	0.940	0.944	0.022	0.088	0.983	0.982
	July	0.960	0.961	0.031	0.047	0.973	0.936
February to May	June	0.960	0.957	0.031	0.054	0.984	0.984
	July	0.910	0.909	0.013	0.168	0.975	0.944
February to June	July	0.960	0.960	0.036	0.044	0.975	0.968

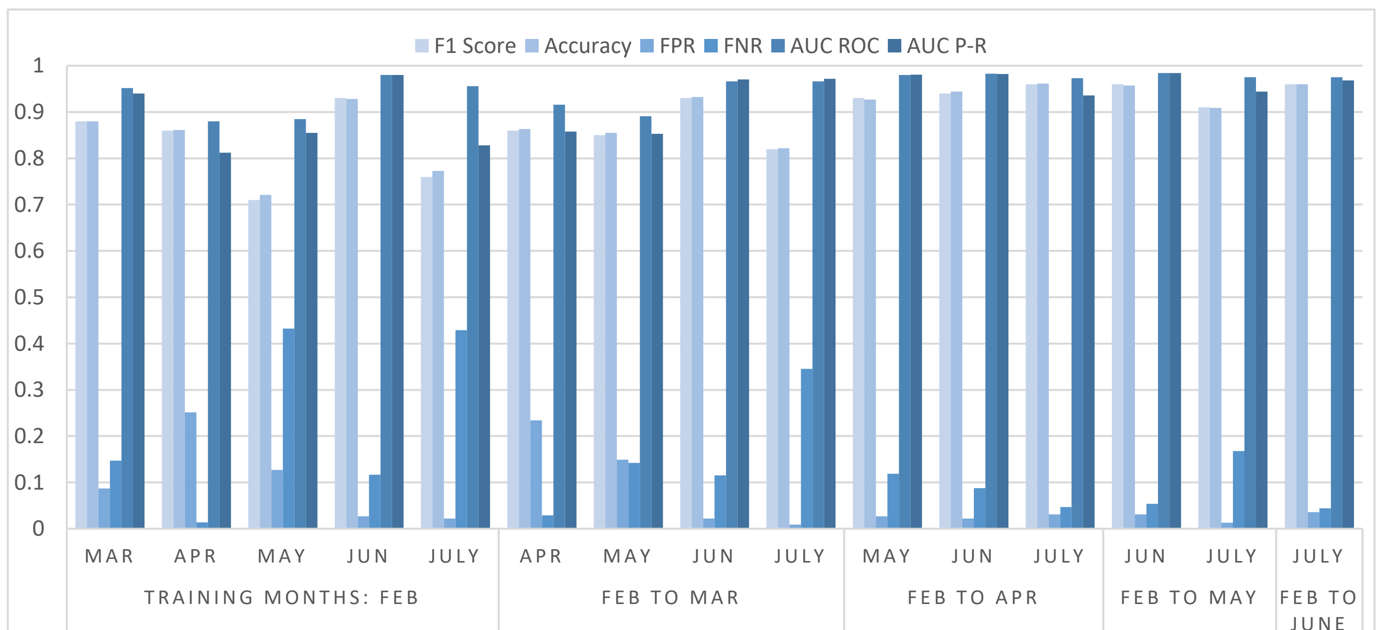

Figure 15 – Visual Representation of the Dynaic Analysis Performance Decay

Chapter 6 – Conclusions and Recommendations

6.1 Main Conclusions

The goal of the current work is the development of a proof of concept malware detection framework based on different methods of statistical analysis in an Android environment, and its resilience to continuously shifting malware trends. To achieve this, a predictive system was first designed to be capable of identifying malicious Android applications with the usage of three different kinds of analytical methods, through user feedback analysis, static code analysis, and finally, dynamic code analysis. This framework allows for an easier and automated detection of Android malware by reducing the manpower needed to scrutinize an entire app store as well as providing a more user-safe environment. As such, this framework presents a possible improvement on already existing methods present on contemporary app stores, alleviating user concerns by reducing the amount of malware they might stumble upon, as well as, focusing the work of security experts on more complex forms of malware that might go unnoticed by traditional methods. The proposed system can also adapt to new forms of malware as they become more and more popular amongst malware developers, by the means of retraining, therefore, accompanying malware's evolution in complexity and adaptation to new technological improvements.

With the purpose of developing the best malware detector for the three instances mentioned, this framework was fashioned to generate and test several types of models under differing circumstances. As such, the datasets were subject to different kinds of data preprocessing techniques to attain the best malware classifier. Subsequently, the algorithms were trained and tested using the full datasets, as well as a following study on their performance decay when trained with fixed monthly sets and tested on the months that followed. The algorithms used to develop the most effective malware classifier were the XGBoost, Random Forest, Support Vector Machine, K-Nearest Neighbour, Gaussian Naïve Bayes, and Bernoulli Naïve Bayes, with their performance measured by F_1 Score, Accuracy, FPR and FNR, AUC ROC and AUC P-R.

As mentioned, the study aimed at detecting malicious Android applications from three types of historical data. The first one, user feedback data, was gathered from Aptoide within a time interval of four months, ranging from October 2019 to January 2020. This

set of data allowed for a deeper understanding of the relationship between applications and the feedback users gave them by demonstrating certain feature patterns.

Training the models with separate user feedback feature groups, such as numerical ratings and symbolic flags yielded subpar results, mainly since each of these groups presented a high degree of correlation between the features which comprised them. The relationship between the different numerical ratings is but a causal one, linked by the popularity of the app and the percentage of users that like/dislike said app, which usually remains stable. The same can be said, to a lesser extent, about the symbolic flags feature group. Although numerical ratings, and to some extent, symbolic flags have a certain degree of correlation in their own categories, combining both of these types of features allows the machine learning classifiers to detect relevant patterns that enable a reasonable malware detection rate, with results of up to 79% F_1 Score and 86% AUC ROC. The performance decay study of this scenario showed a decline in overall performance when the models were tested on the months that follow the training ones, however, when trained with three months of data and tested with the final fourth month, the results showed sufficient improvement to suggest that with a wider time frame of training data better results could be reached.

In the second instance, historical data of static code analysis was used. This data set was comprised of six months of data, ranging from October 2019 to March 2020, gathered from Aptoide's internal security team. In this case, the features presented themselves as having differing levels of correlation between each another, this meant that each feature was able to contribute relevant information for pattern detection. This combination of features allows the algorithms to consistently detect malware apps, with results of up to 86% F1 Score and 91% AUC ROC. Following this, the model performance decay analysis showed a sudden decline of performance when trained with fewer than five months of historical data and applied to the following months. Training with five months of data and testing on the sixth yielded results that started to approximate the original model, suggesting the possibility that with more training data the models would become more resilient to changes.

In the third and final instance, historical data of dynamic code analysis was used. This dataset consisted of six months of historical data, ranging from February 2020 to July 2020, gathered from the Koodous collaborative platform for Android malware research.

In this case, using a combination of features assembled from multiple Android analysis tools made possible a statistical analysis of android apps from different types of activities, services, processes, and accesses. This in turn helped aggregate apps that behaved in similar malicious manners, therefore, making it easier for the classifiers to separate these groups from the ones that behaved in standard fashion. This combination of features gathered allows the algorithms to consistently detect malware apps, with results of up to 94% F_1 Score and 97% AUC ROC. Following this, the model performance decay analysis showed a sudden decline of performance when trained with fewer than five months of historical data and applied to the following months. Training with five months of data and testing on the sixth yielded results that started to approximate the original model, suggesting the possibility that with more training data the models would become more robust to changes.

Regarding the research questions that were set out in the beginning of this work, it was demonstrated that symbolical representations of user feedback such as star ratings and flags can be used to train malware classifiers, to a limited extent, and that both feature groups are relevant to this endeavor, particulary the Fake and Virus flags and the 3- and 5-star ratings. Following this it was demonstrated that Android analysis tools are able produce relevant information to train malware detection systems when their outputs are converted into count-based features. Finally, the best approach to retrain this system to maintain high performance over time was not achieved, mainly due to the need for a dataset with a wider time period, and also a need for a more comprehensive analysis on the variations of both train and test time windows.

6.2 Study Limitations

This framework hinges on model optimization by combining different models, parameters and different preprocessing techniques to output a single most optimized model pipeline. However, several decisions were made with the assumption of common data mining practices in mind, of which there is very little to none theoretical work available to study their effectiveness and impact. In this work the decision to select a K-fold cross validation scheme with a K of ten was made according to standard practices in the industry for this type of dataset. The same goes for the choice of withholding twenty percent of the datasets for validation. Joining these, but more specific to this knowledge

domain, is the malware to goodware ratios used in the training datasets. Although these are considered standard practices, there exists no systematic research on the best practices of these parameters.

Another limitation was the computation power and time available to run the tests. With this constraint, more algorithms from different machine learning schools could be applied and their results compared. Alongside this, the ability to train more candidates through the random grid search would be possible, and therefore, cover a larger spectrum of possible parameter combinations for each of the models and delivering more optimized and consistent results.

Finally, the various datasets lacked app coherence, in a sense that each of the datasets were comprised of different applications. More notably, the set used for the dynamic analysis approach was vastly different from the other two given its originating source. This led to an inability to directly compare the chosen analysis methods.

6.3 Future Research Proposals

As previously stated, many improvements could still be made to the proposed framework, namely the automation of the training of malware classification models at set intervals, or even online training, to keep this system as an always active and updated security measure, as a means to develop a more hands-off approach to malware detection.

Regarding the usage of user generated data to improve malware classification, utilizing app store comments to create new features could help improve detection rates. Text mining processes like sentiment analysis and document clustering could extract potentially beneficial information to improve malware detection in this environment. Along with this, a user centric approach to rate the validity their feedback would lessen the impact of feedback given from bots and spammers as well as elevate the influence of authentic users.

Studying the output of other Android app analysis tools could provide additional relevant features and information to the system. The combination of user feedback, the static code analysis and dynamic code analysis features could also be useful, not only in adding more complex patterns for the models to learn on, but with more features available, selecting more relevant ones and eliminating redundant ones to reduce noise and unneeded information which could improve efficiency and detection rates. Another

possibility is the development of a triage architecture that employs each model in a cascading fashion to avoid wasting the more resource costly methods of dynamic analysis on already detected malware.

Finally, a study on the effects of the variation of test horizons, from days to weeks, to establish the ideal rate at which the models would need to be retrained before starting to lose performance.

Bibliography

Allix, Kevin, Tegawendé F. Bissyande, Jaques Klein, and Yves Le Traon. 2014. "Machine Learning-Based Malware Detection for Android Applications: History Matters!"

Altman, Naomi S. 1992. "An introduction to kernel and nearest-neighbor nonparametric regression." *The American Statistician*, 175-185.

Android Developers. 2016. *UI/Application Exerciser Monkey.* Accessed December 2019. https://developer.android.com/studio/test/monkey.

Antonio, Nuno, Ana Maria de Almeida, Luís Nunes, Fernando Batista, and Ricardo Ribeiro. 2018. "Hotel online reviews: creating a multi-source aggregated index." *International Journal of Contemporary Hospitality Management.*

Arp, Daniel, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, e Konrad Rieck. 2014. "Effective and explainable detection of Android malware in your pocket." *Proceedings of the Network and Distributed System (NDSS).*

Azim, Tanzirul, and IIulian G. Neamtiu. 2013. "Targeted and depth-first exploration for systematic testing of android apps." *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, October: 641-660.

Bergstra, James, and Yoshua Bengio. 2012. "Random search for hyper-parameter optimization." *The Journal of Machine Learning Research*, 281-305.

Bläsing, Thomas, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. 2010. "An android application sandbox system for suspicious software detection." *2010 5th International Conference on Malicious and Unwanted Software*, October: 55-62.

Breiman, Leo. 2001. "Random forests." *Machine learning*, 5-32.

Brumley, David, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Song Dawn, and Heng Yin. 2007. "BitScope: Automatically dissecting malicious binaries." Technical Report, School of Computer Science, Carnegie Mellon University.

Buitinck, Lars, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, et al. 2013. "API design for machine learning software: experiences from the scikit-learn project." *arXiv preprint.*

Chakradeo, Saurabh, Bradley Reaves, Patrick Traynor, and William Enck. 2013. "Mast: Triage for market-scale mobile malware analysis." *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, 13-24.

Chapman, Pete, Julian Clinton, Randy Kerber, Thomas Khabaza, Thomas Reinartz, Colin Shearer, and Rüdiger Wirth. 2000. *CRISP-DM 1.0: Step-by-step data mining guide.* The Modeling Agency. Accessed September 23, 2020. https://the-modeling-agency.com/crisp-dm.pdf.

Chen, Tianqi, and Carlos Guestrin. 2016. "Xgboost: A scalable tree boosting system." *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, August: 785-794.

Chen, Wei, David Aspinall, Andrew D. Gordon, Charles Sutton, e Igor Muttik. 2016. "More semantics more robust: Improving android malware classifiers." *Proceedings of the 9th ACM conference on security & privacy in wireless and mobile networks*, July: 147-158.

Chipounov, Vitaly, Volodymyr Kuznetsov, and George Candea. 2011. “S2E: A platform for in-vivo multi-path analysis of software systems.” *Acm Sigplan Notices*, 265-278.

Deo, Amit, Santanu Dash, Guillermo Suarez-Tangil, Volodya Vovk, and Lorenzo Cavallaro. 2016. “Prescience: Probabilistic guidance on the retraining conundrum for malware detection.” *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, October: 71-82.

Desnos, Anthony. 2012. “Androguard.” *code.google.com.* Accessed June 2020. https://code.google.com/archive/p/androguard/.

Duque, João, Gonçalo Mendes, Luís Nunes, Ana Maria Almeida, and Carlos Serrão. 2020. “Automated android malware detection using user feedback.” *Journal of Cybersecurity (waiting acceptance).*

Duque, João, Gonçalo Mendes, Luís Nunes, Ana Maria de Almeida, and Carlos Serrão. 2020. “Automated android malware detection: system retraining.” *Journal of Cybersecurity (waiting acceptance).*

Elman, Jeffrey L. 1990. “Finding structure in time.” *Cognitive science*, 179-211.

Enck, W, and P McDaniel. 2010. “Not So Great Expectations: Why Application Markets Haven't Failed Security.” *IEEE Security and Privacy* (IEEE Security and Privacy).

Enck, William, Machigar Ongtang, and Patrick McDaniel. 2009. “On Lightweight Mobile Phone Application Certification.” *Proceedings of the 16th ACM Conference on Computer and Communications Security CCS.* Association for Computing Machinery.

Eshleman, Ryan M., and Hui Yang. 2014. “"Hey# 311, Come Clean My Street!": A Spatio-temporal Sentiment Analysis of Twitter Data and 311 Civil Complaints.” *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, December: 477.484.

Filgueiras, João, Luís Barbosa, Gil Rocha, Henrique L. Cardoso, Luís P. Reis, João P. Machado, and Ana M. Oliveira. 2019. “Complaint Analysis and Classification for Economic and Food Safety.” *Proceedings of the Second Workshop on Economics and Natural Language Processing*, November: 51-60.

Forte, Ana C., and Pavel Brazdil. 2016. “Determining the level of clients' dissatisfaction from their commentaries.” *International Conference on Computational Processing of the Portuguese Language*, July: 74-85.

Google. 2018. “Android Security & Privacy 2018 Year In Review.” *source.android.com.* Accessed June 2020. https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf.

Hunter, John D. 2007. “Matplotlib: A 2D Graphics Environment.” *Computing in Science & Engineering* 90-95.

InformationWeek. 2014. *Cybercrime black markets grow up.* Accessed December 2019. www.informationweek.com/cybercrime-black-markets-grow-up/d/d-id/1127911.

JesusFreke. 2009. *Smali.* Accessed June 2020. https://github.com/JesusFreke/smali.

Jolliffe, Ian T., and Jorge Cadima. 2016. “Principal component analysis: a review and recent developments.” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences.*

Kiritchenko, Svetlana, Xiaodan Zhu, Colin Cherry, and Saif M. Mohammad. 2014. “NRC-Canada-2014: Detecting aspects and sentiment in customer reviews.” *Proceedings of the 8th international workshop on semantic evaluation (SemEval 2014)*, 437-442.

Kohavi, Ron. 1995. "A study of cross-validation and bootstrap for accuracy estimation and model selection." *Ijcai*, August: 1137-1145.

Koodous. 2018. *Koodous: Online malware analysis platform.* Accessed December 2019. https://koodous.com/.

Lantz, Patrick. 2011. "Droidbox." *code.google.com.* Accessed June 2020. https://code.google.com/archive/p/droidbox/.

Liu, Yu, Yichi Zhang, Haibin Li, and Xu Chen. 2016. "A hybrid malware detecting scheme for mobile Android applications." *2016 IEEE International Conference on Consumer Electronics (ICCE)*, January: 155-156.

Lopes, João. 2020. "Malware detection methods for Android mobile applications." *ISCTE (under evaluation).*

Maaten, Laurens van der., and Geoffrey Hinton. 2008. "Visualizing data using t-SNE." *Journal of machine learning research*, Nov: 2579-2605.

Machiry, Aravind, Rohan Tahiliani, and Mayur Naik. 2013. "Dynodroid: An input generation system for android apps." *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, August: 224-234.

Mahmood, Riyadh, Nariman Mirzaei, and Sam Malek. 2014. "Evodroid: Segmented evolutionary testing of android apps." *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November: 599-609.

Manning, Christopher D., Hinrich Schütze, and Prabhakar Raghavan. 2008. "Introduction to information retrieval." *Cambridge University press.*

McAfee. 2019. "McAfee: Mobile Threats Report." *www.mcafee.com.* Accessed April 20, 2020. https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007.

McKinney, Wes. 2010. "Data Structures for Statistical Computing in Python." *Proceedings of the 9th Python in Science Conference*, 51-56.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeffrey Dean. 2013. "Distributed representations of words and phrases and their compositionality." *Advances in neural information processing systems*, 3111-3119.

Moser, A., C. Kruegel, and E. Kirda. 2007. "Limits of static analysis for malware detection." *Twenty-Third Annual Computer Security Applications Conference (ACSAC)*, December: 421-430.

Moser, Andreas, Christopher Kruegel, and Engin Kirda. 2007. "Exploring multiple execution paths for malware analysis." *2007 IEEE Symposium on Security and Privacy (SP'07)*, May: 231-245.

Nigam, Ruchna. 2015. "A Timeline Of Mobile Botnets." *Virus Bulletin.*

Ordenes, Francisco V., Babis Theodoulidis, Jamie Burton, Thorsten Gruber, and Mohamed Zaki. 2014. "Analyzing customer experience feedback using text mining: A linguistics-based approach." *Journal of Service Research*, 278-295.

Pandita, Rahul, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. "WHYPER: Towards automating risk assessment of mobile applications." *22nd USENIX Security Symposium (USENIX Security 13)*, 527-542.

Peiravian, Nasser, and Xingquan Zhu. 2013. "Machine learning for android malware detection using permission and api calls." *IEEE 25th international conference on tools with artificial intelligence*, November: 300-305.

Petz, Gerald, Michal Karpowicz, Harald Fürschuß, Andreas Auinger, Václav Stříteský, and Andreas Holzinger. 2013. "Opinion mining on the web 2.0–characteristics of user generated content and their impacts." *International Workshop on*

Human-Computer Interaction and Knowledge Discovery in Complex, Unstructured, Big Data, July: 35-46.

Revivo, Idan, and Ofer Caspi. 2014. *CuckooDroid.* Accessed June 2020. https://github.com/idanr1986/cuckoo-droid.

Rossum, Guido van. 1995. *Python tutorial.* Technical Report, Amsterdam: Centrum voor Wiskunde en Informatica (CWI).

Roy, Sankaras, Jordan DeLoach, Yuping Li, Nicolae Herndon, Doina Caragea, Xinming Ou, Venkatesh-Prasad Ranganath, Hongmin Li, and Nicolais L. Guevara. 2015. "Experimental study with real-world data for android app security analysis using machine learning." *Proceedings of the 31st Annual Computer Security Applications Conference*, December: 81-90.

Sahs, Justin, and Latifur Khan. 2012. "A machine learning approach to android malware detection." *European Intelligence and Security Informatics Conference*, 141-147.

Sanz, Borja, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo G. Bringas, and Gonzalo Álvarez. 2013. "Puma: Permission usage to detect malware in android." *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*, 289-298.

Singh, Anshuman, Andrew Walenstein, and Arun Lakhotia. 2012. "Tracking concept drift in malware families." *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, October: 81-92.

Spreitzenbarth, Michael, and Felix Freiling. 2012. *Android Malware on the Rise.* Technical report, Department Informatik, Friedrich Alexander Universität Erlangen Nürnberg (FAU). Accessed April 2020. https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/2210.

Spreitzenbarth, Michael, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. 2013. "Mobile-sandbox: having a deeper look into android applications." *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, March: 1808-1815.

StatCounter. 2019. *StatCounter: Mobile Operating System Market Share Worldwide.* Accessed April 20 2020. https://gs.statcounter.com/os-market-share/mobile/worldwide.

Statista. 2020. *Statista: Number of smartphones sold to end users worldwide from 2007 to 2020.* Accessed April 22, 2020. https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007.

Tam, Kimberly, Ali Feizollah, Nor B. Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. "The evolution of android malware and android analysis techniques." *ACM Computing Surveys (CSUR)*, 1-41.

Tam, Kimberly, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. "Copperdroid: Automatic reconstruction of android malware behaviors." *Ndss*, February.

The Register. 2013. *The Register: Earn 8,000 a month with bogus apps from Russian malware factories.* Accessed April 2020. https://www.theregister.com/2013/08/05/mobile_malware_lookout/.

Tukey, John W. 1962. "The future of data analysis." *The annals of mathematical statistics* 1-67.

Vapnik, Vladimir, Steven Golowich, and Alex Smola. 1997. "Support vector method for function approximation, regression estimation and signal processing." *Advances in neural information processing systems*, 281-287.

Walt, Stéfan J. van der, S. Chris Colbert, and Gael Varoquaux. 2011. "The NumPy array: a structure for efficient numerical computation." *Computing in science & engineering*, 22-30.

Wang, Lei, Tie Qiu, and Wenbing Zhao. 2018. "Quality, Reliability, Security and Robustness in Heterogeneous Systems." *15th EAI International Conference.*

Wold, Svante, Kim Esbensen, and Paul Geladi. 1987. "Principal component analysis." *Chemometrics and intelligent laboratory systems*, 37-52.

Wu, Dong-Jie, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. "Droidmat: Android malware detection through manifest and api calls tracing." *2012 Seventh Asia Joint Conference on Information Security*, August: 62-69.

Xu, Rubin, Saïdi Hassen, and Anderson Ross. 2012. "Aurasium: Practical policy enforcement for android applications." *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12).*

Yan, Lok Kwong, and Heng Yin. 2012. "Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis." *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 569-584.

Yeo, In-Kwon, and Richard A. Johnson. 2000. "A new family of power transformations to improve normality or symmetry." *Biometrika,*, 954-959.

Zheng, Cong, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, e Wei Zou. 2012. "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications." *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices.* 93-104.

Zhou, Wu, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. "Detecting repackaged smartphone applications in third-party android marketplaces." *Proceedings of the second ACM conference on Data and Application Security and Privacy*, February: 317-326.