Applications of Data analytics and Machine Learning tools to the enhanced design of modern communication networks and security applications

IGNACIO MARTÍN MARTÍNEZ

En cumplimiento parcial de los requisitos para el grado de Doctor en

INGENIERÍA TELEMÁTICA

Universidad Carlos III de Madrid

Director:

José Alberto Hernández Gutiérrez

Fecha de Defensa de La Tesis:

Junio de 2019

Copyright © 2019 Ignacio Martín Martínez

PUBLISHED BY UNIVERSIDAD CARLOS III DE MADRID

Licensed under the Creative Commons License version 3.0 under the terms of Attribution, Non-Commercial and No-Derivatives (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://creativecommons.org/licenses/by-nc-nd/3.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "as is" basis, without warranties or conditions of any kind, either express or implied. See the License for the specific language governing permissions and limitations under the License.

April 2019

To all my family, those we were
and those we will be.

Acknowledgements

Like in any long-term project, there are many persons to thank for, both for their day to day involvement and for their silent suffering of my errors, anxieties and bad moods arisen from such a big thing as writing a Thesis. The first person I would like to recognize is José Alberto, for being the best possible supervisor, supportive, trustworthy and teaching me to learn by doing rather than observing. I would also like to thank all ADSCOM group members for their help and advise, specially present in all those Friday meetings where an idea can be shown, discussed and very much improved. Furthermore, thank you to all members of the department for this friendly environment, specially to all PhD fellow students who have contributed to make days here better.

I would like to thank as well all the people I have worked with during this period, for making things easier specially when the circumstances where adverse. Thank you very much to Iyad Rahwan for letting me fulfill any engineer's dream during my stay at MIT with his team and to Nick Obradovich for his exceptional inputs and educational conversations during that time. Special mention here to Ángel, who trusted me with that project which is starting now to see the light and will produce fruitful results.

Finally, it is impossible no to recognize the infinite support of friends and family, who may not have been completely aware of it, but have helped a lot (what better way to close an sterile week at the office that a good Friday night dinner). Concisely, there are three persons I would like to acknowledge most in this category. Thanks mom for making me keep my feet on the ground with your cautious advise. Thanks dad for continuously reminding me how to use my brain, even when your wise inquiries make me rethink my last achievement all over again. And, of course, thanks to you, Pilar for being always there, forgiving and forgetting those concerns that turned unfairly against you and hearing my craziness out even when your batteries had run out.

To all of you,who have helped and are always there, thank you. This is your baby too.

Published and Presented Contents

- **Android malware detection from Google Play meta-data: selection of important features**; *A. Muñoz, I. Martín, A. Guzmán, J. A. Hernández* in IEEE Conf. Communications and Network Security (CNS'15). Florence, Italy. Sep 2015; doi: https://doi.org/10.1109/CNS.2015.7346893
  - Partially included in Chapter 4.
  - The student contributed with the design and implementation of the experiments regarding Step-AIC experiments mainly.
  - The material from this source and included in the thesis is not indicated neither by typographical means nor references.
- **Android Malware Characterization using Metadata and Machine Learning Techniques**; *I. Martín, A. Muñoz J. A. Hernández, A. Guzmán* in Security and Communication Networks, Hindawi, June 2018, vol 2018; doi: https://doi.org/10.1155/2018/5749481
  - Completely included in Chapter 4.
  - The student contributed with the design and implementation of the experiments, including the study of features, permission analysis and ML system development and performance assessment. Tables and Figures were also developed by the student who participated on the writing of the article.
  - The materials from this source and included in the thesis are not indicated neither by typographical means nor references.
- **Insights of Antivirus Relationships when Detecting Android Malware: A Data Analytics Approach**; *I. Martín, J. A. Hernández, S. Santos, A. Guzmán* in ACM Conf. Computer and Communications Security (CCS'16). Viena, Austria. Oct 2016, pages 1778-178; doi: https://doi.org/10.1145/2976749.2989038
  - Completely included in Chapter 5.
  - The author contributed with the design and implementation of the experiments, the results and conclusions, creating the figures and tables and in the writing of the article.
  - The materials from this source and included in the thesis are not indicated neither by typographical means nor references.
- **Analysis and Evaluation of Antivirus Engines in Detecting Android Malware: A Data Analytics Approach**, *I. Martín, J. A. Hernández, S. Santos*, in European Intelligence and Security Informatics Conference (EISIC'18). Karlskrona Sweden, Oct 2018

  - Completely included in Chapter 5.
  - The author contributed with the design and implementation of the all the experiments, the design and creation of the figures and tables and in the writing of the article.
  - The materials from this source and included in the thesis are not indicated neither by typographical means nor references.

- **SignatureMiner: A fast Anti-Virus signature intelligence tool**; *I. Martín, J. A. Hernández, S. Santos* in IEEE Conf. Communications and Network Security (CNS'18). Beijing, China. Apr 2018 **Best Poster Award**; doi: https://doi.org/10.1109/CNS.2018.8433141
  - Completely included in Chapter 6.
  - The author contributed in the article elaboration (writing, figures, tables...), the design of the solution and implemented most of the code available online at Github.
  - The materials from this source and included in the thesis are not indicated neither by typographical means nor references.

- **Machine Learning based analysis and Classification of Android Malware Signatures**; *I. Martín, J. A. Hernández, S. Santos*, in Elsevier Future Generation Computer Systems, August 2019, vol 97, pages 295-305; doi: https://doi.org/10.1016/j.future.2019.03.006
  - Completely included in Chapter 6.
  - The author contributed both in the article elaboration (writing, figures, tables...) and in the design and implementation of the analysis and experiments carried out in the article.
  - The materials from this source and included in the thesis are not indicated neither by typographical means nor references.

- **CloneSpot: Fast detection of Android Repackages**; *I. Martín, J. A. Hernández*, in Elsevier Future Generation Computer Systems, May 2019, vol 94, pages 740-748, doi: https://doi.org/10.1016/j.future.2018.12.050
  - Completely included in Chapter 7.
  - The author lead the article, implemented most of experiments and contributed to the design of experiments and solutions supervised by the thesis director.
  - The materials from this source and included in the thesis are not indicated neither by typographical means nor references.

- **Is Machine Learning Suitable for Solving RWA Problems in Optical Networks?** *I. Martín, J. A. Hernández, S. Troia, F. Musumeci, G. Maier, O. González de Dios*, in European Conference on Optical Communications (ECOC'18). Rome, Italy, Sept 2018; doi: https://doi.org/10.1109/ECOC.2018.8535562
  - Completely included in Chapter 8.

  - The author led together with his advisor this work, contributing in the design and implementation of the experiments as well as in the conclusions and the elaboration of the manuscript, figures and tables included.
  - The materials from this source and included in the thesis are not indicated neither by typographical means nor references.

- **Netgen: A Fast and Scalable Tool for the Generation and Labeling of Networking Datasets**, *I. Martín*, *J. A. Hernández, O. González de Dios*, under review.
  - Completely included in Chapter 8.
  - The author contributed in the design and implementation of the solution, including manuscript elaboration (including figures and tables) and code development.
  - The materials from this source and included in the thesis are not indicated neither by typographical means nor references.

- **Machine-Learning-Based Routing and Wavelength Assignment in Software-Defined Optical Networks**, *I. Martín*, *S. Troia, J. A. Hernández, Alberto Rodríguez, F. Musumeci, G. Maier, Rodolfo Alvizu, O. González de Dios*, under review.
  - Partially included in Chapter 8.
  - The author led together with his advisor this work, contributing in the design and implementation of the ML experiments and the parts regarding Netgen. The author led the writing of the paper, specially creating all figures and tables included from it in the thesis.
  - The materials from this source and included in the thesis are not indicated neither by typographical means nor references.

Abstract

Lately, *Artificial Intelligence* and *Machine Learning* (ML) have become game-changing technologies due to their ability to generalize from data and infer algorithmic behaviors that consider larger casuistic that humans are able to. In short, these technologies pursue the installation of human-like intelligence to computer tasks so they can overtake different functions. Despite, their implantation and development in many fields is still too early stage, not to mention the requirements and needs they entail.

Therefore, the aim of this thesis is to advance in the application of these technologies and for that we will consider an specific field: The Internet Infrastructure. To this aim, contributions focus on two main specific areas, namely cybersecurity and optical WDM networks.

On the security side, we propose a new approach for malware detection and application quality assessment that relies in application *meta-information*, that is, the data describing the application (such as description, category, permissions...) instead of application code. This approach is detailed and validated in two specific applications: ML-based detection of malware and scalable repackaging detection through meta-data semantic clustering.

The first application consists on the usage of meta-data as Machine Learning features with a labeled collection of malware applications to detect whether they are malware or not. Resulting algorithms are capable of detecting malware to a good extent in certain conditions, reaching F-score values of nearly 0.9.

Arising from the observations from Machine Learning analysis, Antivirus (AV) engines coming from multi-scanner tools are inspected using data analytics and AI technologies aiming at the understanding of their lack of consensus at the detection and categorization levels. The main aim for this study is twofold: advancing on the understanding of AV detection patterns and policies and the improvement multi-engine detection by proposing different aggregation and cleaning tools.

Initially, AV engine detections are inspected, showing that most engines disagree when detecting malware to the extent of not completely agreeing in the detection of a single application. Moreover, different detection patterns are observed, namely leader, follower and eccentric engines. At the end, an estimation of the risk of malware per application based on Structural Equation models is proposed.

On the family side, we propose a lightweight categorization scheme that achieves comparable scores to other alternatives in the literature at a smaller train cost: SignatureMiner. Using such system, we normalize and categorize AV signatures into 41 distinct families and three broader categories, namely adware, harmful and unknown. Then, an ML classifier to assign and specific category to unknown malware is proposed with high performance.

Another application explored for meta-data is that of repackaging detection. Using similarity clustering, a large collection of unlabeled applications from Google

Play are inspected and compared to detect potential repackaged applications and their victims. This approach is capable to unveil nearly 420K applications potentially cloned within the Google Play application market.

On the network side, we contribute to the introduction of Machine Learning in the field by proposing an integral pipeline framework that improves the development of ML-powered network protocols as enhanced heuristics that emulate optimal solutions in many areas. Such framework is based on data generation, modeling and validation and network implementation. In this thesis, we focus on the first two steps by developing proof of concept solutions for both.

Dataset generation and data labeling is addressed with Netgen, a versatile network data generator based on Net2Plan. Netgen functionality is presented and performance and abilities demonstrated. Finally, this thesis addresses the modeling of Routing and Wavelength Assignment (RWA) in its ILP version as an ML problem. The assumption is that ML can be useful to develop an ML-powered heuristic for RWA that performs better than regular heuristics and much faster than ILP and heuristics. Results support the viability of this approach, opening the scheme for other complex network protocols.

In sum, this thesis builds different AI-based components to enhance the functionalities and capabilities of different elements in the proposed fields, defining systematic approaches and methodologies to this aim. That way, all works in this document contribute to the design and development of the concept of *AI as a Service* (AIaaS), that proposes a paradigm for the integration of AI technologies over specific knowledge areas with limited expertise in both AI and the specific area.

Contents

Acknowledgements vii

Abstract xiii

1 Introduction and Motivation 1

1.1 Brief History of Artificial Intelligence 1

1.2 Motivation and Overview 3

1.3 Thesis Objectives and Goals 5

1.4 Thesis Structure and Contributions 6

2 State of The Art 9

2.1 Security in the Android Ecosystem 9

2.1.1 Android Malware Detection 10

2.1.2 Repackaging Detection 14

2.2 Antivirus Analysis and their Detections 18

2.3 ML Applied to Optical WDM Networks 25

2.4 Conclusions and Progress Beyond the State of the Art 32

3 Methodology and Tools 35

3.1 Data Mining 36

3.1.1 Distances and Item Similarity 36

3.1.2 Locality-Sensitive Hashing Methods 37

3.1.3 Market Basket Analysis and Frequent Itemset Mining 38

3.2 Machine Learning 39

3.2.1 Supervised Problems 40

3.2.2 Unsupervised Problems 41

3.2.3 Machine Learning Workflow 42

3.2.4 Machine Learning Performance Measurement 44

3.2.5 Feature Selection 46

3.2.6 Deep Learning 47

3.3 Other Statistical Methods 49

3.3.1 Latent Variable Models 49
3.3.2 Graph Modeling 49
3.4 Summary and Conclusions 50

4 Android Meta-data for Malware Detection 53
4.1 Android Application Meta-data 53
4.2 Meta-data Collection for Malware Detection 55
4.2.1 Intrinsic Application Features 56
4.2.2 Social-related Features 57
4.2.3 Entity-related Features: Developers and Certificate Issuers . . 57
4.2.4 Malware Detection Attributes 58
4.2.5 Dataset Benchmark 58
4.3 Analysis of Meta-data Features in Legitimate and Malware Applications 59
4.3.1 Predictive Power of Permissions 61
4.3.2 Feature Importance and Selection: A Machine Learning Approach 62
4.4 Malware Detection Model 64
4.4.1 Determining Model Size 64
4.4.2 Solution Modeling and Results 65
4.4.3 Robustness of the Models 65
4.4.4 Performance and Computational Time 67
4.5 Summary and Conclusions 68

5 Data-driven Interrelation Analysis of AV engines 69
5.1 Detection Matrix and Dataset Insights 70
5.2 Peer Relations among AV Engines 72
5.2.1 Followers and Eccentrics 72
5.2.2 Principal Component Analysis 73
5.2.3 Correlation between Engines 74
5.2.4 Association Rule Learning 75
5.3 Latent Variable Modeling for Malware Risk Assessment 77
5.4 Summary and Conclusions 80

6 AV Label-based Analysis of Malware Families 83
6.1 SignatureMiner: A fast Anti-Virus Signature Intelligence Tool 84

6.1.1 SignatureMiner Performance 86
6.2 Malware Categorization and Classes using SignatureMiner 86
6.3 Malware Family Classes and Categories Interdependences 91
6.3.1 Correlation of Malware Categories 91
6.3.2 Graph Community Clustering to Detect Class Redundancies . 92
6.3.3 Grouping AVs by their Detection Schemes 93
6.4 Identifying Unknown Malware 95
6.5 Summary and Conclusions 97

7 Android Meta-data for Repackaging Detection 99
7.1 Meta-data Application Collection 99
7.2 CloneSpot: Fast Detection of Application Duplicates 100
7.2.1 Clustering Qualitative Meta-data through Min-hashing 101
7.2.2 Market-scale Detection of Application Duplicates 102
7.2.3 Intra-group Detection Scoring 102
7.3 CloneSpot service: Fast Retrieval of Potential Clones 106
7.3.1 Application Removal in Google Play 107
7.3.2 Real-time Repackaging Detection through the CloneSpot Service 109
7.4 Summary and Conclusions 111

8 Optical WDM Networks Configuration from an ML perspective 113
8.1 Netgen: Automated Tool for Network Data Generation 114
8.1.1 Net2Plan: An Open-source Network Planner 115
8.1.2 Netgen Architecture 115
8.2 Modeling RWA as an ML classifier 117
8.2.1 Methodology 118
8.2.2 Dataset Generation and Labeling 119
8.2.3 Machine Learning Classification Models 121
8.3 Results 122
8.3.1 Dataset Generation Efficiency of Netgen 122
8.3.2 ML-based Heuristic to Solve RWA (Spanish 5 Node topology) 123
8.3.3 ML-based Heuristic to Solve RWA (Abilene topology) 124
8.3.4 Complexity and Time Impact of ML Predictions 125
8.4 Summary and Conclusions 127

9 Conclusions and Future Work **129**

9.1 Summary of Main Contributions 129

9.2 Future Work 132

9.3 List of Publications during Thesis Period 133

Bibliography **137**

List of Figures

1.1 Main Events in AI history 2

2.1 Multiscanner tools 24
2.2 Summary of ML methods in Networking 32

3.1 Neural network example 48
3.2 Neural network example 50

4.1 Malware detections per application 58
4.2 Feature comparison 59
4.3 Feature hashing permissions 61
4.4 Filter analysis of features 63
4.5 Incremental models 64

5.1 Detections per sample 70
5.2 Engine activity 71
5.3 PCA of AV engines 73
5.4 Engine correlation 74
5.5 Detailed correlations 75
5.6 Graph scheme of Followers 77
5.7 Logistic Z_{sem} 79
5.8 Logistic Z_{sem} distributions 80

6.1 Most Frequent AV Signatures 85
6.2 Per-family AV detection frequencies 90
6.3 Communities in malware families 92
6.4 AV communities (Family patterns) 94
6.5 AV weights (LR) 96

7.1 ASI score rankings 105
7.2 Repackaging example 1 105
7.3 App-set size distribution 108

7.4 Repackaging example 2 109

8.1 NetGen diagram 116
8.2 Studied topology 117
8.3 Abilene Network Illustration 120
8.4 Accuracy and loss for DNN 122

List of Tables

2.1 SoA for malware detection 17

3.1 Truth Table 46

4.1 Dataset overview 56
4.2 Summary of meta-data features 60
4.3 Step-AIC selection results 62
4.4 Benchmark Results 66
4.5 Robustness Test 67

5.1 Association rules 76
5.2 AV coefficients 78

6.1 SignatureMiner rules 88
6.2 Category Correlation 91
6.3 Classifier results 96
6.4 Harmful amounts in unknown samples 97

7.1 Summary of features 100

8.1 Time consumption of NetGen 123
8.2 5-Node results 123
8.3 Abilene results 124
8.4 Complexity of ML-RWA solution 126

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ML	**M**achine **L**earning
AGI	**A**rtificial **G**eneral **I**ntelligence
AV	**A**nti**V**irus (engine)
PoC	**P**roof **o**f **C**oncept
API	**A**pplication **P**rogramming **I**nterface
GUI	**G**raphical **U**ser **I**nterface
CLI	**C**ommand **L**ine **I**nterface
TPU	**T**ensor **P**rocessing **U**nit
CSV	**C**omma **S**epparated **V**alues
JSON	**J**ava**S**cript **O**bject **N**ode
KDD	**K**nowledge **D**iscovery in **D**atabases
DL	**D**eep **L**earning
RL	**R**einforcement **L**earning
BoW	**B**ag **o**f **W**ords
HMM	**H**idden **M**arkov **M**odel
LR	**L**ogistic **R**egression
SVM	**S**upport **V**ector **M**achine
DT	**D**ecision **T**ree
RF	**R**andom **F**orest
DNN	**D**eep **N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etwork
PCA	**P**rincipal **C**omponent **A**nalysis
SEM	**S**tructural **E**quation **M**odel
LSH	**L**ocal **S**ensitive **H**ashing
MSE	**M**ean **S**quared **E**rror
MAE	**M**ean **A**bsolute **E**rror
RMSE	**R**oot **M**ean **A**bsolute **E**rror
TP	**T**rue **P**ositive
TN	**T**rue **N**egative
FP	**F**alse **P**ositive
FN	**F**alse **N**egative
PR	**P**recision **R**ecall
AIC	**A**ikake **I**nformation **C**riteria
ROC	**R**eceiver **O**pperation **C**haracteristic
AUC	**A**rea **U**nder the **C**urve
LVM	**L**atent **V**ariable **M**odel
SEM	**S**tructural **E**quation the **M**odel
ASI	**A**pplication **S**imilarity **I**ndex
APK	**A**ndroid **AP**plication **P**ac**K**age

IPS	**I**ntrusion A**P**rotection **S**ystem
SM	**S**ignature **M**iner
IP	**I**nternetworking **P**rotocol
TM	**T**raffic **M**atrix
DPI	**D**eep **P**acket **I**nspection
OBS	**O**ptical **B**urst **S**witching
BER	**B**it **E**rror **R**ate
QoS	**Q**uality **o**f **S**ervice
QoE	**Q**uality **o**f **E**xperience
SDN	**S**oftware **D**efined **N**etworking
NFV	**N**etwork **F**unction **V**irtualization
BoF	**B**ag **o**f **F**lows
WDM	**W**avelength **D**ivision **M**ultiplexing
EON	**E**lastic **O**ptical **N**etworks
ILP	**I**nteger **L**inear **P**rogramming
RWA	**R**outing and **W**avelength **A**ssignment
RMSA	**R**outing and **M**odullation **S**pectrum **A**ssignment
RWC	**R**outing and **W**avelength **C**onfiguration

Chapter 1

Introduction and Motivation

This chapter devotes to the introduction, the historical background and the motivation of this thesis, highlighting contributions and objectives. Specifically, Section 1.1 begins with the history of AI and its initial developments, while Section 1.2 introduces the main motivations behind the thesis. More specifically, Section 1.3 clearly states the objectives pursued upon this document. Finally, Section 1.4 overviews the structure and content of the document.

1.1 Brief History of Artificial Intelligence

Artificial Intelligence (AI) comprises a set of technologies that attempt the emulation of human intelligence in computer systems. The AI term was coined in 1956 by John McCarthy, Marvin Minsky and Claude Shannon, even though work had begun in the early 50s. At its origin, AI attempted to create computer programs that had real intelligence, as if they were humans, and for that purpose, different methods and algorithms have been explored in detail; concisely *Information Retrieval* algorithms

near the 70s, that have been very useful in the development of search engines or large database organization, and *Machine Learning* algorithms that learn patterns from data and gained popularity around the 80s, even though they have become much more relevant today.

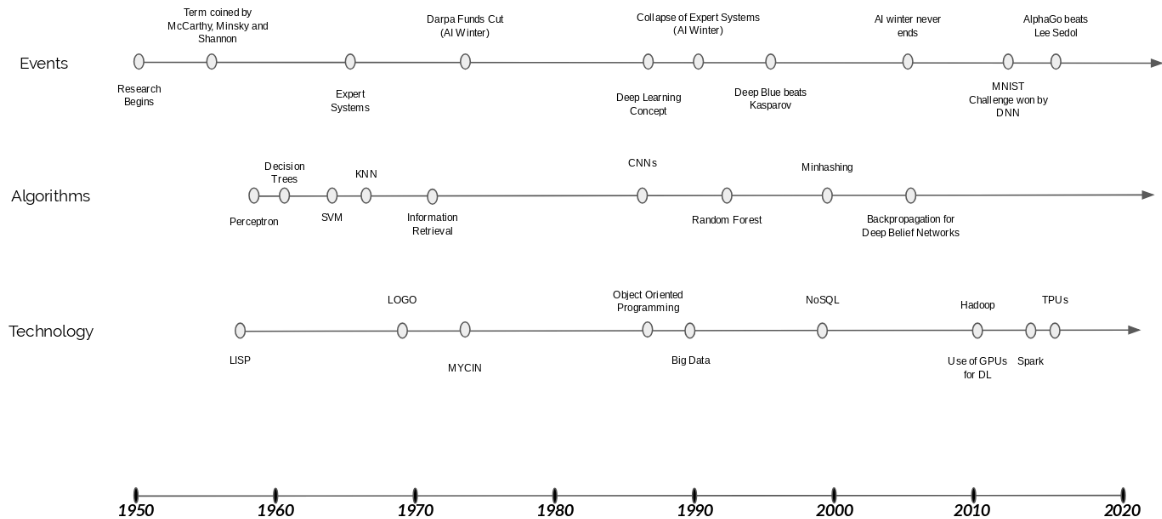


FIGURE 1.1: The timeline in the figure reflects some of the most relevant milestones in the AI history. Such events are separated among global events, algorithms and technologies.

The field of AI advanced quickly with the apparition of the previous key subfields which have procured algorithms to deal with a very broad range of *intelligent* tasks, such as performing semantic web search, numerical predictions or even process images or audio. Indeed, one of the key components in the development of AI has been the advent of *Machine Learning* (ML), which attempts to make machines *learn* patterns and behaviors from large data sources in such a way they are able to perform decisions based on them in future events.

Actually, Machine Learning has had a huge impact on data analysis and, ultimately, on Artificial Intelligence itself, ranging from simple linear regression to much more complex and powerful *Deep Learning*. At its origins, ML was a family of useful algorithms which perform *descriptive analytics*. Regression and classification algorithms were mainly used to understand and describe datasets rather than providing *predictive analytics*. From 2010, with the advent of *Big Data*, computing capacity has increased and ML has become a really powerful tool to assist machines undertaking more advanced tasks, such as prediction or decision making.

In spite of its early advancements and abundant milestones, there have been moments in history where AI has experienced limitations and its capabilities have stalled. This has given birth to *AI winters* where research has been almost completely abandoned. These winters have occurred for different reasons, mainly regarding profound limitations algorithms have encountered during time. For instance, one of these AI winters occurred around the late 80s and was mainly due to the collapse of the *expert systems* as they were rather limited by the restricted hardware technology and the lack of consistent and labeled datasets at that time.

Nowadays, we are experiencing a new *AI Spring*: data sources are limitless thanks to the Internet; hardware has evolved and is driving the discovery of new powerful and computationally intensive algorithms, such as *Deep Learning*. Researchers have achieved stunning breakthroughs in the last decade, by solving complex tasks, such as Image recognition [1], Speech recognition [2] or even beating humans at computationally unfeasible games, like Go [3].

As a result, new algorithms, technologies and hardware are created every year enabling new and enhanced applications. Together with a thriving ecosystem populated by a plethora of companies, investors, platforms and technologies, the field is advancing very fast and enabling the improvement of a growing number of different research areas.

As a summary, Fig 1.1 locates the main events in the recent AI history in time along with the most relevant algorithmic and technological breakthroughs and milestones. It can be observed that most theoretical advancements were ready around the nineties, even though full progress has been impossible until the vast improvement of computing capabilities.

1.2 Motivation and Overview

Artificial Intelligence is a game-changing technology that can revolutionize every single aspect of daily life, from clothing design and supply chain management to autonomous vehicles or automated translation. Despite, AI technologies are not trivial to use and hence, their development is still in a very early-stage in many fields and areas. Actually, problems different to classification or regression require expert practitioners in each area capable of mapping constrained problems into the rigid structure of classification/regression. This is very hard in many fields where workers' profiles are not experts on data analysis and statistics.

Moreover, it is worth remarking that many of these problems come from fields with data-less backgrounds, so the introduction of AI technologies does require a paradigm shift where existing tools must be adapted to include data collection and analytics policies among other requisites.

In this light, this thesis is proposed to advance the adoption of AI technologies by introducing *intelligent components* in two Internet-related fields: Cybersecurity and Optical WDM Networks. Both areas are key in the development of the industrial Internet and can be deeply optimized by AI to generate better expert systems that reduce operational costs and optimize processes. Furthermore, they have both historically neglected data collection and monitoring in their problem resolution process, so the introduction of any data-driven application must necessarily contemplate data collection and labeling procedures.

In the field of cybersecurity, we focus on Android malware, which has emerged as a consequence of smartphone popularity worldwide. Android has a flamboyant ecosystem crowded with applications, devices and markets which has attracted malware developers too. In this thesis, we propose a novel method to search, identify and detect malware based on *meta-data*, that is, all those inherent application

data pieces not included in the source code, but informative of the application such as the title, description or category.

While previous work mainly focuses on other characteristics of Android apps, like application code or behavior, our aim is to study meta-data and use it in the detection of malware applications. Hence, we first perform an in-depth analysis of quantitative meta-data features and their capabilities for detecting malware using ML algorithms without code inspection or sandboxing. Such capabilities have been demonstrated in a ML system powered by a random forest that can predict malware with an F-score value of 0.89.

From that initial work, we found that Android malware analysis has been partly sustained on *multi-scanner* detection tools which provide the detection information from several antivirus (AV) tools for a given malware sample. There has been discussion in the literature regarding the combination and utilization of such information to provide malware *ground truth*, which has been relevant within the scope of malware detection inside this thesis. Hence, we continue with the analysis of AV engines in large, including tools for AV signature normalization (*SignatureMiner*) as well as the detailed inspection of AV interrelations to unveil interesting patterns and groups of AV engines according to their behavior, namely *followers*, *leaders* and *eccentrics*

In addition, AV engines are also inspected according to the malware families they assign to different malware samples, including a proposal for a categorical distinction of malware families according to the origin and objectives of the different threats they contain. Up to 41 different malware families are extracted from this analysis along with four groups of AVs according to their detection behaviors with respect to families. At the end, an ML classifier that determines whether a sample is adware or harmful malware is proposed, achieving an F-score of up to 0.84.

Afterwards, we return to meta-data of qualitative type as a solution to facilitate and improve the detection of *repackaged* applications by means of data mining and locality sensitive hashing techniques, such as *min-hashing*. The detection of application clones which have been repackaged from legitimate ones is usually a hard task, specially inside of large markets with millions of applications. Our solution, the CloneSpot methodology, is proposed and validated over a 1.3M application-wide dataset, detecting up to 420K potential clone applications.

Finally, in the field of Optical WDM Networks, this thesis has surveyed the more recent works where AI in networking is getting relevance as a way to manage future networks together with the new *Software Defined Networking* paradigm. As a result, we propose a framework workflow for the application of ML algorithms from data generation to network implementation.

The objective for such framework is twofold: (i) define and specify the details for an ML network workflow in its different stages and (ii) demonstrate how classical protocols, such as *Routing and Wavelength Assignment* (RWA), can be modeled by ML for solution approximation. In this example, numerical results show that near-optimal RWA approximations can be obtained by machine learning, much faster than other classical solutions including Integer Linear Programming (ILP) or heuristics.

All these contributions have been developed with the common goal of applying AI/ML solutions to non-straightforward problems, so they have required specific modeling and the adaptation of different tools, algorithms and methodologies to the specific details of the problems addressed at each distinct chapter.

1.3 Thesis Objectives and Goals

The main contribution of this dissertation is the consolidation and application of the latest ML algorithms and methods to the enhancement of the Internet infrastructure by producing incremental advances in two main fields: Cybersecurity and Computer Networks.

- **Cybersecurity**: The main objective is to improve Android application security in the context of Google Play by analyzing the meta-data available in application markets and Antivirus signatures from multi-scanner tools. Main contributions:
    1. Data-driven inspection of meta-data features from Google Play, indicating which ones are the most relevant in the detection of malware.
    2. ML-powered detection of Android malware inside Google Play using as Ground Truth multi-scanner malware detection outcomes.
    3. In-depth analysis of multi-scanner tools to produce a methodological approach for enhanced malware labeling.
    4. Normalization of AV engine outputs and cross-engine analytics aiming at improving the categorization of malware samples.
    5. Detection of application clones, either repackaging or any other plagiarism attempt by clustering applications' descriptions and titles.
- **Optical WDM Networks**: The main aim is to provide ML solutions to approach the planning and management of network problems, specially in the context of Optical WDM Networks. Advancements:
    1. Proposal of a data-driven framework for the introduction of ML in the computer Network ecosystem in general and in the optical WDM networks field in particular.
    2. Enhancements to the acquisition, generation and labeling of data collections within computer networks assisted by well-known and open source planning solutions.
    3. Approximation of network algorithms and protocols in the context of Optical Networks using ML algorithms that learn from optimally resolved examples.

1.4 Thesis Structure and Contributions

This thesis focuses on the application of Artificial Intelligence and Machine Learning technologies to two interconnected fields where problems are usually beyond traditional ML classification/regression alternatives. The results throughout this work are strongly supported by academic publications in different journals and conferences, which have achieved recognition and prizes in some occasions. The thesis document is structured in seven different chapters organized thematically.

Chapter 2 performs an extensive revision of the State of the Art regarding Android malware detection, multi-scanner analysis of AV engines and ML applications in computer networks. Additionally, Chapter 3 provides an extensive summary of the main methods, tools and technologies mainly used in most of the solutions proposed along the thesis.

In the following chapter, we explore the detection of Android malware using *meta-data*, that is, every piece of information available regarding an application that is not directly contained within its code. In Chapter 4 we define meta-data in the context of Android applications and propose a malware classifier based on quantitative meta-data with labels from multi-scanner tools. This chapter's results are supported by a journal article published in Hindawi *Security and Communication Networks* and a publication in the *Conference on Communications and Network Security* (Florence, 2015).

Afterwards, Chapter 5 uses AI technologies to homogenize, analyze and classify malware detection signatures from different AV engines obtained from multi-scanner tools. Specifically, malware detections from 80 thousand *suspicious* samples are analyzed and used to train an Structural Equation Model (SEM) system that estimates the risk on a sample to be malware based on the AV engines involved in its detection.The work of this chapter has been presented in prestigious conferences like the ACM *Conference on Computer and Communications Security* (Vienna, 2016) and further extended at the *European Intelligence and Security Informatics Conference* (Karlskrona, 2018).

Chapter 6 continues the analysis of application detection signatures from a malware family perspective and uses the resulting conclusions to train an *adware-harmful* classifier to help differentiating malware families. The signature normalization approach proposed, *SignatureMiner* was presented at the *Conference on Communications and Network Security* in Beijing in 2018 where it was awarded *Best Poster Award*. In addition, many contents of this chapter belong to a paper that has been accepted to Elsevier *Future Generation Computer Systems* journal.

Then, in Chapter 7 we revisit meta-information of Android applications to attempt the identification of groups of potentially repackaged applications using qualitative fields, such as application title or description. This chapter is based on a paper published in Elsevier *Future Generation Computer Systems*.

In Chapter 8, we propose a workflow to apply ML in computer networks and focus on two main stages: the generation of labeled networking datasets for ML applications and the solution of non-straightforward computer network problems

through ML. Concisely, we focus on modeling the *Routing and Wavelength Assignment* problem as a classification problem, where each of the classes is a complete network *routing and wavelength configuration*. This chapter is based on a publication in the *European Conference on Optical Communications* in Rome in 2018 and a paper under revision.

Finally, Chapter 9 summarizes the main relevant conclusions and aspects extracted from this work and presents the academic contributions during this period in detail. Additionally, this chapter outlines the main directions regarding future work and continuation possibilities of the thesis. For full publication details, the reader may refer to Section 9.3.

Chapter 2

State of The Art

Throughout this chapter, we review and survey the literature for the different fields and disciplines involved in this thesis, presenting the most relevant works. In Section 2.1 Android Security solutions are studied, specially focusing on AI-powered and *repackaging* solutions while Section 2.2 revises the literature of AV engine analysis. Afterwards, an extensive review of the application of ML and AI to networking is performed in Section 2.3. The chapter concludes in Section 2.4 with a brief recap to enumerate the gaps in the literature that have motivated the different contributions of this thesis.

2.1 Security in the Android Ecosystem

The Android ecosystem is so extensive and popular that the amount of threats within is also limitless. Such popularity exists mainly due to Android's extensive community and device penetration, which could be deeply compromised with the advent of malware and in the absence of proper security solutions. Consequently,

both academia and industry have conducted a massive research effort on malware detection, mitigation and prevention.

This section reviews prior works on Android application security focusing on ML-based and repackaging detection systems. At the end of the section, Table 2.1 summarizes the basic details of the most prominent solutions, including performance and dataset sizes.

2.1.1 Android Malware Detection

At present, malware detection is a very complex process that investigates applications to discover undesired behaviors and take the required actions for their removal. Malware fighting methods range from legacy techniques such as static analysis or heuristics to recent approaches, namely dynamic analysis, *sandboxing* or *anomaly detection*. Idika et al [4] explore and analyze many of these fields on their survey. More recently, the authors pf [5] have reviewed malware techniques from a renovated perspective. Also, the authors of [6] extensively summarize malware types, their implementation details and even propose some defensive techniques against them.

Lately, traditional approaches have been combined with new technologies to further enhance the detection of malware, both in general and, more specifically inside the Android Ecosystem. For instance, Machine Learning methods have been proposed [7] and extensively reviewed [8] for cybersecurity applications, like malware detection. In both reviews, different algorithms and their implementation details are commented and discussed.

Signature-based Detection

Signature-based detection is the earliest and most established detection technique, regularly used by AV engines. Its main objective is finding a unique stream of bytes from the code of a malware sample that unequivocally identifies such malware [9]. The previous work summarizes static application analysis timeline and its importance within AV engines.

Furthermore, the authors summarize malware classes and basic signature detection methods along with the most relevant issues, which involve novel malware techniques such as *polymorphism* or *metamorphism* capable of avoiding detection easily. Indeed, many limitations of signature-based analysis are reviewed in [10], including detection evasion techniques such as *obfuscation* and *binary transformation*.

Dynamic Analysis

Dynamic analysis, also known as *behavior detection*, consists on the execution-driven analysis of suspicious samples. Instead of analyzing static code, which can be easily hindered, dynamic analysis proposes to execute samples within a controlled environment to observe malicious actions at runtime. Dynamic and static analysis are

perfectly compatible and some authors, like the ones of [11], combine static and behavior based techniques with ML classifiers to improve malware detection.

Sometimes, behavior analysis is referred to as *sandboxing*, since applications are executed inside a virtual machine that isolates samples and their actions for safety purposes. Such a virtual machine is called *sandbox* in reference to the safe playground for children. The authors of [12] present the *CWSandbox*, which provides a testing environment to safely run suspicious applications and generate an extensive report of behaviors, loaded libraries, modified files, memory locations, etcetera; to help malware analysts with the dynamic analysis of applications.

In [13], the authors propose a distributed firewall application that integrates the *Cuckoo* sandbox [14] to enhance its functioning with an in-house system for the dynamic analysis of applications and URLs. Recently, VirusTotal released *Droidy* [15], a sandbox system capable of extracting information from malware samples including Network and SMS activities, Java reflection calls, file system interaction, etcetera.

In spite of its advantages, Petsas et al [16] review anti-analysis techniques that can be used to avoid dynamic analysis and even detect the presence of a hyper-visor (and thus that the application is running on a virtual machine). In addition, they test their methods against state-of-the-art anti-malware solutions, finding many of them vulnerable to their attacks. Apart from proposing some countermeasures, the authors raise concerns regarding the effectiveness of current analysis techniques.

Finally, the authors of [17] perform an extensive experiment that compares static, dynamic and hybrid detection techniques while using *Application Programming Interface* (API) calls and opcode sequences, showing that obfuscation complicates static detection of opcodes but not so much of API calls. Moreover, the authors demonstrate that no hybrid approach is able to dispute the performance of a fully dynamic approach.

ML-powered Malware Detection

Malware detection typically poses a classification problem that might be addressed with the help of ML classification techniques. The work from Baskaran et al [18] offers a detailed survey on works applying ML in the cybersecurity realm.

For instance, the authors of [19] propose a malware detection framework that relies on "a multi-stage combination (cascade) of different versions of the perceptron algorithm" aimed at reducing false positives. The classification task relies on 308 binary features to classify over clean and malware applications, the latter coming from the Virus Heaven collection [20]. Results show a very good overall detection score with a very low number of false positives, down to 0.1%.

Similarly, in [21], the authors use permissions and control flow graphs along with a one-class *Support Vector Machine* (SVM) to differentiate malware from goodware applications. The authors in [22] explore Bayesian learning techniques over a set of features extracted through static analysis: permissions and "code properties indicative of potential malicious payload". Malware applications are obtained

from a labeled dataset containing up to 49 known malware families and results seem promising, with an *Area under the Curve* (AuC) up to 0.977.

Elish et al [23] propose a single-feature classification system, the *TriggerMetric*. This feature is computed by modeling the relations between users' behaviors and application operations, specially regarding critical system functions. This metric is extracted through a complex static process of dependence analysis following *data-flow analysis*.

Androdialysis [24] explores the intents of each application as features for the classification task inspired their "semantic richness" in detailing operations that could be indicators of malware. In addition, the authors argue that the combination of intents with other well-known features, such as application permissions, can yield even better results. At the end, intents are compared and combined with permissions to demonstrate that they do not overlap and that their joint analysis can improve malware detection.

Actually, API calls have been extensively chosen as ML input features, probably due to their promising results in many works. In [25], the authors leverage a hybrid Wrapper-Filter SVM which tries to classify malware using API calls and compute relevant statistics from applications. In a similar approach, the authors of [26] use API calls and permissions to train SVMs and *Decision Trees* (DT). Yerima et al [27] try different ensemble methods, such as *Random Forests* (RF) over API calls and sorted code chunks to detect Android malware. Fereidooni et al [28] propose an ML system for malware detection based on API calls, intents and suspicious actions.

Android Malware Detection Systems

Android malware detection features have been specifically studied in works like [29], where the monitorization of 30 goodware and 5 malware apps is proposed to extract and analyze different features for malware detection, namely, CPU usage, network transmission data, process IDs and names and memory information including virtual memory. For detection, the authors propose a simple ML classifier based on activity traces. Mas'ud et al [30] also consider Android system calls inside their detection vectors that they have fed to different ML classifiers in an attempt to validate the selection of malware indicators rather than actually targeting classification accuracy. Indeed, their results show good performance using API calls.

ANDRUBIS [31] is an automated tool for the analysis of suspicious samples based on the analysis sandbox *Anubis*. In this work, a million Android applications have been collected and inspected to highlight differences between *goodware* ("good" software) and *malware* (malicious software). The ANDRUBIS platform is capable of analyzing applications coming from different sources, like crawling markets or user uploads, at a very impressive rate of 3,500 samples a day. Later, in [32], the authors present the results and gained insights of making ANDRUBIS open to the public, providing an extensive analysis on malware trends and their evolution during the collection of a nearly one million sample collection.

Droidchain authors [33] propose a new model to examine Android applications following a combination of static analysis to create behavior chains that rely on API

call graphs describing undesired behaviors. To detect malware, the authors reverse-engineer different malicious applications to build their behavior chain and look for them in new samples afterwards. The authors distinguish four types of malicious behavior chains for malware detection, namely: "Privacy leakage", "SMS financial charges", "Malware installation", and "Privilege escalation".

Moreover, *Drebin* [34] is a system that gathers features from application code and manifest (permissions, API calls, etc) and uses Support Vector Machines to detect and identify different malware families. The resulting Android application gives users an estimation of the danger of any application being malware prior to installation along with a description of the main reasons behind such value.

Other authors have inspected different features to detect malware different than application code. This is the case of [35] where the authors perform sentiment analysis over applications' comments extracted from Google Play in order to obtain indicators of potentially harmful applications. This approach is one of the few in the literature that considers application meta-data, which will be used in this thesis as input features for ML classification.

Permission-based Detection of Android Malware

In general, permissions have been extensively studied for malware detection within the Android Ecosystem. The Android permission system defines a series of protected resources, including API calls for hardware (led light, sending SMS) or software (booting options, contact list) manipulation. These permissions have to be declared at installation time for any application to use them. These properties have motivated researchers to inspect permissions under the assumption that the concurrence or abuse of certain subsets of permissions may be critical in identifying malware.

However, works like *PUMA* [36] have indicated that the difference in permissions between goodware and malware is not directly obvious. Despite, the authors are able to train a random forest classifier that obtains 0.92 AuC on a small dataset of over two hundred applications.

The *Droid Permission Miner* system [37] extracts features, mainly permissions, for malware detection using the manifests of a set of applications from the Contagio Dump blog. Furthermore, the authors perform feature selection over permissions differentiated by their malware status and conclude that high accuracy malware detection is possible by using roughly 15 permissions.

In a different approach, Hein et al [38] collect Android application permissions and train a *Self Organizing Map* (SOM) to infer permission-based models informative regarding the risks the users suffer from installing applications. Likewise, Barrera et al [39] follow a similar methodology involving SOMs that recommends splitting general and very often used permissions like *INTERNET* into smaller categories and collapsing some other more infrequent and self-defined permissions into a generic category.

Alternatively in [40], the authors present a pattern mining algorithm capable of contrasting requested permissions versus used permissions. In fact, they show how many permissions declared in the Android Manifest are not used later on the application. Following a similar approach, the authors of [41] extract "real permissions" used by the applications in contrast to the permissions declared in the app manifest.

Finally, the authors of [42] systematically review and discuss the Android permission model, including what makes it a good reference for the internals of Android Operating System. Moreover, the paper presents a detailed collection of attacks against the Android permission model. Similarly, Felt et al [43] review the Android permission model and analyze potential patterns of developer error. Besides, they present the *Stowaway* tool, capable of detecting overprivileged applications.

2.1.2 Repackaging Detection

Repackaging is a very extended malware development technique that relies on using the code from legitimate applications inside any market to introduce malicious code, re-compile and offer them in any application market as if they were the original copies. Since the bytecode produced by the *Android Dalvik Virtual Machine* can be reverted back to *smali*, which is close to Java, any Android application is vulnerable to this technique. Indeed, any Android application can be *decompiled*, modified and re-uploaded to the market to lure users into downloading clones rather than the legitimate version at a very low cost. A survey on repackaging detection methods can be found at [44]. Interestingly, only Li et al [45] track repackaging origins in an extensive analysis of *piggybacked* apps and their basic characteristics.

Application Code Inspection

One of the most popular approaches for repackaging detection is the inspection of application code seeking for patterns and suspicious data structures. For this, different parts of the source code of any two applications are compared to look for hints on whether they have been cloned. *Andarwin* [46] systematically analyzes application code by representing each application as a *Program Dependent Graph* (PDG) that maps methods and relations inside applications. To provide potentially cloned applications, the system clusters them together using min-Hashing of PDGs. The authors remark that this is a very scalable approach, mainly due to the automatic similarity approximation provided by clustering.

Guan et al [47] propose targeting the input/output symbolic representations of each app to detect clones based on semantic similarities. In their paper, the authors present their prototype, *RepDetector* and evaluate it with repackaged, obfuscated and Google Play apps. Furthermore, the authors of [48] search for anomalies introduced in the data section of application code by *Smali* decompilers. Actually, the authors propose a novel technique, the *String Offset Order* (SOO) that looks for these anomalies and any other element falling out of the standard guidelines of the Android code data section.

Following the code line, execution patterns of applications have been explored too. In [49], the authors propose *MIGDrodid*, a system based on the comparison of *Method Invocation Graphs* computed from the smali code of each app to detect the injection of potentially malicious code inside applications. This way, MIGDroid is capable of assigning a *threat score* to each code part relative to the sensitivity of each of the APIs invoked. To evaluate the system, the authors apply the MIGDroid methodology to $1,260$ Android Malware samples, achieving an average "detection effectiveness" of 95.95% and a false positive rate of 8.9%.

Similarly, the authors of [50] compute Control Flow Graphs of each application's code following a similar approach to the one in MIGDroid. Then, using such graphs, they apply a centroid-based clustering algorithm that measures app similarity. Such approach is validated and demonstrated through cross-market analysis of applications from five different markets. Results show a *False Negative Rate* (FNR) of down to 0.4% and a *False Positive Rate* (FPR) of 0.38%.

Finally, at the interactive level, User Interface (UI) code and visualizations have been inspected and compared. Chen et al [51] develop a large-scale system to detect resembling structures and intersections in UI code. This methodology, called *MassVet*, relies on a very efficient differential analysis of any application against the entire market to find the aforementioned similarities. Their methodology is demonstrated over a collection of 1.2 million applications discovering $127,429$ malware applications.

The authors in [52] propose a novel approach based on the inspection and clustering of the UI birthmarks of each application that can be extracted from the view hierarchy in XML. To do this, the authors leverage *Local Sensitive Hashing* (LSH) over these birthmarks containing runtime information to locate near neighbors within them. Then, similar pairs are compared using Hungarian similarity scores and clustered into groups of clone sets. The authors highlight that the effectiveness of this approach is rooted on its focus on UI rather than any other application code part that may be easily obfuscated.

Finally, *RepDroid* [53] automatically detects application duplicates by extracting their *Layout Group Graph* (LGG) from runtime UI traces, which are also resistant to classical obfuscation attacks and enables comparison through similarity scores. On their validation, the authors report an FPR of 0.08%.

Meta-data for Repackaging Detection

Recently, meta-information has been inspected to look for application similitudes too. The authors of [54] investigate meta-information from application markets to provide insights and patterns that are very focused on permissions. They also propose a detection system based on similarity clustering and anomaly detection over semantic patterns in different meta-data features, including permissions, user ratings or downloads among others.

The authors of [55] develop a white list system to check incoming APKs from a database of legitimate applications and compare application icons when no white

list match is found. The solution consists on the detection of repackaging applications independently of the specific application store: the application targets elements "an attacker is reluctant to significantly alter", such as the application name and icon. Their validation results report a detection rate of up to 91% within a collection of almost 400 applications.

In [56], the authors propose various pairwise similarity metrics of different meta-information fields, such as application name, description or icon to unveil application copies. In spite of proposing an indexing system, the authors do not discuss on specific details regarding the complexity and time consumption of the approach.

Market-scale Solutions

In repackaging, the scalability of solutions is a great concern, since most repackaged applications have original victims that are not necessarily known in advance. Thus, it is important to look for scalable solutions that can dive into huge application markets to quickly find not only cloned applications but also their victims to support their findings.

In this light, many authors have addressed the problem by proposing market-scale solutions, like *Andradar* [57], which monitors several Android markets in real-time tracking application changes and removals using "lightweight identifiers", like the package name, developer's certificate fingerprint and method signatures. This solution has enabled the tracking of different applications through time, that facilitates an in-depth analysis over 8 alternative markets involving a total of 318,515 applications and the study the publishing patterns of malware developers on 16 different markets.

Indeed, many of the aforementioned solutions are scalable, such as that of Chen et al [51] regarding differential analysis or the one in [55], where the authors propose a white list filter and efficient comparison techniques based on hashing to scale out. Besides, the authors of [58] perform large-scale inspection of 200,000 Android apps in order to determine which ones have been cloned. For that, they focus on application intents and exit points leveraging labeled collections from other studies, such as [59, 50, 45], in supervised classification.

Repackaging Countermeasures and Consequences

The repackaging detection literature does not only contain detection proposals, but also other interesting works, including countermeasures and impact reports. To fight against repackaging, some approaches propose active countermeasures to avoid repackaging from the beginning of the development process: The authors of [60] look for inspiration on Copyright Watermarking to embed applications with authorship verification as an anti-plagiarism solution. Their specific proposal consists on introducing "non-stealthy" watermarking within the application protected by means of *Self-Decrypting Code* (SDC). Similarly, Zhou et al [61] propose a system to reproduce and watermark Android applications using dynamic graph mechanisms.

TABLE 2.1: Summary of works regarding malware and repackaging detection in the literature.

Paper	Methods	Detection Features	Target	Data	Acc.	FPR(%)
Devesa et al [11]	Various classifiers	Behavior logs	Automated dynamic analysis system	1.5K	0.96	1.3
Willems et al [12]	Dynamic analysis, malware techniques	System calls	Sandbox system to extract behavior reports	-	-	-
Vasilescu et al [13]	Firewall tables, dynamic analysis	Incoming traffic	Distributed firewall with integrated sandbox for in-house analysis	-	-	-
Damodaran et al[17]	Hidden Markov Models	API calls and opcode sequences	Comparison of Static and dynamic malware analysis	785	-	-
Gavrilut et al [19]	Perceptron multi-stage combinations	Distinct binary features	Malware detection focused on reducing false positives	300K	0.88	0.1
Sahs et al [21]	One-class SVM	Permissions and control flow Graphs	Malware detection	-	0.25	-
Yerima et al [22]	Bayesian Learning	Permissions and code properties	Zero-day malware detection	1K	0.93	-
Elish et al [23]	Data Dependence Graphs, threshold-based classification	TriggerMetric (user relations)	Data-flow application analysis for malware detection	4K	0.979	2
AndroDialysis [24]	Bayesian Network	Application intents and permissions	Explore intents as features for malware detection	7K	0.95	-
Huda et al [25]	Hybrid Wrapper-Filter SVM	API calls	Malware detection targeting malicious activities	67K	0.96	-
Peiravian et al [26]	SVM and DT	API calls and permissions	Malware detection	3K	0.95	-
Yerima et al [27]	Ensemble methods	API calls and sorted code chunks	Malware detection	6.8K	0.97	-
Fereidooni et al [28]	XGBoost and feature selection	API calls, intents and suspicious actions	Malware detection and family-based classification	16K	0.92	-
Ham et al [29]	Various Feature Selection and Classification	CPU usage, network transmission data process ID and names and memory info	Malware detection with runtime monitoring parameters	15K	0.99	-
Mas'ud et al [29]	Various Feature Selection and Classification	Android API calls	ML-based malware detection	-	0.83	23
Andrubis [31]	Anubis for app collection	Java and Native Code	Implementation of high throughput detection system and feature analysis	900K	-	-
DroidChain [33]	Using API call sequences for behavior chains	Chain models (API calls)	Malware detection based on behavior chains	1.2K	0.8	-
Drebin [34]	Static analysis and linear SVM	Application Code and Manifest	On-device Malware detection with explained decisions	120K	0.94	-
Nannen et al [35]	Sentiment Analysis	Google Play comments	Detection of malware application	-	-	-
PUMA [36]	Random Forest	Permissions	Analysis of permissions for malware detection	239	0.92	-
Aswini et al [37]	Feature selection and diverse classifiers	Permissions	Permission-based malware detection	436	0.82	22
Hein et al [38]	Self Organizing Map	Permissions	Inform users on application's risks	300	-	-
Barrera et al [38]	Self Organizing Map	Permissions	Show permission hierarchies and usage	1.1K	-	-
Moonsamy et al[40]	Pattern mining	Permissions	Differentiate used vs required permissions	2.5K	-	-
Andarwin [46]	Program Dependent Graph and min-hashing	Application code	Repackaging detection	265K	0.96	3.7
RepDetector [47]	Similarity detection	Application input/output symbolic representations	Repackaging detection	1K	0.99	-
Gonzalez et al [48]	Anomaly detection	Data section of DEX code	Repackaging detection	85K	0.98	2.9
MIGDroid [49]	Method Invocation Graphs	Smali code	Assign Threat score to APIs involved	1.2K	0.95	8.9
Chen et al [50]	Control Flow Graphs and clustering	Application Code	Detect application repackages in different markets	150K	-	0.38
MassVet [51]	Differential analysis with market	UI Structures	Compare applications with markets at scale	1.2M	-	-
Soh et al [52]	LSH clustering	UI birthmarks	Detect repackages from their UIs	521	-	0.08
RepDroid [53]	Layout Group Graph	Runtime UI traces	Repackaging detection	125	-	0.08
Teufl et al [54]	Clustering and anomaly detection	Application market metadata	Provide insights on meta-data and repackaging detection system	-	-	-
Gurulian et al [55]	Hashing methods	Application name and icon	Whitelist system and icon-based comparison for repackaging detection	400	0.91	-
Kywe et al [56]	Pairwise similarities	Application name, description and icon	Detect repackaging by pairwise similarity metrics of meta-data fields	30K	-	9.2
Andradar [57]		Package name, developer certificate items	Monitoring of Android markets with realtime application tracking	320K	-	-
DecisionDroid [58]	Various classifiers	App intents and exit points	Repackaging detection	200K	0.98	-

In [62], the authors suggest provisioning applications with *self-protection* capabilities. Such protection capabilities include the injection of "randomized detection code" into the bytecode of any Android application that is split across the original code and prevents correct application functioning when it has been repackaged.

Recently, the authors of [63] proposed the introduction of malware-style *logic bombs* in legitimate applications to prevent repackaging. Such logic bombs would be hindered inside legitimate applications and triggered upon repackaging detection to corrupt application code.

In a different approach, some authors have used repackaging to improve application security, as in [64], which proposes the introduction of a user-level sandbox into applications automatically through repackaging. Such a sandbox would be in charge of enhanced security mechanisms that would include a fine-grained permission management system, behavior monitoring or network interception.

Similarly, the authors of [65] repackage applications with a privacy reporter component capable of auditing the use of personal user data. For this purpose, such component would monitor data access attempts along with permissions used to be sent to a third party application that works like an Intrusion Protection System (IPS).

Finally, the consequences of being a repackaging victim have been studied for many areas, such as banking [66], messaging applications [67] or even smarthome devices [68]. In most cases, the authors try to identify basic vulnerabilities, propose countermeasures and identify the risks and impact of such attacks.

2.2 Antivirus Analysis and their Detections

Antivirus (AVs) applications are in charge of the detection and mitigation of malware in computers, mobile phones or any other device. Historically, AVs have relayed in static signature-based analysis to detect malware, but due to the advancements of the malware industry, they have required to include new techniques, such as dynamic analysis or sandboxing.

Usually, AV engines have been the main barrier against malware at user level, protecting all kind of devices. Nowadays, AV engines alone are not so useful any longer, albeit they can offer still a first defense layer against well-known attacks. In addition, the maintenance of efficient teams of analysts to deal with new incoming threats, specially zero-day attacks, makes AVs still useful to some extent.

In this light, AV engines have been persistently scrutinized to unveil flaws and understand their detection rules and policies. Recall the work from Moser et al [10] which performs a comprehensive review on the limits of static analysis. Other authors, like [17, 16] also show such limitations mainly affecting AV engines.

AV Resilience to Anti-detection Techniques

For instance, Rastogi et al [69] perform a systematic evaluation of Android anti-malware products by studying their detection abilities in the presence of anti-detection

techniques, such as obfuscation. The authors study the evolution of many AV products over a year period, showing that engines make an effort to improve the detection abilities of its signatures. Nonetheless, the authors show most AV engines fail against common evasion techniques even when signatures are strengthened. In the end, the authors explore potential improvements for current solutions.

Similarly, in [70], the authors review common evasion techniques and perform an experiment to evade detection in VirusTotal using "two of the simplest tests". In [71], the authors discuss the limitations of malware detection using sandboxes and demonstrate their theses by fingerprinting ten different sandbox solutions. These solutions can be then evaded using *Divide-and-conquer* type of attacks. They even show that *Google Bouncer*, the first-response AV engine introduced by the company in Google Play, can be avoided by such techniques.

In the mobile world, the authors of [72] perform a review of different solutions for Android malware detection, both in-device and in-cloud. Besides, they revise the key points in designing AV engines to adapt them to mobile AV versions and focus on detection from different perspectives.

Malware Detection Systems

Malware detection systems can be grouped into two different paradigms: (i) online services, sometimes web-based, that produce fast responses upon suspicious application queries for direct usage, such as pre-installation checks or user security, and (ii) powerful offline services that analyze large collections of applications to enforce security at market scale.

On the one hand, there has been extensive research regarding lightweight applications, typically in-device, that can inform users on the risks they are taking with the installation of any application. For instance, *Drebin* [34], which has been presented before, develops an Android application that assesses new applications' risk along with a verbose description of the key observations for that estimation. Furthermore, the authors of [73] propose the *Secloud* system that facilitates the application of powerful intrusion detection solutions and other methods by emulating each device environment within a resource-friendly cloud environment rather than the same device.

Andromaly [74] is conceived as an in-device malware detection system, similar to traditional Antivirus engines, that monitors the device and applies machine learning classifiers to separate goodware from malware applications. In the paper, the authors perform feature selection as well and try different ML algorithms to produce more accurate results. Applications can be analyzed either locally or in the cloud and users are encouraged to mark applications they consider malicious.

On the other hand, mass detection offline systems have been very popular too, like the *DroidAPIMiner* approach [75] that uses semantic information extracted from bytecode in different lightweight classifiers to obtain fast and accurate application prediction. *RiskRanker* [76] is a scalable solution to analyze applications at market scale that even supports the detection of the so-called *zero-day* attacks.

In addition, *Gravity* [77] is a massively parallel AV engine build over the source code of *ClamAV* that relies on GPUs to provide very fast and efficient threat detection. Actually, the authors claim Gravity is able to process applications at 20Gbps, reaching peaks of 110Gbps and a 100x fold with respect to regular ClamAV. In a different approach, *LeakMiner* [78] identifies and prevents data leakage by decompiling applications into bytecode, identifying points of data usage and following the execution path through activity graphs to identify potential leak points. As a reference, the reader can consider the survey in [79], which extensively reviews other relevant detection systems and techniques proposed in the literature for the Android ecosystem.

Finally, it is worth considering the gathering and collection of threat intelligence, including malware behaviors, AV-like signatures and more relevant data. One of the most promising solutions is *CrowDroid* [80], a framework that centralizes the collection and analysis of application traces along with the detection of anomalies and uncommon behaviors. Indeed, the authors propose their platform as a "crowdsourcing system" that relies on users sharing their devices' information to collect and detect malware. Such detection is based on a two-group clustering algorithm over "application system calls" (API calls).

Moreover, in [81], its authors propose a system to collect and analyze Android malware aiming at the creation of quality detection signatures that collects applications from different sources in the Internet and produces signatures based on the hashing of sequences of API calls. The *DeepSign* approach [82] relies on a Deep Learning system to generate malware signatures from binary application features using an autoencoder network architecture. Such system executes applications inside a Sandbox and feeds the resulting logs to the network which output is fixed to 30 values. Finally, the solution is validated by training an SVM classifier that achieves testing accuracy values over 95%.

AV Performance and Alternatives

AV solutions are conceived as regular system programs loaded with privileges and capabilities to perform their protection tasks including extensive system control and risky access permissions. As any other application, AV engines have been shown to contain vulnerabilities like any other software and, which must be immediately addressed to prevent larger damages to the host system upon detection [83].

In spite of their benefits, AV performance and the impact in their host devices has also been studied in depth. In truth, many AV solutions have been very criticized for different reasons, such as unjustifiably large performance overheads or even taking actions that could be considered malicious.

Al-Saleh et al [84] analyze the intrusiveness of two AV solutions (Symantec and Sophos) using a Windows event logger in an attempt to evaluate the impact on system performance. Their results show that the presence of an AV engine noticeably impacts on regular program execution, not only in terms of CPU usage, but also in IO operations and paging issues. Before, in [85] AV engine overhead was formally characterized and used to evaluate engines from an *on-access perspective*, that

is, when the engine is monitoring files and system settings in the background.

In this light, cloud computing has emerged as a key trend for malware detection systems due to its computing potential and versatility. In-cloud malware solutions can be conceived as lightweight approaches that get installed in host devices just to collect monitoring information and other important systems and send it to a dedicated cloud environment provisioned with large computing capabilities to check samples and potential threats and return devices responses and action policies.

Indeed, cloud computing can be used to collect and identify malware signatures, like the authors of [86] do by proposing the *Automatic Malware Discovery System* (AMDS). AMDS installs some lightweight cloud signatures in-devices that are tested before sending samples to cloud environments. Also, cloud computing has been proposed to collect and identify malware signatures. In [87], the authors propose *uCLAVS*, a cloud environment using many detection schemes and AV engines. Furthermore, the system proposes an ontology that rules the interrelations between web and security components such as firewalls or vulnerability scanners.

In [88], the authors deepen on the concept of "Antivirus as a Network Service" by sending new files generated by system applications to a cloud server. Besides, the authors remark the importance of considering the detection outcome of more than one AV engine as enabled by cloud computing. As a proof of concept, they propose *CloudAV*, composed by 10 AV engines and two behavioral detection engines, showing 35% performance increase with respect to single AV engines. Using a dataset of more than 7,000 samples, the authors show a 98% performance.

The authors in [89] perform an extensive analysis of malware detection solutions and propose a framework involving many alternatives that can be used as a modular, scalable and private cloud system. Such system would be open source and extended with innovative tools applied to malware detection such as data mining. At this point, it is worth recalling the *Crowdroid* system [80] presented above. Such work is proposed as a centralization alternative to gather crowdsourced detection signatures from different devices.

Multi-scanner Detection Tools

Lately, several online *multi-scanner* services have appeared to integrate malware detections coming from different engines with the goal of collaboration and threat sharing. These tools are typically web services to upload potential malware applications, URLs, files or similar sources in order to get detection reports from many popular AV engines. These detection reports do not only include whether each engine considers malware a sample or not, but also details regarding the threat, versions or even target operative system. Some of the most popular multi-scanner solutions are VirusTotal [90], Meta-Scan [91] or Jotti [92].

These tools enable analysts or any other interested actor to have more than one "expert" opinion for each suspicious sample. Actually, many authors [93, 94] have demonstrated the advantages of considering more than one AV engine on malware decisions. Using multi-scanner detections from VirusTotal, the authors in [95, 94]

compare different engines and model each engine's confidence through a hyper-exponential curve using as reference a previously labeled collection of applications. Furthermore, the authors observe that full detection coverage of their malware collection is achieved by 25% of all possible AV pairs and almost 50% of the possible triplets.

In [93], the authors perform an empirical analysis of different AV engines from ViruстTotal using malware samples obtained from a honeypot and conclude that using more AV engines improves overall detection, showing that engines alone are limited as well. The authors also point to regression cases, where AV engines stop detecting a certain malware sample. Following the same line, the authors of [96] analyze dates from VirusTotal labels in order to track their evolution in time including changes and withdrawals. To do this, they consider samples obtained from a custom honeypot network.

Besides, the authors in [97] use VirusTotal labels and human experts to detect malware using temporally consistent labels. The authors analyze a million application dataset and involve "humans in the loop" to further improve the performance of detection systems. In fact, the authors claim to outperform regular AVs in terms of detection and FPR.

AV-Meter [98] analyzes nearly 12K labeled applications and discusses AV engine behaviors and decisions. In their analysis over a manually labeled dataset, the authors include different custom evaluation metrics. Moreover, in [83] 30 of the top AV solutions to detect and prevent malware are compared. In their analysis, the authors reverse-engineer some AV engines to try to understand their detection logic, finding in the process different detection vulnerabilities and flaws.

Quarta et al [99] leverage VirusTotal detections to analyze AV engines following a black-box approach: only input samples and different detection results from each engine are considered. For this, the authors develop a set of tests within the CRAVE framework to test several engines and their anti-evasion capabilities in VirusTotal. Together with their findings, the authors make a powerful statement regarding the importance of analyzing and testing AV engines to clarify the "obscurity" they have.

In this light, multi-scanner tools are limited and have still several shortcomings. New threats and attacks can be built leveraging these tools to improve their anti-evasion capabilities, like the case of [100], where its authors discover malware developers using multi-scanner tools directly integrating multi-scanners into their malware development pipelines to avoid detection. The authors further propose a methodological approach to detect such malware developers in VirusTotal submissions using ML classification over submission details and meta-data.

Actually, Willems [101] discusses good and bad features of multi-scanner tools, including trends and observations from the malware-development community. Among other observations, the author discusses that beginner malware developers (known in the security jargon as *Script Kiddies*) still use public multi-scanner tools for malware development, probably without knowing that their samples are being shared with the community, whereas more professional developers are aware of this fact and have developed their own multi-scanner tools.

Finally, two main problems with AV engines are worth noting, since they have been further confirmed by multi-scanner tool detections. Those are the *lack of consensus* AVs show in detecting samples as malware and the *naming inconsistencies* in the categorization of threats.

In [102], Hurier et al perform an extensive review of AV engine consensus in terms of detection and class labels, showing the deficiencies in the process. In general, they find very generic malware labels along with the observation that AV engines involved in more detections tend to perform better. Besides, the authors propose different metrics and scoring systems to quantify the lack of consensus aiming at the assessment of ground truth datasets.

In addition, using multi-scanner tools, Maggi et al. [103] show how different engines name malware families differently. In their work, they extensively review such inconsistencies, that they observe very often whenever more than one engine is involved in detection. Their specific approach relies on using a graph to model the relations between different vendors' naming schemes.

Therefore, the rest of this section will develop on malware categorization and review the most popular and extended solutions to these problems, which are key to unlock the potential benefits and improvements multi-scanner tools can give to the ecosystem.

Malware Categorization

Not all malware is equally malicious, neither does it pursue the same goals nor follow the same methods. The different types any malware sample can belong to are usually called *malware classes or families*. Different families indicate a distinct type of behavior or target victim. Such schemes can be useful, specially to detect and prevent similar threats with the same type of actions.

Through the literature many authors have attempted to categorize malware in different families. The most straightforward strategy is that of code analysis. For instance, *Dendroid* [104] proposes a novel text mining approach to extract malware families from application code. For that, the authors propose the calculation of code structures similarity to accurately assign the appropriate family to any incoming sample. In addition, the authors perform evolutionary analysis over family code through dendrograms. It is worth noting the complementary approach of Zheng et al, *DroidAnalytics* [81], which is also capable of collecting and categorizing zero-day malware samples into existing families.

With the proliferation of AV engines and, specially, multi-scanner tools, many authors have leveraged the latter to infer and study different malware families. In [105], the authors discover up to 49 distinct malware families by examining a collection of more than 1,200 applications and systematically studying the internals through time. In [106], the authors consider multi-scanner outputs and majority keyword voting to obtain labeled malware samples. Then, their approach relies on supervised learning and clustering over application behavior to identify well-known families. Thanks to this process, the authors are able to create a nearly 25,000 sample dataset containing malware applications categorized in families.

Finally, the authors in [107] use *Droidbox*, a sandbox application, to demonstrate how AV engines are not completely proficient at detecting and identifying certain malware families. In some way, this paper deepens more in the aforementioned *lack of consensus*.

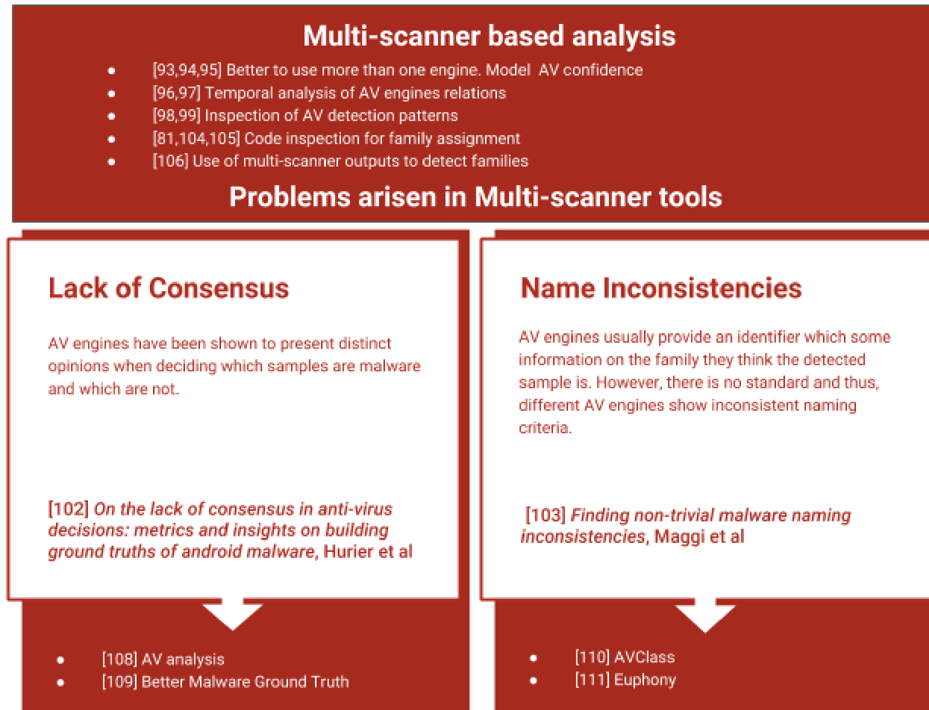


FIGURE 2.1: Multi-scanner tool-based analysis, current problems and solutions

Solutions to Multi-scanner Tool Limitations

Previously, we have reviewed the literature regarding multi-scanner tools and identified some issues, such as their *lack of consensus* as well as their frequent *naming inconsistencies*. Since using many engines is beneficial in malware detection and facilitated by multi-scanners, there is a clear need for solutions to achieve consensus and consistency among engines or, at least, filter out the less useful ones.

Some authors have developed solutions to both problems, mainly oriented to mitigate the potential disagreements among engines. The authors in [108] review a large collection of malware samples from *VirusTotal*, *SGNET* and *ANUBIS* and show how malware was becoming preeminent. Furthermore, they debate the difficulties in determining whether the lack of detection of a malware sample is a failure committed by the AV engine.

In the work of Kantchelian et al [109], the authors propose a novel Generative Bayesian model that takes as input the detection reports of different engines coming from multi-scanner tools and outputs the probability that a detection is malware. In short, this system uses the AV engines that have detected a sample to estimate how likely is such sample to be malware. Hence, determining whether a sample

is malware becomes partially dependent on the predominant detection patterns of many engines.

Regarding family controversies and inconsistencies, solutions aim at the normalization or homogenization of detection labels. *AVClass* [110] is proposed as a tool for the extraction of malware families from multi-scanner detection signatures. To this end, AVClass processes the text of these signatures to obtain common tokens to which it applies frequency mining. After some cleaning and using large signature datasets, AVClass accurately computes the normalization of all the signatures within a malware sample and applies majority voting to determine the most likely class.

Euphony [111] is a more recent approach that follows a similar methodology. In this case, the authors use heuristics to clean the report signatures and create a graph connected according to the different signatures associated to the same samples. Then, to aggregate labels they compute the most notorious sub-graphs and name each cluster after the predominant substring within.

Fig. 2.1 recaps works motivating multi-scanners, analyzing distinct AV engines in the light of them and proposing solutions to the above-described problems of lack of consensus and naming inconsistencies.

2.3 ML Applied to Optical WDM Networks

AI and Big Data have emerged into the networking field as game-changing technologies capable of enhancing network protocols by means of ML-based approximations that are faster and better than common heuristics. Actually, AI tools are really well suited to be applied in conjunction with Software Defined Networking (SDN) technologies, a networking paradigm that seeks making networks programmable and flexible. Both fields combined have opened huge possibilities to the extent of *Knowledge-defined networking* or *Cognitive Networking* [112, 113].

Some interesting tutorials on algorithms, frameworks and applications of AI technologies to networking, including open challenges and specific examples of *Intelligent Networking* may be found in different works, such as [114, 115, 116, 117]. The authors of [117] specifically overview the potential, challenges and future of *Deep Learning* (DL) in computer networks. Recently, Boutaba et al [118] have performed an extensive survey of ML applications to networking along with "a purposeful discussion of the feasibility of the ML techniques for networking", where the authors discuss how ML techniques application to different network environments together with the most prominent challenges.

Furthermore, the authors in [119] summarize the advantages of putting SDN and Big Data technologies together. Concisely, they focus on how Big data technologies can take decisions that would be easily implemented and applied to the network through SDN controls. Furthermore, the authors recap specific applications in traffic engineering, cross-layer design and even defeating security attacks.

ML-powered Network Traffic Classification

Traffic Classification has been one of the most straightforward network problems to be solved through ML. In general, traffic flows can be observed and quantified into a diverse set of features, like bytes sent and received, the amount of created connections, etcetera, which can be directly used to classify flow types.

Therefore, many researchers have attempted direct ML classification, like the authors in [120], who join classifiers with Deep Packet Inspection methods (DPI) within a framework applicable through SDN. The classification is performed according to Quality of Service (QoS) levels and uses as features flow-level information, such as inter-arrival packet times, packet length, protocols or any other relevant parameter. The resulting accuracy is above 90%.

In fact, some works date from before the beginning of AI popularity in networking [121]. There, several approaches and open challenges were indicated. Recently, the authors of [122] propose a fuzzy clustering system over traffic statistics to discover and label different flows in a semi-supervised manner. The reported results show up to 95% classification accuracy.

In a different approach, the authors of [123] propose the *Bag of Flows* (BoF) methodology to identify traffic flows by correlating the ones observed in a network. In a more recent paper, the authors extend the BoF model and propose a novel method for traffic flow class discovery [124]. This method, the Robust statistical Traffic Classification (RTC), targets "zero-day applications", which represent those new flows appearing after the training process. RTC relies on an initial step involving K-means clustering and Random Forests to isolate zero-day flows and extract relevant features that can be afterwards used by the original BoF approach.

Shafiq et al [125] review classification proposals from port-based schemes to DPI and motivates the use of machine learning in order to go beyond. In [126], its authors survey over different traffic classification techniques, including those fueled by ML.

From that point, traffic size prediction has followed. For instance, Poupart et al [127] propose a data mining system to estimate flow sizes motivated by traffic classification works. In their approach, the authors propose using common features from the initial packets of a flow to estimate its size and share it with the rest of network controllers. In [128], their authors advance on traffic prediction by using a *Hidden Markov Model* (HMM) over "easy to collect features", namely, number of flows in the network at a time interval. In an interesting study [129], the impact of IoT devices on network traffic and connections is also evaluated, showing different models for IoT traffic.

ML-powered Routing

Routing is a complex network problem that has been solved through many algorithms, such as *Disjtra* or *distance vector*. In general, classical routing algorithms do not consider real time parameters, such as overall network load. However, using

ML, routing can be made more efficient, by dynamically tracking traffic loads and re-routing the most conflictive demands.

Mao et al [130] review different routing strategies, including network traffic control, deep learning and computing architectures and propose a per-node "routing strategy" based on classifiers that compute the routes to follow for incoming packets at each step. In a similar approach, the authors of [131] propose a routing algorithm based on a Convolutional Neural Network (CNN) trained with past network states. Such solution is specially effective in congested scenarios, where it outperforms traditional routing protocols.

ML-powered Quality of Service Management

Quality of Service (QoS) is a paradigm for policing and traffic management that attempts to differentiate network services according to their use and measures the overall service provided. Lately, QoE (Quality of Experience) has appeared as the user-centric metric that contrasts to the application-centric QoS.

By design, QoS defines a series of methods and parameters that have to be adjusted in order to provide network users with a certain "service level" and for that, ML can be useful, as these parameters usually need estimation from network states and observation. Moreover, QoS and QoE are similar measures obtained at each of the ends in the Service level agreement (service providers and users) that usually are hard to correlate. ML and data analysis can also help in relating both to improve final user satisfaction.

In [132], the authors investigate QoE and QoS along with their potential relations. Concisely, the authors measure the quantitative relation of QoE "opinion scores" and QoS parameters, such as down times or packet loss along with the correlation analysis of network measurements and the aforementioned QoS parameters. At the end, the authors conclude that users showing better QoE are also those with larger online times.

Mushtaq et al [133] point out the need for relating QoS and QoE and provide some ML-based methods to improve the assessment of user experience. In their analysis, the authors consider QoS network parameters, such as packet losses, jitter or delay as well as video characteristics (bit rate, frames, etc) and use ML algorithms over user feedback to find the most relevant factors in common between QoS and QoE.

The authors of [134] propose a QoE estimation system that uses QoS parameters and user emotions collected by video recording of the users facial expressions. The estimation of QoE is performed by testing different ML models. Furthermore, in [135], the authors use decision trees to estimate QoE from limited user feedback aiming to reduce the number of subjective tests to be performed.

Recently, some authors have addressed the prediction of QoS for narrow areas of networking, usually based on data-driven and observational approaches. As an example, the authors of [136] propose a data-driven technique to infer QoS information for Industrial IoT using Kernel Least Mean Square algorithm (KLMS) over an

user-item matrix like in recommender systems. Such matrix contains different QoS features of different web services measured for different users.

For further reference, the authors of [137] perform a survey on QoS-QoE correlation methods based on ML that provide basic definitions and descriptions along with a comprehensive collection of relevant papers.

ML-powered Provisioning and Resource Allocation

Resource allocation and provisioning is another important area inside networking, as the regular operation of any network requires the availability of usable resources for expected and unexpected demands. In fact, network operators are usually obliged to overprovision, that is, reserve much more resources than eventually needed to prevent the collapse of the network due to traffic bursts. Hence, AI could be a useful tool to better adjust the amount of resources dedicated to a network and, in combination with SDN, develop responsive solutions that allocate more resources to networks upon need.

In [138], the authors propose a *Reinforcement Learning* (RL) provisioning system capable of maximizing monetary gains by tuning parameters of *DiffServ* networks. To obtain this gain, the authors leverage the response ability of RL techniques to provision the network dynamically according to each moment's requests rather than static allocation. The main novelty of this work is the use of RL to adjust the pricing policy to the actual network usage to save money for clients and operators.

The authors in [139] develop a DL-based traffic predictor to facilitate resource allocation in the near future. In a first stage, traffic is predicted using a *Deep Neural Network* (DNN) that facilitates the assignment of resources; afterwards, the authors propose a second DNN to perform resource allocation. In their results, the authors show smaller blocking probability values and larger resource occupation rates than using other well-known solutions.

In [140], Hayasi discusses the use of ML in SDN network environments with a twofold objective: (i) detect anomalies and failures and (ii) support the automated management of such network. Then, the author surveys different applications of ML to those objectives in an SDN network taking into account the improvement of the business processes.

This type of solutions has been proposed in optical networks too, specifically on detecting failures and problems within the network. As an example, the authors in [141] propose a system to detect and avoid power excursions in ROADMs using a DNN. Moreover, Gutterman et al [142], propose an NN regressor to predict the amount and sign of power excursions when adding a new wavelength to a *ROADM*.

ML-powered Network Traffic and Congestion Control

Traffic and congestion control is essential for networks in general to adequate the throughput and traffic passing through them. This way, resources can be efficiently

distributed following fair policies that do not treat traffic from any source or destination arbitrarily.

For instance, the authors of [143] propose a maximum likelihood estimator to identify the cause for TCP packages getting lost at origin. Furthermore, in [144] the authors explore the use of supervised learning to determine the specific type of a packet loss in the context of wireless networks. For this end, the authors try to develop different models (i.e. decision trees, Xboost, neural nets) to classify network losses into two categories: link errors and congestion.

In [145] the authors propose a loss classification technique for Optical Burst Switching (OBS) networks that differentiates between contention and congestion packet losses. For that purpose, they propose a new measure called *Number of Bursts Between Failures* (NBBF) that models contention and congestion losses as Gaussian distributions with different parameters. The authors rely on hidden Markov models and expectation maximization to infer the NBBF metric from past cases and create a classifier to separate between congestion and contention cases.

Regarding queues, where most congestion can occur, some works have addressed their management from an ML perspective. For example, Harari et al [146] propose in their letter *NN-RED* a NN architecture to predict future queue occupations and decide when to drop packets.

The authors of [147] demonstrate how an ML classifier hosted in an intermediate node can predict the TCP congestion window size used on the sender of a TCP connection. For this, they use ML over TCP cross-traffic flows and metrics, including bandwidth, delay, jitter and packet loss.

In addition, congestion can be approximated through the estimation of different network parameters which values might indicate congestion, such as throughput or *Round Trip Times* (RTT). For the former, the authors of [148] propose a lightweight SVR (Support Vector Regressor) system that estimates network throughput by combining history-based records with "end-to-end path properties".

Edalat et al [149] propose the *SENSE* system (Smart Expert for Network State Estimation) that provides "near-future" estimations of the network conditions by means of Machine Learning. Specifically, it modifies a combination of fixed-share learning and exponentially weighted moving average (EWMA) which works better than other proposed approaches, such as the EWMA used in TCP or unaltered fixed-shares.

Autonomous Networking

Lately, research has aimed at the development of the autonomous network paradigm, where networks are composed by intelligent components prepared for *self-adaptive and self-managed networking* to enable full automation of computer network management and configuration [150]. In their paper, Barron et al propose an autonomous system that correlates and monitors network events that are analyzed using ML to provide automatic response and actions.

Broadly, autonomous networking includes all the processes and systems that support network self-configuration and self-management in order to reduce costs and improve user experience (sometimes known as *Quality of Experience*). Very often, such autonomous processes involve the creation of network data planes or monitoring systems.

In this light, Mestres et al propose in [112] a new paradigm for networking, called *Knowledge-defined networking* where a *Knowledge plane* is added into modern network structure. Zorzi et al [151] proposed *Cognitive Network Management* as a framework, where novel AI technologies such as DL or RL could be used for autonomous network management. Ayoubi et al [113] have recently reviewed and extended this concept into a prototype system that deploys the *MAPE* control loop (Monitor-Analyze Plan-Execute over a shared Knowledge) into networks supported by ML to implement cognitive network management.

Challenges in Autonomous Networking

In spite of previous success, there are still many problems and challenges that complicate the introduction of a cognitive network management system. One of the most relevant limitations is that of *data scarcity*. The key element in AI success is the availability of quality labeled datasets. In the field of computer networks, such availability cannot be always ensured, as very few entities, mainly network operators and vendors, may have access to observational data which typically lacks accurate labels, is not consistent, unstructured or contains other flaws.

Actually, many authors raise this issue in their works, specifically referring to the lack of data, the costs of collection and the importance of simulation data [152]; the necessity of standardized and benchmark datasets made available to the public, using in their work publicly accessible datasets [112]; the challenges arising from the collection of such data, such as storage or data analysis [153] or showing the opportunity for upcoming technologies, such as SDN, to introduce network data monitoring systems aware of ML needs, notwithstanding the potential risks and overheads associated with the process [154].

There are tools in the literature for traffic generation, such as the well-known Iperf [155] or the D_ITG [156] which relies on stochastic processes to generate realistic traffic at packet level. Still, the labeling process of such datasets requires other planning tools, such as Net2Plan [157]. In this line, projects such as *GEANT* [158] or *SNDLib* [159] have contributed by sharing traffic matrices and other network components in different datasets openly.

ML in Optical Network Problems

AI and Big data popularity have been also proposed to improve optical networks, both at physical level and the link layer. A survey on ML applied to optical networking problems is available at [115].

For instance, the authors in [160] propose an ML system to estimate the quality of transmission of optical lightpaths based on simple parameters known before deployment. Concisely, the authors predict whether the Bit Error Rate (BER) will exceed a predefined threshold.

Shahkarami et al [161] propose a simple ML-powered anomaly detection system for BER values in optical networks that trades-off model complexity and monitoring details. Moreover, the previously described works from Mo et al [141] and Gutterman et al [142] address the detection of power excursions in ROADMs, specially when adding new wavelengths.

Similarly, the authors of [162] propose an ML estimator for the *Signal To Noise Ratio* (SNR) of networks. Indeed, the system is conceived as a learning system introduced between brownfield and greenfield planning to help users reducing SNR by design.

The authors of [163] develop the *VENTURE* system to monitor and optimize optical networks. The system is supported by traffic prediction to optimize overprovisioned resources, specially optical transponders. In the same line, the previous work from Yu et al [139] uses DNNs to predict and optimize the assignment of resources within a datacenter optical network.

Nevertheless, in [164], its authors review some of the major risks and problems arising from the incorrect application of ML to optical networks in particular. Specifically, authors show the risk of overestimating performance gain, specially when neural networks deal with *Pseudo-Random Bit Sequences* (PRBS) or other types of random sequences. Particularly the authors raise concerns regarding reproducibility of such experiments and state the importance of transparency in this line of research.

Routing and Wavelength Allocation Problem

Routing and Wavelength Allocation (RWA) is the algorithm for mapping traffic demands to routes and wavelenghts (i.e. lightpaths). RWA and similar problems have been solved by complex Integer Linear Programming (ILP) optimization formulations, which provide costly but optimal numerical results [165].

However, the complexity introduced by these methods is usually translated in terms of too long execution times, being completion times too long for large networks. In this light, many heuristic solutions have been proposed to provide faster configurations at the expense of producing suboptimal results.

Other studies have addressed the RWA problem and similar ones from different perspectives in order to propose a more efficient and less suboptimal solution. For instance, in [166] the authors address the *Optical Regenerator Placement* (ORP) problem, similar to RWA using genetic algorithms. In detail, the authors re-state the ORP problem and propose the genetic algorithm solution.

In their early work, Pointurier et al [167] propose an RL system that tackles the routing part of the RWA problem. Concisely, the authors leverage the generalization abilities of RL to select the most optimal route among a predefined set of them. Wavelength assignment is assumed to follow a first fit strategy. Results show that

01	Traffic Classification	• Deep Package Inspection • Unsupervised learning (Clustering) • Supervised learning (ML classification) • Hidden Markov models
02	Routing	• Dijkstra and Distance Vector • Supervised learning (ML classification) • Deep Learning (CNN)
03	Quality of Service	• Supervised learning (ML classification) • Quality of Experience (QoE) correlation • Sentiment analysis from Facial Images • Recommender systems
04	Resource Allocation	• SDN-powered networks • Reinforcement Learning • Deep Learning • Supervised learning (ML classification)
05	Network Traffic and Congestion Control	• Maximum Likelihood estimator • Hidden Markov Models • Expectation Maximization • Supervised learning (ML classification) • Exponential Moving Weighted Average
06	Autonomous Networking	• Self-managed Networking • Knowledge-defined Networking • Cognitive Network Management • MAPE control loop • Challenge: Data Scarcity
07	Optical WDM networks	• Supervised learning (ML classification) • Anomaly detection • Deep Learning and Reinforcement Learning • SDN (ROADMs) • ILP and Heuristcs for RWA • Genetic Algorithms

FIGURE 2.2: Summary of the most prominent methods used to address each of the problems surveyed.

the proposed solution can reduce the blocking probabilities with respect to other methods.

Chen et al propose in [168] a self-learning routing protocol based on RL and DL to provide routing and spectrum modulation assignment in the context of optical networks. The solution trains a Deep Reinforcement learning solution called *Q-learning* that learns a set of policies to apply to the network RMSA configuration according to different key elements of the network, such as topology or spectrum utilization. The authors claim to achieve a "significant reduction" in the blocking probability obtained by the solution.

Finally, the authors in [169] review all basic concepts regarding *Elastic Optical Networks* (EONs) and perform an extensive review on all available *Routing and Spectrum Allocation* (RSA) algorithms. RSA algorithms represent a more general case of the specific RWA problems.

As a synopsis of this section, Figure 2.2 aggregates the distinct tools used to tackle the problems surveyed. Such tools mainly come from AI and ML, but other relevant tools are also included.

2.4 Conclusions and Progress Beyond the State of the Art

Throughout this review, we have identified different gaps in the surveyed areas which correspond to those parts we have contributed to with the work of this thesis. Concisely the specific gaps we target on this thesis are the following:

1. Analysis datasets in the literature are not very large, only some few large datasets; in Android malware we analyze from 100K applications to up to 1.3M and propose scalable approaches that can work with even more data.

2. Meta-data from Android application markets has been considered, but has not been analyzed in depth for the detection of malware. In Chapter 4 we propose a detection scheme for malware based on meta-information.

3. We find and tackle unresolved issues regarding AV engines lack of consensus, which are observed and analyzed in Chapter 5 along with the discovery and verification of different behavioral patterns for engines.

4. We propose SignatureMiner to homogenize AV signatures and use it to inspect classes and categories over a large dataset of them in Chapter 6. This way, we propose a solution for the *Name Inconsistencies problem*. Furthermore, we use the new consistent names to infer a categorization scheme for malware families and propose a ML classifier that identifies Unknown malware families.

5. We find a lack of scalable and fast alternatives to detect application repackaging at market scale, which is why we propose our approach: CloneSpot that also relies on meta-data (Chapter 7).

6. There is a clear lack of labeled data and standardized datasets in the networking field. Netgen is proposed to alleviate such problems. Using this tool, it is possible to quickly generate traffic matrices and the solution for different network problems. (Chapter 8)

7. Only some few networking problems have been addressed from an ML perspective; those that correspond to a classification/regression problem. We propose in Chapter 8 an alternative methodology to emulate ILP solutions of RWA, which is a perfect example of non-straightforward classification/regression problem.

Chapter 3

Methodology and Tools

In this chapter, we revise the main methods and technologies utilized along the work in this thesis, specially regarding AI and ML technologies. Artificial Intelligence is a very broad discipline that comprises algorithms, technologies and routines which attempt to provision a computer (or any other device) with some sort of intelligence, typically for narrow problem solving. Strictly speaking, current AI is aimed towards enhancing machines' abilities to perform specific tasks rather than obtaining what is known as *Artificial General Intelligence (AGI)* [170].

Through this chapter, we initially perform a brief review on data mining algorithms and techniques in Section 3.1, to continue with an overview of the main algorithms and issues regarding ML in Section 3.2 as well as other relevant statistical methods in Section 3.3. Finally, we conclude the chapter in Section 2.4 with a brief recap and overview of the elements in the chapter.

3.1 Data Mining

Data Mining [171] consists on the automatic extraction of patterns from data in any form that has evolved from the well-established field of *Knowledge Discovery in Databases* (KDD). In general, any procedure that extracts valuable information from available data, not only in databases, but also in text (text mining) or the Web (web mining) among others is considered data mining.

At present, data mining involves any process aimed at understanding and analyzing data collections with unknown or nonexistent structure coming from different sources. There are many resources and procedures to label and categorize data mining. In this chapter, we will target data mining as the ability of extracting information from data, considering other fields such as Machine Learning outside of the Data Mining domain.

3.1.1 Distances and Item Similarity

Distance metrics are frequently useful to compare two or more elements in a quantitative manner, that is, how close they are. Strictly speaking, a distance inside a space of points, that receives as input a pair of points x and y from that space, is a function $d(x, y)$ produces a real number. Such number is non-negative, zero only in case both points are identical ($x = y$), equal to the symmetrical distance ($d(x, y) = d(y, x)$) and satisfies the well-known triangle inequality:

$$d(x,y) \leq d(x,z) + d(z,y) \quad \forall x,y,z \tag{3.1}$$

that intuitively indicates that the distance metric has to be the length of the shortest path between the given points. Below follows a short summary of popular distance metrics and functions.

Euclidean Distance

Is the most straightforward and common definition of distance inside an n-dimensional Euclidean space where points have n dimensions and for which distance is the ℓ_2 norm:

$$d(\vec{x},\vec{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{3.2}$$

being $\vec{x}, \vec{y}$ two points in the n-dimensional Euclidean space. The distances based on different norms can also be used, and they receive the generic name of ℓ_r norms. In this case $r = 2$ is the above-defined ℓ_2 norm. All these come directly from the algebraic Euclidean space definition and have been the most popular metrics for numerical values.

Cosine Distance and Cosine Similarity

Cosine distance is a metric to estimate how far away two numerical vectors are. The cosine distance measures the distance between two vectors in terms of the cosine of the angle separating them. When translated to actual degrees or radians (by means of the arc-cosine function) it becomes the *angular distance*.

Cosine distance is better known by its opposite metric, *cosine similarity*, that measures how similar two vectors are. Usually, similarity is left as a cosine value, so the value is bounded between -1 and 1. Cosine similarity is computed as follows:

$$\vec{CS}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\|a\| \times \|b\|} \tag{3.3}$$

Edit Distance

The edit distance is a special distance metric best suited to compare strings numerically. Simply, the edit distance between two strings a and b is the smallest possible number of character changes to be applied to string a so it becomes string b. For instance,

$$ed(hello, world, hello, world!) = 1 \tag{3.4}$$

The prior equation depicts and example of the edit distance. Basically, the distance between the two input texts is the trailing exclamation sign and, therefore, has edit distance equal to one.

Jaccard Similarity

The Jaccard similarity is a similarity metric to compare itemsets, that is, groups of elements of any type. In general, the Jaccard distance compares sets using union and intersection of the elements inside both of them. The Jaccard similarity between two sets of items A and B is defined as follows:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.5}$$

That is, the amount of items present in both sets (intersection) divided by all items appearing in any of the two sets (union). The Jaccard similarity is bounded between 0 and 1 and the more common items two sets have, the larger their Jaccard similarity is.

3.1.2 Locality-Sensitive Hashing Methods

Hashing functions are *one-way* functions that transform an arbitrary length input into a fixed-length string of bytes that cannot be inverted; that is, once hashed, the

function cannot be reverted back to the input content. Common Hash functions are MD5 or SHA1. In addition to their well-known applications in security mechanisms and protocols, hashing functions can be leveraged as very efficient indexers.

In this context, *Locality-Sensitive Hashing* methods comprise a set of techniques that propitiate the aggregation and indexing of similar elements within large collections of data. Different techniques can be applied in different situations and the rest of this section is devoted to summarize the most well-known.

Feature Hashing

Feature hashing [172], also known as the *hashing trick* is a dimensionality reduction method that consists on applying hashing functions over the existing input features of an ML problem. These features can be structured and numerical or even arbitrary and unstructured, like free text.

The idea for the dimensionality reduction mechanism is to leverage collisions in certain hashing methods to define *feature buckets* in which input features can be mapped and consistently reduced to the number of buckets assigned to the algorithm.

This way, large feature sets can be compressed into a fixed predefined length that can be useful to reduce feature space in very large datasets following a methodological approach.

Min-hashing

Min-Hashing is a very fast algorithm for grouping together similar sets of objects of any type. It relies on the collisions of hash functions to group together similar itemsets such that, for any two collections, their probability of collision is the *Jaccard similarity* between them. Therefore, the more similar two items compared by min-Hashing are, the more likely they will output the same min-Hash falling into the same *bucket*.

A classical example of min-Hashing is that of text similarity. For similarity grouping, all the texts in a collection are split into several chunks of equal size k called *shingles*. Hash functions are applied to each shingle in such a way that for each text, a number of hashes (that can be converted to numbers) is obtained, of which the minimum is selected as index and called the *min-hash*. Even when used with large text corpora, once the min-hash is computed for each entry, the aggregation is instantaneous, since similar elements just need to be grouped by min-hash.

3.1.3 Market Basket Analysis and Frequent Itemset Mining

Frequent Itemset Mining (best known as the *Market Basket Analysis* model) are a set of techniques and algorithms that seek for patterns from data by associative observation. The market basket analysis term comes from the inspection of the elements

bought together by any client inside a supermarket under the assumption that there exist inherent relations within elements of many clients' shopping lists.

For instance, it is very probable that someone who buys hot-dog bread will also buy hot-dog sausages. While this insight is straightforward, continuing on this analysis, other interesting patterns may be that such a shopping list probably includes ketchup or fries as well. From a business viewpoint, this information is useful to understand and design how to better configure market shelves so people find most commonly bought together items at a glance.

Rule Mining

Rule mining or association rule learning seeks finding patterns or rules that govern the apparition of frequently common items, that is, rules capable of predicting the probability of an item to appear given the presence of others. The most prominent method for rule mining is the *Apriori* [173] algorithm. The apriori algorithm iteratively counts the number of times each set of items appears together following a *breath-first search* strategy to mine itemsets efficiently. The following list summarizes the most important metrics and values that association rules find and compute:

- Support: is computed as the frequency of an item with respect to the total.

$$Support(X) = \frac{\text{Frequency of item X}}{\text{Total number of transactions}} \tag{3.6}$$

- Confidence: shows how frequently a certain rule appears within all detections with respect to the observed behavior.

$$Conf.(X \rightarrow Y) = \frac{Support(X \cap Y)}{Support(X)} \tag{3.7}$$

- Lift: computes the ratio of frequencies for each part of the rule if they were independent from each other:

$$Lift(X \rightarrow Y) = \frac{Support(X \cap Y)}{Support(X) \times Support(Y)} \tag{3.8}$$

A detailed explanation of distances, hashing methods and frequent itemset mining along with other data mining methods and their specific details and demonstrations may be found in [174].

3.2 Machine Learning

Machine Learning (ML) is one of the core enabling technologies of Artificial Intelligence. Machine Learning algorithms are able to analyze past data records and discover patterns and structures that might be repetitive in the future. Provided it has

sufficient data, an ML algorithm is usually able to generalize and perform accurate predictions over unseen data points.

ML tools are optimization systems that try to minimize the value of a given error function, such as the logarithmic probability or similar. During this optimization, which is called training, some weights or internal parameters are updated in the models thereby enabling the prediction of future samples. As a result, machine learning tools are very powerful mathematical algorithms capable of *learning* the way to solve a problem inspired by past data observations. In this process, ML is able to identify patterns from data and look for them in new data observations.

Generally, ML problems go through two phases (neglecting, for now, model validation): (i) the *training phase*, when the algorithm is fed with training data to learn patterns and adjust itself and (ii) the *prediction phase*, when the already trained model is used to make predictions upon new data samples.

Usual tools for solving distance-based ML algorithm statements are well-known mathematical optimizers, such as *Stochastic Gradient descent*, *Newton's method* or *Coordinate Descent methods*. In addition, many algorithms often have *hyper-parameters*, which allow fine-tuning the algorithm to better suit an specific problem.

Machine learning algorithms are divided into two families: supervised and unsupervised. While the former are yielding most ground-breaking results, they require labeled samples that the latter do not, making them less versatile. The book in [175] provides a good overview of machine learning algorithms, specially feature engineering and classical supervised and unsupervised algorithms.

3.2.1 Supervised Problems

Supervised machine learning problems are those where the expected output is known at training time. All the input features or variables are coupled with one or more output variables, called *class* or *target*. This target can be either discrete and limited (classification) or continuous and unlimited (regression).

In this context, supervised ML algorithms are in charge of inferring a function $f(\vec{x})$ from the training set $\vec{x}$ that maps inputs and outputs minimizing the error, that is, making $f(\vec{x}) - y$ minimal. There are many different algorithms that tackle either regression or classification, even though the formulations of their optimization targets have been different lately. A list of some of the most well-known and famous classification and regression algorithms follows:

- *Linear regression*, where the relation between the input and the output is assumed to be linear. For classification, the *Logistic Regression* model [176], is based on a linear regression and adjusted with sigmoid curves for probability estimation.
- *Bayesian Networks* [177], that take into account Bayes theorem to model output data from input in a probabilistic manner.
- *Gaussian processes*, which compute Gaussian mixtures to approximate the observed regression distribution.

- *K-Nearest-Neighbors* [178], that looks for patterns among the closest points for each datapoint.
- *Support Vector Machines (SVM)* [179], which develop different classifiers that attempt to maximize the margin of the classification border to the closest datapoints on each side.
- *Decision Trees* [180], that select the most separating features and compute thresholds for them in order.
- *Random Forests* [181], that compute many naïve decision trees from random subsamples of data and features and uses them to solve the classification problem by majority voting.
- *Neural Networks* [182] that are stacked collections of single *neurons* connected by means of non-linear activation functions. Lately, Neural Network models have been deepened, introducing more neurons and more layers in their architectures which has enabled enormous breakthroughs.

Supervised algorithms are very good at solving estimation (regression) and categorization (classification) problems from a very simple problem statement. Nevertheless, they require labeled data (a.k.a. ground truth), which is a scarce resource. *Labeled data* means that for every datapoint inside a collection there is an associated label identifier that gives the output of the system for such datapoint, that can be either an amount or a category.

For instance, consider a classification problem where the objective is to classify whether a bank client will default a credit card based on past financial information: there, the label is whether the client will default (yes class) or not (no class) and any classifier should be trained with data points labeled by both yes and no classes. The previous example is a *binary classification*, notwithstanding that most machine learning algorithms provide support for multi-class problems.

In addition, *ground truth* is usually employed to refer to labeled data that is completely accurate, frequently by external validation. In short, ground truth is the inherent information associated to some observational data which can be available, for example in a collection of suspicious URLs that have been manually inspected and tagged as dangerous, or not, like the collection of user likes suggesting certain preferences of users but do not match to their exact preferences.

3.2.2 Unsupervised Problems

Unsupervised machine learning problems address the organization and structuring of unlabeled data collections lacking ground truth. An unsupervised ML algorithm attempts to derive some structure, category or *clustering* scheme by looking at input features only. In this case, algorithms are in charge of inferring boundaries within datapoints that can separate observations to meaningful classes.

The most typical case of unsupervised learning problem is *clustering*, which given a collection of features organizes them in groups according to some criteria, usually distance. Some specific examples are *K-means* or hierarchical clustering [183].

The K-means clustering algorithm initially defines K randomly picked *cluster centroids*, which are the center points for each of the K clusters to be computed. Each point is assigned a group according to its closest centroid and afterwards centroids are recomputed by averaging the center among all newly assigned points in the cluster. This process is repeated until no reassignment is performed.

In the case of hierarchical clustering, pairwise distances of all elements are computed and the closest points are assigned to the same cluster. Then, the process is repeated until all points get assigned to a single group, typically conforming a *dendrogram* that displays closer elements together and connected to the rest hierarchically according to their group-wise average distances.

Other interesting unsupervised method is *Principal Component Analysis* (PCA), which is capable of reducing the dimensionality of a feature set by rearranging its dimensions into a new algorithmic base. Since this algorithm is typically used in feature reduction scenarios, it will be described in the corresponding section below.

Similarly, *Deep Learning* is usually considered an unsupervised *pre-training* phase given the ability of DNNs to non-linearly transform input features into new and richer features that ease the classification task, sometimes without the need of labels.

3.2.3 Machine Learning Workflow

Data Separation: Train and Test

One of the key indicators of any type of ML algorithm is their *generalization ability*, that is, whether a given model is capable of predicting completely unseen samples correctly. To quantify this and the model predictive power, any ML workflow starts by dividing all data in at least two sets: *train* and *test*. Train data is used for model estimation whereas test data is used to check a model's generalization ability by comparing its predictions with the ground truth labels. Despite this process is also performed over train data, the test set is more relevant since its data has never been seen by the model and can better assess its response to completely new prediction cases.

When the resulting model is very good in the train set but performs poorly on the test set, it is said that such model is *overfitting*, that is, memorizing all training examples instead of learning patterns and generalizing from them. The reader should note that a model that exclusively learns all training examples by heart will not be able to generalize and thus will show poor performance.

Model Validation and Tuning

For robustness, data can be split into more chunks and combined differently to produce models trained and tested by different datasets. One way of implementing

this strategy is by using *K-fold cross-validation*, where data is split into K chunks to train and test K different models over different iterations being the test set a different chunk each time. Using cross-validation yields more robust results against the random variation arisen from the partitioning between test and train data.

In addition, cross-validation can be used for *hyper-parameter tuning*: In order to fit the most appropriate hyper-parameters to a model, different value combinations are tested over an entire cross-validation cycle each. At the end, the best combination is selected and their results are verified through a new *validation* set, which would be a third split of the data and cannot be used during the cross-validation hyper-parameter tuning process.

Data Pre-processing

In any machine learning process, it is important to follow certain policies to ensure the quality and optimality of the data analyzed. Let us define *pre-processing* as the set of actions that convert a raw dataset into an ML-ready dataset, containing clean, normalized and numerical fields.

The first step is to curate data, deal with missing and incoherent values and ensure the correct management of the different events that may arise during the analysis. Moreover, data can be *normalized* to facilitate training, specially for distance-based algorithms like logistic regression or SVMs.

Normalization consists on scaling all the values in the dataset consistently following some predefined criteria over the training set. Common normalization schemes are *min-max* normalization, where the data is scaled to fit in the value range of the train set and *standard* normalization, where the data gets subtracted the mean and divided by the standard deviation of the training set.

In addition, non-numerical features must be converted to their numeric approximations to facilitate the classification or regression tasks. There exist different approaches for text-to-number mapping in the literature which can be separated into two groups:

- **Categories**: they comprise textual features that comprise a closed set of words or characters (a.k.a. categories) that identify a subset of elements within a dataset. In general, these can be mapped through direct indexing, even though *one-hot encoding* is preferred to prevent unintended biases. One-hot encoding consists on converting a categorical feature with N categories into N different binary features, one per category that are assigned value one when the category they represent is the one from the current datapoint and zero otherwise.
- **Free-text**: Free text variables are more tricky to deal with, as the specific features must be created from the text and may not represent the optimal subset. In the literature there are many methods for this transformation, such as *TF-IDF*, *word-embeddings* or the very recent and popular approach of *word2Vec*.

3.2.4 Machine Learning Performance Measurement

Most machine learning problems attempt to predict or estimate quantities, generally approximating a real function $y = \hat{f}(x)$ with an estimated function $\hat{y} = f(x)$ that attempts to replicate as best as possible the observed behavior. There are different metrics to assess the performance of a machine learning model over different train, test or validation sets. The following list summarizes some of the most popular metrics:

- **Regression**: Scores measure the absolute difference between target and predicted values. The variables y and $\hat{y}$ denote the real and the estimated outputs respectively
  - **Mean Squared Error (MSE)**: Squared difference between target and prediction:
$$\text{MSE} = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2 \tag{3.9}$$
  - **Mean Absolute Error (MAE)**: Absolute difference between target and prediction:
$$\text{MAE} = \frac{1}{n}\sum_{i}^{n}|y_i - \hat{y}_i| \tag{3.10}$$
  - **Root Mean Squared Error (RMSE)**: Root of the average squared difference between target and prediction:
$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2} \tag{3.11}$$
- **Classification**: In classification, it is useful to turn to a binary problem that has a positive class and a negative one for illustration purposes. Given a binary classification problem, each predicted sample must fall into one of the following states: True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN). Table 3.1 depicts a *confusion matrix* of the classification task that visually indicates which of these corresponds to each pair of actual and predicted value combinations.
  - **False Positive and Negative Rates (FPR and FPN)**: Amount of misclassification cases due to failures to correctly assign one class:
$$\text{FPR} = \frac{FP}{TN + FP} \quad \text{FNR} = \frac{FN}{TP + FN} \tag{3.12}$$
  - **Accuracy**: The number of correctly classified samples against the total number of classes. Accuracy is not robust and specially misguiding when the number of positive and negative classes is unbalanced.
$$\text{Acc} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.13}$$

- **F-score**: F-Score trades-off *precision*, which indicates how many classes labeled as positive are indeed positive, and *recall*, which measures how many positive classes the model detects from all positive classes in the real data.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.14}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.15}$$

$$\text{F-score} = 2 \times \frac{Prec \times Rec}{Prec + Rec} \tag{3.16}$$

- **Multi-class F-score**: When F-score is formulated for the Multi-class problem, precision and recall values are computed for each one-vs-all possible combinations and averaged through classes, being the computation of the F-score the same for the average metrics.

$$\text{Prec}_j = \frac{c_{jj}}{\sum_k c_{jk}} \quad \overline{\text{Prec}} = \frac{\text{Prec}_j}{M} \tag{3.17}$$

$$\text{Recall}_j = \frac{c_{jj}}{\sum_k c_{kj}} \quad \overline{\text{Recall}} = \frac{\text{Recall}_j}{M} \tag{3.18}$$

$$\text{F-score} = 2 \times \frac{Prec \times Rec}{Prec + Rec} \tag{3.19}$$

where the total number of classes is assumed to be M, c_j corresponds to the number of elements assigned to class j and c_{jk} is the number of elements from class k assigned to class j.

- **Area Under the Curve** (AuC): In any classification problem it is possible to create two types of curves by using different sets of hyper-parameters, typically decision thresholds. Such curves can be ROC (Receiver Operating Characteristic) that trades-off FPR and FNR and PR (Precision-Recall) that trades-off precisely precision and recall. The intuition for these curves is that the larger the area they cover the better a model is, since the compared values typically behave oppositely. This way, the Area under the Curve can be used as a metric of goodness of fit for a model overall, indicating a good balance among the used hyper-parameter.

• **Clustering**: Due to the absence of Ground Truth in clustering algorithms, the scoring systems are mere indicators of how homogeneously distributed resulting clusters are.

- **Rand Index**: measures the similarity between two data clusters. It estimates how likely it is that two different items in the dataset fall into the same cluster.

- **Separation and compactness**: In general, a good clustering scheme must comply with two principles: clusters should be widely separated from one another whilst elements inside each cluster should be as compact, that is, as close together, as possible.

TABLE 3.1: Index of detection for any ML classification task

		Predicted	
		Positive	Negative
Actual	Positive	TP	FN
	Negative	FP	TN

– **Within Cluster Sum of Squares** (WSS): this metric is defined as the sum of all the deviations between each observation and their corresponding cluster centroid. Usually, WSS is used as a measure for cluster compactness, as smaller WSS indicate closer cluster points. However, this measurement is highly influenced by the number of samples as it just sums all deviations.

$$WSS = \sum_i (x_i - \bar{x_c})^2 \quad (3.20)$$

These metrics are typically computed for test, train and validation sets separately in order to provide different information per case. Test and validation scores can be used to check model generalization capabilities as well as the performance to be expected upon new datapoints. In addition, training scores can provide further information to help identifying concrete problems a model may show, like overfitting.

3.2.5 Feature Selection

In ML, available data are usually the key element for success. Algorithms and methods can improve and enhance results slightly, but quality training data is the core of ML performance.

In fact, not all available features should always be used; sometimes the meaningful input variables are small subsets or combinations of existing features whereas at other times many more variables and their combinations are required to obtain accurate results. In general, it is a good idea to keep the number of input features as small as possible without losing accuracy, as larger feature models are more difficult to understand and manage. Moreover, the more variables a model considers, the easier it falls into the well-known *curse of dimensionality*, where the excess of input features makes the required number of observations grow very fast to obtain acceptable performance.

In this light, feature selection comprises a a pre-modeling step to help determining the most relevant and useful variables in any learning task. Traditionally, feature selection was a manual task undertaken by a human *data scientist* who inspects input variables looking for relations, removes duplicated or redundant variables and deals with inconsistencies. In general terms, there are three types of feature selection methods:

- **Filter methods**: These score and rank variables according to their statistical relation with the target variable. They are directly applied to data collections

and before ML. For example, *F-measure*, *Gini index* or the *Chi-squared test* are filter methods that produce a score for each input feature.

- **Wrapper methods**: These methods define a strategy to add or remove ranked sets of features according to some criteria (i.e. filter method) and compute, for each of them, the expected performance inside an ML context. This way, feature selection is performed by choosing the best performing subset of features at the end of the process. Typical examples of wrapper methods are *Forward-Selection*, *Backward elimination* or *Recursive Feature Elimination* (RFE).
- **Embedded methods**: These methods are integrated into ML algorithms and perform *implicit* feature selection by introducing additional constraints in target optimization functions to prevent the use of redundant features. In other words, to select features, these methods rely on the *constrained* training process of their hosting ML optimization algorithms. Of these methods, *regularization* is the most extended case.

Lately, feature selection has turned towards *feature engineering*, where the process of data pre-processing is automated and features are transformed following pre-defined methods and algorithms that can be directly introduced in the ML workflow. *Principal Component Analysis (PCA)* or *Factor Analysis* are typical cases of feature engineering.

PCA [184] is a feature engineering process that attempts to rearrange the input features into a new orthogonal feature space. Each of the new features, called components is sorted by their contribution to the overall variance of data, such that less components are required to explain the same amount of variance with respect to the initial feature set. To do this, PCA relies eigenvalue decomposition of the input feature matrix to create a new algebraic basis along the direction of maximum variability.

Furthermore, deep learning can be used as a process of feature engineering where input features are combined and selected optimally to create better inputs to ML algorithms. The most relevant example of unsupervised feature selection using DL is the well-known *autoencoder* architecture, which intermediate layer contains less features that input/output. In this setting, the optimization goal is to reduce the difference between the input and output, that should be identical. This way, the NN is forced to learn a non-linear compression scheme that reduces features in the intermediate label without losing information.

3.2.6 Deep Learning

Deep learning (DL) or *Deep Neural Networks* (DNN) is the most relevant breakthrough in Machine learning in the last decade [185]. Deep learning is rooted on Artificial Neural Networks, which consist on layers of simple artificial neurons such as the perceptron. Although neural networks date back from the early 50s, they were abandoned during the AI winter due to their huge computational costs and data requirements and have been retaken recently, as they achieved new milestones like the stunning results in the MNIST ImageNet challenge [1].

In short, a deep learning algorithm stacks layers of neurons together. In the case of DNN they are conceived with many more layers than simple NNs to create a *deep* structure that combined with non-linear activation functions provides one of the most accurate and versatile machine learning algorithms. Fig 3.1 depicts a sample scheme of neural network showing a couple of hidden layers. In DNNs, the number of layers is a typical hyper-parameter which requires adjustment.

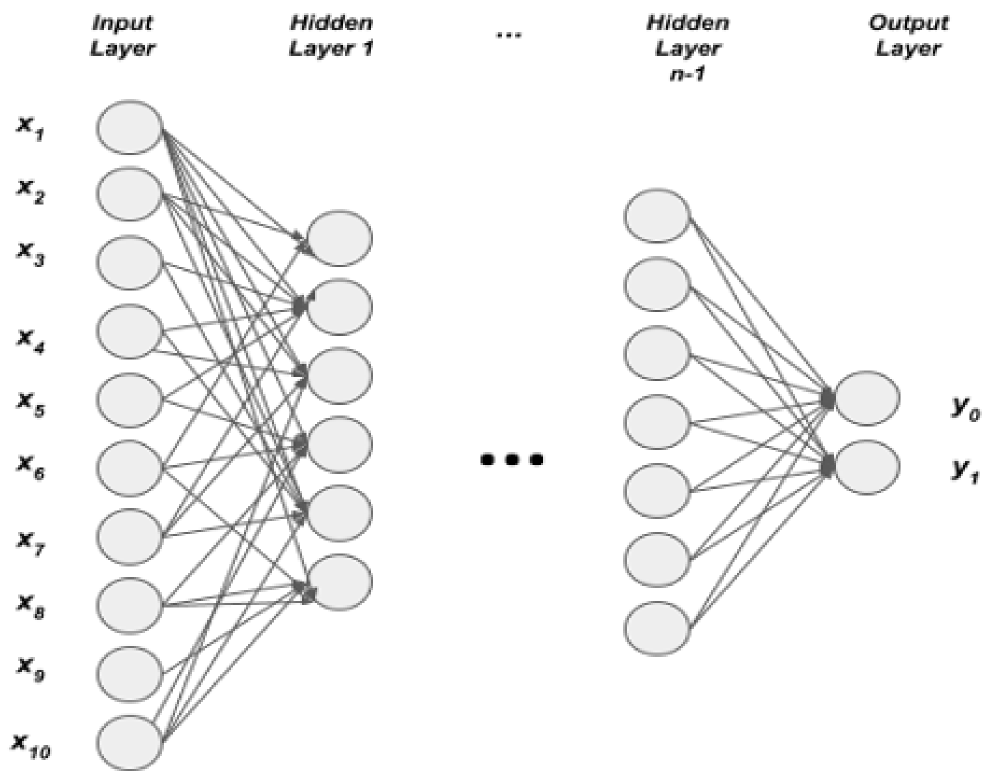


FIGURE 3.1: Sample image of an Artificial Neural Network. DNNs follow the same scheme but instead of having a single hidden layer, they have many of them, each one containing many neurons.

The key issue in deep learning is its ability to successively transform input variables at every new layer until the classification task is easily separable by a simple classifier. The produced output at each layer is a non-linear transformation of the input that is more suitable for the next step.

Many researchers have developed more focused and task-oriented architectures, such as *Convolutional Neural Networks* (image recognition), *Recurrent Neural Networks* (Speech Recognition) or *Long-Short Term Memory Neural Networks* (Time series analysis). In fact, the architecture of a deep neural network is a broad field of research with different proposals for different situations [186].

Deep Learning has still shortcomings. Any DNN requires a lot of data to produce generalizable results and is prone to overfitting, although techniques such as dropout [187] have been effective in mitigating such effect. Dropout is a robustness technique that randomly *kills* certain neuron connections at training time, in such

a way that different stages in the training phase have to deal with slightly modified versions of the NN architecture during each training phase. This way, learning the entire dataset instead of patterns is harder for the network since training data is randomly distributed over different subsets of networks.

Furthermore, training a deep neural network requires operating over billions of parameters, which results in a computationally costly process. However, hardware improvements in GPUs or the new TPUs proposed in [188] are improving the training phase and enabling better models. Together with the development of much more efficient software and distributed computation models, DL can bring several breakthroughs and advancements in many fields, like autonomous vehicles or real-time translation.

3.3 Other Statistical Methods

3.3.1 Latent Variable Models

Latent Variable Models (LVM) are a family of statistical models that attempt to identify statistical relations and causality by means of assuming the existence some unobserved *latent* or *hidden* variable. Such variable would explain the observable interdependencies and relations of the existing variables. Even though latent variables do not exist explicitly, they explain such relations and thus, exist implicitly.

In this light, Latent Variable Models provide a range of tools and methodologies to hypothesize and confirm or deny whether latent variables exist, being possible in some cases the estimation of such variable's values.

Some examples of LVMs are *mixture models* which represent the presence of subpopulations inside a broader population, *factor analysis* [189] that helps in describing the variability among observed correlated variables in terms of latent variables that are called *factors* or *Structural Equation Models* (SEM) [190] which is a set of mathematical and statistical models along with computer algorithms that assess whether hypothetical structures fit to a given data collection.

3.3.2 Graph Modeling

A graph is a data structure representation formed by points joined together by lines connecting some of them according to the different relations among the different entities represented by points. Concisely, graphs are mathematical structures that model pairwise relations between different objects.

A graph is generally made up of nodes (previously points) that are connected with other nodes by means of edges (the previous lines) following relational patterns, that is, according to the relations existing between nodes. Graphs can be directed, where each edge has a direction consistent with the relation between the given pair of nodes or undirected, where edges simply denote an existing connection shared by both nodes. Graph theory facilitates the study of networks and the

relationships among its members by using different methods, such as distances, similarities or adjacencies.

Another good feature regarding graphs is that they can be represented graphically to offer a very efficient visual representation of entity relations across a network at a glance. Relations between graph entities (nodes) are generally modeled by different data structures such as the adjacency matrix which directly states pairwise relation strength, or the incidence matrix that represents the number of times each node is incident with each edge.

In addition, graph analysis facilitates several procedures easier to solve graphically rather than analytically, such as sub-graph computation, relationship exploitation or connection exploration.

Graphs have been used broadly in the literature, specially to represent people within organizations, like companies or Social Networks. In this context, graphs can be used to infer communities within the full graph, using for instance the *edge betweeness* [191] algorithm; or weighting the strength of relations in a Social Graph [192].

3.4 Summary and Conclusions

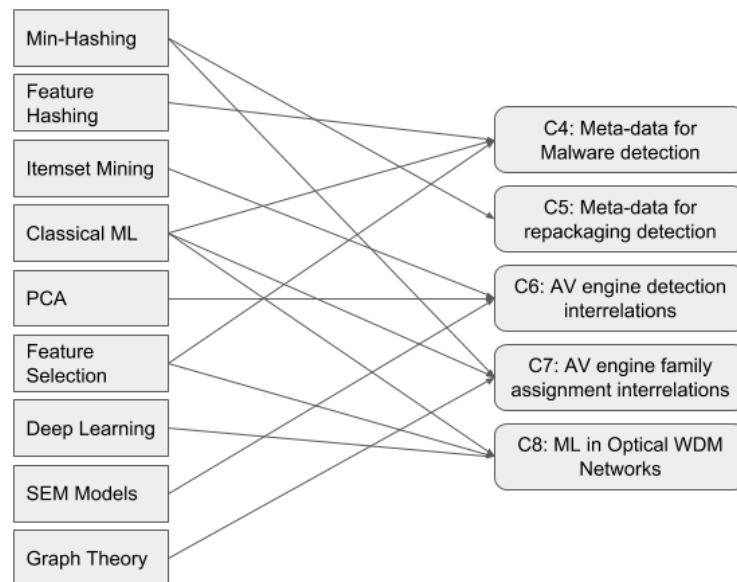


FIGURE 3.2: Summary of the main technologies used along the thesis and their pairing with the different chapters and contributions

This chapter has presented the methods and tools employed to reach the different contributions in the thesis. Most of the tools reviewed here are either used in some proposal or needed for the understanding of the former. The interested reader can refer to different books for further references, in data mining [174], machine

learning [175], deep learning [193], Structural Equation Modeling [194] or Graph theory [195].

Overall, the most utilized methods reviewed here are ML and Data Mining, specially hashing techniques. Actually, some ML classifier or regressor is used in all chapters, with the exception of Chapter 5, where SEM is used instead.

In Chapter 4, different ML classifiers, namely logistic regression, SVMs and random forests are used along with filter and wrapper selection methods, namely Step-AIC, Chi-Squared or the Gini index. Besides, feature hashing is used to reduce the number of permission-related features. Indeed, the main goal of the chapter is developing malware detection classifiers using meta-data fields which are evaluated, selected and improved through feature selection techniques.

Chapter 5 uses many of these tools, including rule and itemset mining, SEM equations, correlations and PCA. All those tools are helpful to unveil AV engine relationships. Correlation indicates engines of the *follower* kind, but specially its lack points to *eccentric engines*. PCA shows the need of many engines for complete representation of the dataset whereas rule mining unveils the existing causal relations among followers. Finally, SEM tools are used to verify the prior discoveries and to infer malware risk.

Concerning Chapter 6, min-hashing along with some standard text-processing enables the good performance of *SignatureMiner*. Then, correlations and graph modeling techniques are used for the analysis of malware families and two ML algorithms, logistic regression and random forests, are leveraged to analyze each engine contribution and to classify malware between harmful and adware.

Min-hashing is the core technology as well in Chapter 7 together with two distance metrics, namely edit distance and cosine similarity. In this chapter, min-hashing is used to aggregate similar applications according to their meta-data and distances are used to compute scores for each app-set.

Finally, in Chapter 8, the most relevant algorithms and tools used are ML algorithms again, in this case logistic regression with embedded regularization (ℓ_1 and ℓ_2) and a deep neural network. For completeness, Fig 3.2 visualizes all technologies and their usage through this thesis.

Chapter 4

Android Meta-data for Malware Detection

This chapter introduces our ML-based malware detection system which uses meta-data as features and multi-scanner outputs as labels. The proposed feature set, meta-data, is presented in Section 4.1 and the specific dataset and its contents for this chapter are detailed in Section 4.2. In 4.3, different meta-data fields are analyzed using ML and feature selection techniques to assess their predictive power and Section 4.4 describes the specifics and experiments of the malware detection system. Finally, Section 4.5 ends the chapter and highlights the most relevant conclusions.

4.1 Android Application Meta-data

Android is the prevalent mobile operating system in the market. Its share reaches about 86% of the total [196], which makes it a target objective for many stakeholders,

both benign (manufacturers, users, developers...) and malicious (*scammers*, *malvertisers*....). As a result, the Android ecosystem is huge and diverse, including tons of distinct system *flavors*, alternative application markets and millions of applications.

Within such ecosystem, the number of potential threats to device and user security are limitless, whereas the ability to fight against them is much more restricted. One of the main reasons for this is the difficulty to analyze and track every malware sample published per day. Even under the best conditions, malware analysts cannot keep up with the pace of malware developers, of more than $360,000$ new samples daily [197].

Traditionally, suspicious samples are manually analyzed by experts, also known as *malware analysts*, who have to decide whether an application sample is malicious or not. This approach is not scalable and alternatives are needed to boost malware detection, such as large-scale classification of application features. Other approaches based on application code classification or automated inspection are useful, but easy to evade by modern malware (See section 2.1 for more details).

In this light, *Meta-data* poses a great candidate for fast Android application analysis, as it includes data that cannot be easily hindered. *Meta-data* or *meta-information* is the set of features containing relevant details about an application which are not contained in the application itself. Instead of being part of the binary program, meta-data provides informative details of application functions and features to the user before installation. Although incomplete, the following list presents some of the most relevant meta-information features within an Android application context:

- **Application Title**: The title given to the application.
- **Application Category**: The category each application is associated to, typically assigned by the application market.
- **Application Description**: Textual description of the application functionalities and features. The quality of this description entirely depends on the developer of the application.
- **Application Developer**: The name of the developer account responsible for the application, which is managed by each distinct market.
- **Upload Date**: The date of upload of the application.
- **Update Date**: The date of the last application update.
- **Version**: The current version of the application.
- **Number of Downloads**: The total number of user downloads of the application.
- **Application Size**: The total size of the application in bytes.
- **Application Permissions**: Items that identify the application capabilities inside a device.

Actually, many other features could be considered meta-data, as long as they describe intrinsic aspects of the application and are not part of the application code. Meta-information exists, either explicitly or implicitly, for every single application. For instance, application title, description or category are present even though they are not reported. Hence, a very good indicator for application quality is its meta-data: An application with extensive and detailed meta-data is often more trustworthy than another lacking details and verbosity in descriptions.

As a result, legitimate developers have no problem in generating and maintaining quality and updated meta-data, since application transparency attracts more users. On the contrary, the meta-data of malware applications must hinder application malicious practices to lure users into installation, even though malware developers may not spend time and effort in curating it. Hence, malware applications' meta-data could display traits that differentiate malware from legitimate applications.

Furthermore, since meta-data is a requisite in many trustworthy markets most applications must declare at least a subset of meta-data fields to appear visible for victims. For all this, meta-data poses as a good indicator for malware detection, even capable to detect *zero-day* malware that cannot be targeted by classical AV engines. In this light, this chapter reviews and evaluates meta-information fields as predictors for malware detection.

4.2 Meta-data Collection for Malware Detection

The collection of applications studied during this chapter has been collected from Google Play store, the official Android market with one of the largest application counts of over two million applications. The market is installed in Android mobile devices and accessible via web [198]. In both places, each application is presented by its meta-data and other applications recommended by the market.

Concisely, the collection comprises $118,000$ Android applications obtained from Google Play during 2015 by the Tacyt cyber-intelligence tool. Tacyt is a platform developed by *Telefonica Security & Privacy* which downloads applications from different application markets like Google Play and enriches them with external sources of information, such as antivirus detections. Applications are served in a market-like platform which enables users to perform application search and advanced analytics. Tacyt information includes application packages, meta-data, other binary details and even antivirus detections from different multi-scanner tools. This way, Tacyt users are provided with a very large, rich and complete database of Android applications they can use for analysis and malware prevention [199].

The main meta-data fields in this dataset are related to quantifiable information, such as permissions, size, number of downloads, star ratings and, in some cases, categorical items like developer or certificate issuer names from which numerical reputations can be extracted. Besides, every application has been inspected for malware by different AV engines from multi-scanner tools provided by Tacyt. Table 4.1 provides the most prominent figures for the collection.

TABLE 4.1: Dataset overview: The applications have been collected from Google Play in 2015.

Dataset element	Dataset Figures
Number of applications	118, 846
Applications flagged as malware	69, 918
Number of single-detection applications	34, 025
Number of developers	53, 780
Number of certificate issuers	44, 244
Number of different permissions	21, 541
Number of intrinsic features	15
Number of social-related features	7
Number of reputation features	2

Features are classified into four different categories, namely *intrinsic application features*, which are related to the pure application information, *social-related features*, that indicate how popular an application is, *entity-related features*, which keep track of developers and certificate issuers, and *malware detection attributes*. Below follows a comprehensive description of the aforementioned feature categories. As a summary at the end of this section, Table 4.2 gives the details of every feature in the dataset, their description, type and specific category.

4.2.1 Intrinsic Application Features

Intrinsic application features are related to information about the application binary (a.k.a APK from Android aPplication pacKage), including its size in bytes, the application category, the number of images or files included, etc. In total, this group comprises 14 features.

Application permissions are also included in this intrinsic category. Due to Android Permission system, there are over 21K different permissions used by the applications in the dataset, being the most popular ones:

- android.permission.internet (found in 96.07% of apps)
- android.permission.access_network_state (91.15%)
- android.permission.read_external_storage (54.5%)
- android.permission.write_external_storage (54.12%)
- android.permission.read_phone_state (39.81%)

Permissions can be *self-defined* and, therefore, many permissions seldom appear in the dataset more than once. Self-defined permissions in an Android application refer to those applications created ad-hoc as a combination of already existing permissions to suit the specific needs of application developers.

Due to this open design, the potential one-hot encoding space size for permissions is enormous. Indeed, out of the potential few tens of permissions defined by the Android system, we are reporting up to 21K distinct permissions, which would force 21K different features for classification. Instead, we will consider *feature hashing* (see Section 3.1 for details) to reduce the number of permissions involved efficiently.

4.2.2 Social-related Features

These are the result of social interaction within the market. The group contains seven features collected from the opinions and votes of different Google Play users that install applications and rate them according to their experience. Using this information, Google Play weights results and offers an average vote and other interesting figures of the quality of each application.

The rating system is formed by stars assigned once per user and application which are reported separately in each application's meta-data page. Therefore, the features include the number of votes for each of the five possible scores, the weighted average and the total number of votes per application.

4.2.3 Entity-related Features: Developers and Certificate Issuers

Google Play gives detailed information about application developer (name, email address and website) and the issuer of the certificate (expedition or expiration dates, name of issuer, subject names, etc). Most of these are categorical or textual values that cannot be directly utilized but they might be transformed into numerical values.

In the dataset, there are around 53K different developer names and 44K certificate issuer names involved in the development and signature of more than 120K applications, suggesting some developers and issuers are responsible for more than one application. Typically, it is common for malware and goodware developers to continue producing what they develop and hence, tracking their *reputation scores* might be helpful to discriminate applications.

In this light and following [200], we have created two new features called *developerRep* and *issuerRep* which are computed as the amount of applications declared malware from the ones they have developed/signed. Thus, malware developers will have this reputation very high whilst goodware developers should have it close if not equal to zero. When used for classification, both metrics are computed only using the training set, forcing new entities' reputation, those appearing only in test/validation sets, to zero.

The reader must note that Google Play allows self-signed applications, i.e. applications where the same entity develops and signs an application. However, their reputations may change, as many issuers may not sign their own applications exclusively and not all developers self-sign their applications.

4.2.4 Malware Detection Attributes

The Tacyt system provides a number of AV detections coming from multi-scanner tools. For this chapter, these detections are considered in binary form (malware/goodware) per different engine. Out of the near 120K applications approximately 69K have been tagged as malware by at least one of the 61 engines in the multi-scanner tool connected to Tacyt.

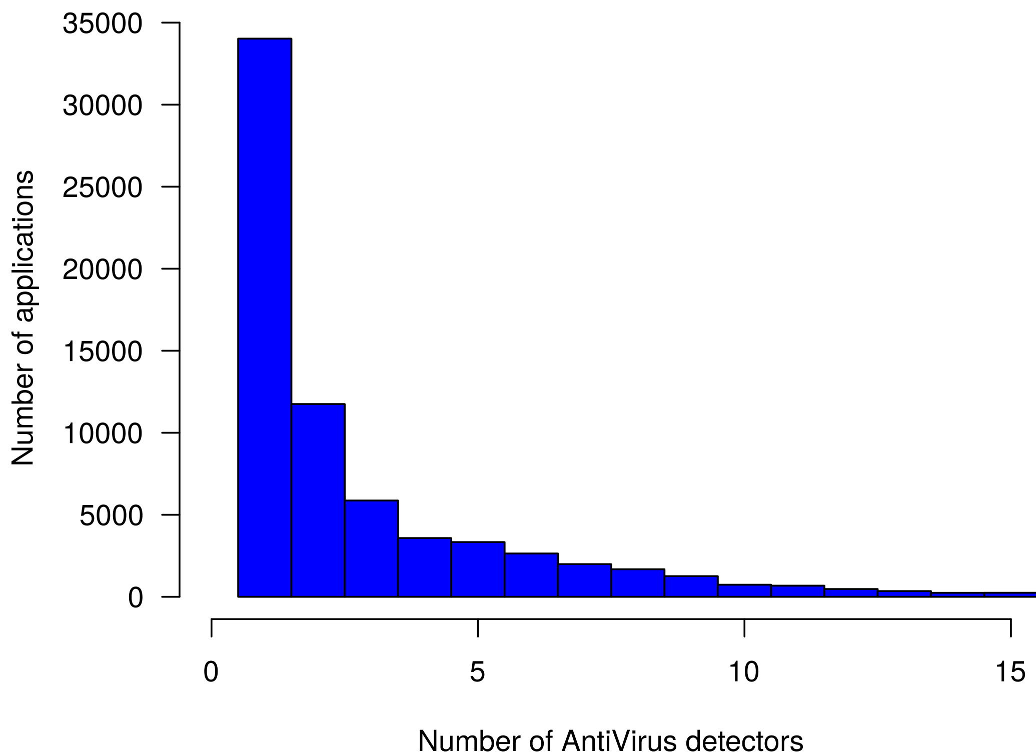


FIGURE 4.1: Histogram of AV detectors per malware application.

The detection pattern of these applications follows a zipf-like pattern, where most malware applications account a single Antivirus (AV) detection and a small amount of samples get a very large detection count. Particularly, the first quantile is one, the median is two and the third quantile is four AV engines.

Fig. 4.1 depicts the histogram of application detection counts, where the zipf-like shape is clear with most applications detected by a single engine ($34K$ applications) and an average detection of three. To mitigate differences arising from different AV labeling schemes, we will consider these distribution quantiles as thresholds to establish the ground truth for experiments.

4.2.5 Dataset Benchmark

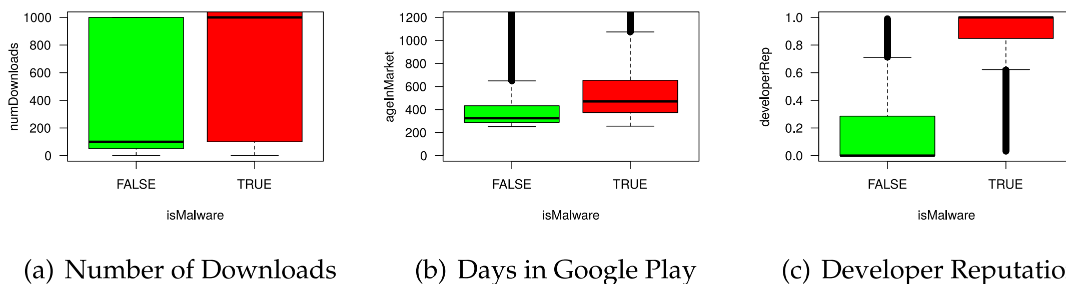
To facilitate experiments and perform an exhaustive analysis, nine different datasets have been created by randomly selecting different subsets of 50K applications with certain constraints. For each different subset, two elements are modified: malware

percentage and AV threshold. Each set contains an amount of 2, 25 or 50% of malware applications and a threshold for malware of 1, 2 or 4 AVs following detection count quantiles.

For example, the dataset containing 25% malware and the malware consideration of 1 AV engine will be referred as (1-AV, 25%) and will contain 75% goodware and 25% malware randomly selected among all applications whereby at least one AV engine has detected malware. Exceptionally, the size of the (4-AV, 50%) dataset contains roughly 36K application samples due to the lack of malware applications detected by four AV engines or more.

These benchmarks are conceived to systematically analyze and compare different detection patterns, algorithms and methodologies both in terms of malware sensibility and amount. This way, we can evaluate how malware sensitivity (engine detections) and amount affect different classification schemes.

4.3 Analysis of Meta-data Features in Legitimate and Malware Applications



(a) Number of Downloads (b) Days in Google Play (c) Developer Reputation

FIGURE 4.2: Goodware/Malware boxplot comparison for three features: Number of downloads, number of days since the application was uploaded and developer reputation.

Some features will be more predictive than others, as noted in Fig. 4.2. In this sample figure, three boxplots for malware/goodware classes are shown for three features: the number of times the application has been downloaded from the market (see Fig. 4.2(a)), the time the application has been in Google Play (see Fig. 4.2(b)) and the developer reputation (see Fig. 4.2(c)).

The figure clearly elucidates that downloads will not be a very useful feature, as no clear separation can be found, being both goodware and malware in a similar 25-percentile (around 10) as well as 75-percentile (48), values (Fig. 4.2(a)). Concerning the number of days in Google Play in Fig. 4.2(b), quantiles suggest a longer stay of malware applications in the market, so malware apps could be indicated by those remaining longer unchanged. Finally, the reputation boxplot in Fig. 4.2(c) does show how developers generate either only malware or goodware, posing as a very nice feature for classification.

TABLE 4.2: Overview of all the meta-data fields collected and analyzed.

	Name	Description	Value
Intrinsic features			
1	size	Application size in bytes	Numeric
2	categoryName	Assigned Google Play Category	Categorical
3	ageInMarket	Number of days the app has been on Google Play	Numeric
4	lastSignatureUpdate	Number of days from last app signature update	Numeric
5	timeFromCreation	Number of days since the application was developed	Numeric
6	lastUpdate	Number of days since the application was last updated	Numeric
7	certVal	Number of days from which application is valid	Numeric
8	oldestDateFile	Number of days from the creation of the oldest file in the application	Numeric
9	numPerm	Total number of permissions required by the application	Numeric
10	numFiles	Total number of files the application contains	Numeric
11	numImages	Total number of images the application contains	Numeric
12	numDownloads	Total number of times the application has been uploaded	Numeric
13	versionCode	Google Play reported version of the application	Numeric
14	f+number features	Each of the different Feature hashes	Numeric
Social-related features			
15	totalVotes	Total number of rating votes given to the application	Numeric
16	OneStarRatingCont	Number of one-star votes received	Numeric
17	twoStarRatingCont	Number of two-star votes received	Numeric
18	threeStarRatingCont	Number of three-star votes received	Numeric
19	fourStarRatingCont	Number of four-star votes received	Numeric
20	fiveStarRatingCont	Number of five-star votes received	Numeric
21	meanStar	weighted average rating of the application	Numeric
Entity-related features			
22	developerRep	Developer reputation metric	Numeric
23	issuerRep	Issuer reputation metric	Numeric
Label			
L	isMalware	True if flagged by one or more AV engines	Boolean

In what follows of this section, feature selection and engineering analysis is performed over quantitative meta-data fields, aiming at understanding which variables are more suitable for malware detection.

4.3.1 Predictive Power of Permissions

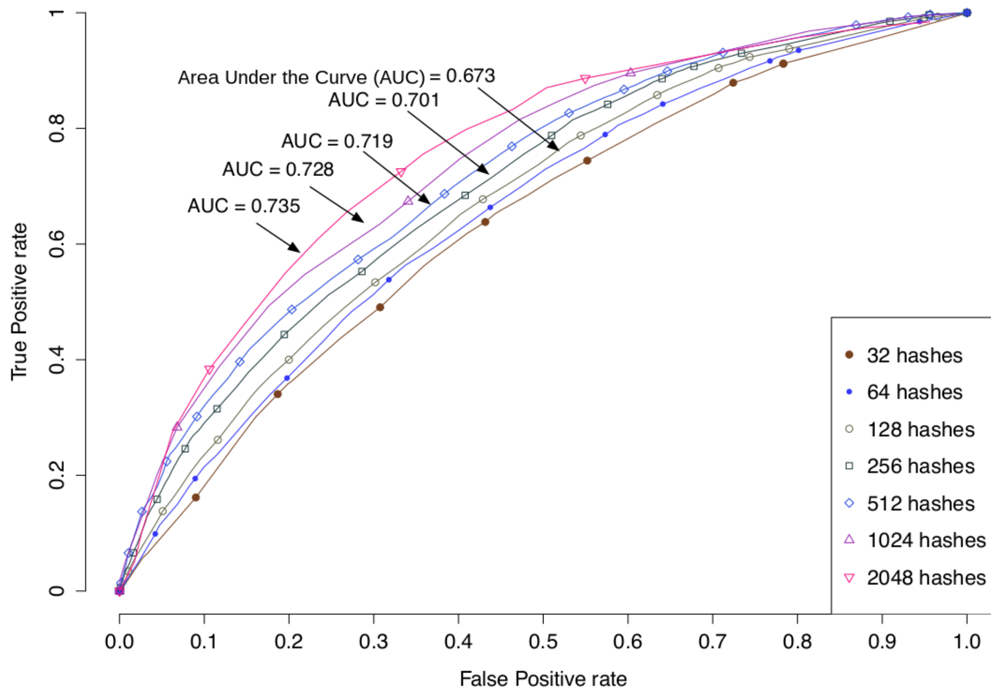


FIGURE 4.3: ROC curve for malware detection using different numbers of feature a hashes over permissions. For this experiment no other feature is considered.

Researchers have studied the permissions used by an application and their ability to detect malware in the past. For instance, the authors in [37] achieve F-score values in the range of 0.6 to 0.8. However, we have previously shown how permissions comprise a very sparse and high-dimensional matrix of up to 21K applications. *One-hot-encoding* vectorial representation of these applications is going to be sparse and have very little prediction power, so we use feature hashing to reduce dimensionality (for feature hashing reference the reader can recall Section 3.1).

In our experiments, we try different hashing spaces, namely 32, 64, 128, 256, 512, 1024 and 2048 hashes, running a 10-fold cross-validation process to correctly assess the feature amount vs performance trade-off. Using logistic regression, we try different threshold values and compute the *Area Under the Curve* (AuC) for each of the hashing spaces. Fig. 4.3 shows the ROC curves and the corresponding AuC values using LR for the different number of hashes for the (4-AV, 50%) dataset. As

observed, the more hash functions used, the higher AuC achieved, in the range of 0.7 (for 256 hashes and above).

Specifically, 512 hashes shows a good trade-off between model performance and the total number of features in the figure. For this specific case, average F-score is 0.675 whereas the area under PR curve (AuPR) is 0.685 in the LR case. For Random Forest algorithm, the model achieves 0.653 and for SVM 0.659, both in terms of F-score. In any case, experiments show that permissions alone are a good indicator for malware.

4.3.2 Feature Importance and Selection: A Machine Learning Approach

Beginning at 512 permissions hashes, 35 Android-defined categories and the remaining 22 features, variable selection is performed to evaluate and rank features attending to their predictive power. In addition, this feature selection sheds light on meta-data features and how it differs with malware applications. This section covers two of the main types of feature selection methods: *wrapper* and *filter* methods (details are available in Section 3.2).

Wrapper Methods: Step-AIC

Step-AIC performs feature selection by measuring variable importance using the *Akaike Information Criteria* over a subset of features that are incremented with a different new variable at each step. Step-AIC is used to obtain the 8 most relevant features over a random sample of applications, considering as malware any application that accounts at least a single detection.

TABLE 4.3: This table summarizes the parameter of the model computed by the step-AIC algorithm

Features	Estimate	z-score	p-value
developerRep	7.34	9.26	$1.93\text{x}10^{-20}$
timeFromCreation	$-4.48\text{x}10^{-02}$	-7.93	$2.05\text{x}10^{-15}$
numPerm	$1.14\text{x}10^{-01}$	4.28	$1.79\text{x}10^{-5}$
issuerRep	3.08	3.68	$2.24\text{x}10^{-4}$
cat.LIFESTYLE	$-8.35\text{x}10^{-01}$	-2.68	$7.27\text{x}10^{-3}$
ageInMarket	$5.40\text{x}10^{-04}$	1.67	$9.42\text{x}10^{-2}$
oneStarRatingCont	$3.71\text{x}10^{-05}$	1.65	$9.69\text{x}10^{-2}$
cat.OTHER	$-4.55\text{x}10^{-01}$	-1.59	$1.09\text{x}10^{-1}$

Table 4.3 depicts such top-8 features sorted by *p-value*, which indicates how likely any feature is to be predictive by chance, being smaller *p-values* indicators of high chance of causal relation.

Overall, entity related features, involving developer reputation (developerRep) and certificate-issuer reputation (issuerRep) are the most relevant and, thus, the

most predictive ones. Anyway, developerRep has by far the smallest *p-value*, that makes it the best possible malware predictor by itself.

Other interesting features involve the number of days in market (ageInMarket and uploadDate), that show really small *p-values*. The amount of permissions each application requires also ranks high within the resulting features. Interestingly, only one star ratings are relevant for the step-AIC model, together with two specific categories: LIFESTYLE (cat.LIFESTYLE) and OTHER (cat.OTHER).

Step-AIC achieves almost 0.59 F-score in training and 0.39 in testing, which is a low score, mainly driven by the small sample in the experiment. In fact, step-AIC is a very demanding and greedy algorithm that makes difficult extending the experiment over the entire quantitative dataset. For this same reason, permission hashes were not considered for this experiment. Next sections will perform other lightweight feature selection approaches to the entire dataset.

Filter Methods: Relevance Metrics

Throughout this section we measure different importance metrics, namely *Pearson's Chi Squared*, the *Gain Ratio*, the *Information Gain* and the *Mean Decrease in Node Impurity* over each feature in the dataset and rank them to assess their performance.

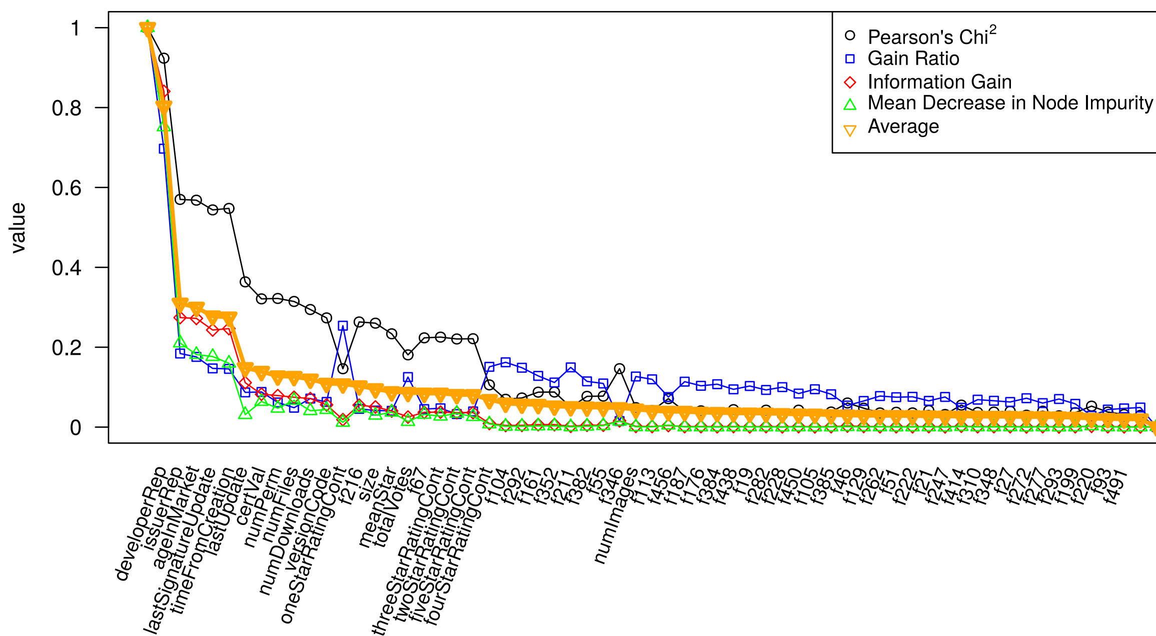


FIGURE 4.4: This figure shows the different features sorted by average score on the four studied filter methods, namely: Pearson Chi Squared, Gain Ratio, Information Gain and Gini index.

Fig. 4.4 illustrates such scores, sorted by average. All metrics are normalized with respect to the maximum value and computed over the (4-AV, 50%) dataset. Again, it can be observed that *developerRep* feature is the most important feature in the dataset in terms of all four metrics and followed closely by *issuerRep*.

It is worth noting that these two reputations achieve nearly 0.8 on their own, so they could be selected as the only detection features. Despite, considering other features would be useful to avoid missing detections if the malware developer or issuer account is new and has no reputation value associated.

The next features in terms of importance are related to dates, namely *ageInMarket*, *lastSignatureUpdate*, *timeForCreation*, *lastUpdate* and *certVal*, suggesting that application time periods are different for malware and goodware developers. Additionally, application element counts (*numPerms*, *numFiles*, *numDownloads*) present small, though slightly relevant importance to distinghish between malware and goodware.

Finally, permission hashes and social ratings show very little influence in detecting malware. It is worth noting that the scores presented have been scaled by the maximum (*developerRep*) and, thus, the results only imply there are better variables to detect malware other than social related features and permissions.

4.4 Malware Detection Model

4.4.1 Determining Model Size

Model size is quite an important decision mainly due to the costs associated for training, storing and managing larger models, not to mention the well-known *Curse of Dimensionality*. On the other side, too small models can be biased and affected by subtle changes in meta-data, damaging performance and its detection abilities. In sum, choosing the right size for the model is required as a robustness policy that works even against feature alteration, corruption or absence.

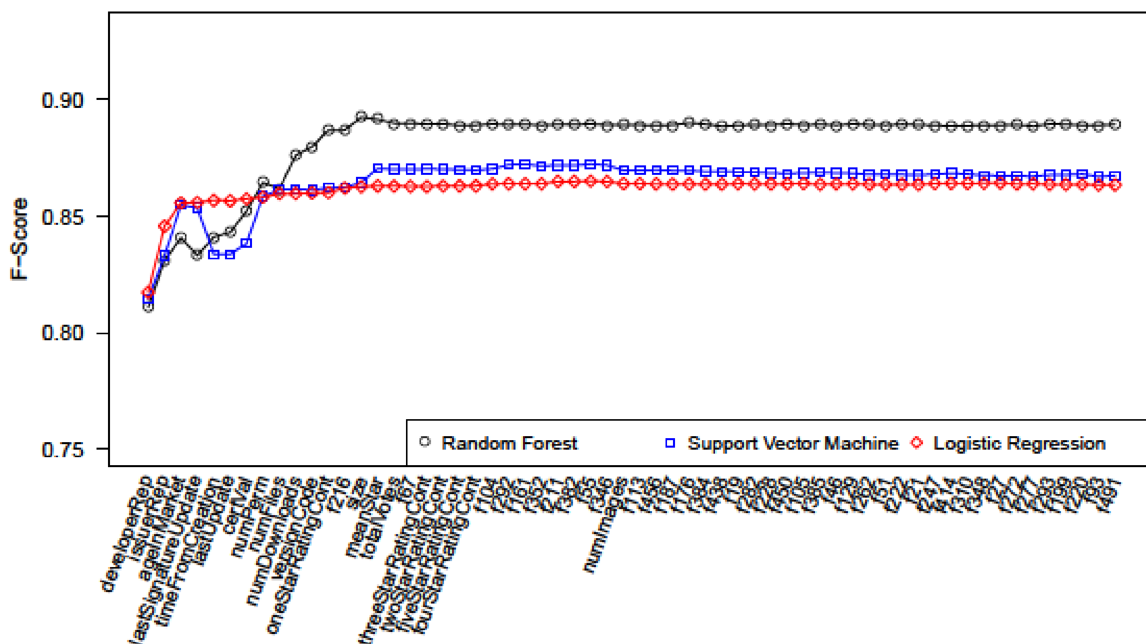


FIGURE 4.5: F-score values for different methods and feature sets. Variables are sorted in order of addition to the models.

To better adjust models, we perform an experiment that consists on running tenfold cross-validation to compute the F-score values of models built step by step incrementing one variable at a time. The new variable is selected by ranking features following the ordering in Fig. 4.4. For this experiment we use SVM, RF and LR classification algorithms.

Fig. 4.5 shows the results of such experiment, where RF is the clear winner with an F-score value of nearly 0.89 compared to 0.86 of LR and 0.87 of SVM. In general, all three models show that few more variables can improve the model more than reputations, showing a flat curve from approximately 15 features onwards.

This trend does show that the increase of the feature space and complexity beyond 15 features is useless, as no further gain is achieved. In fact, in the case of LR, this threshold is reached at 6 variables. Accordingly, we set the number of features in the model to be the top 15 according to Fig. 4.4 ranking.

4.4.2 Solution Modeling and Results

Models are trained and tested over the benchmark datasets, using only their top-15 features, namely: *developerRep*, *issuerRep*, *ageInMarket*, *lastSignatureUpdate*, *timeForCreation*, *lastUpdate*, *certVal*, *numPerm*, *numFiles*, *numDownloads*, *versionCode*, *oneStarRatingCont*, *f216*, *size* and *meanStar*. Table 4.4 shows the training/test F-score, precision and recall values for each dataset and ML model.

Indeed, table results show that RF performs slightly better than the rest while the overall performance increases when the amount of malware in the set increases, showing the best results when malware and goodware are balanced. Actually, in most 2%-malware cases, the difference between train and test error suggests generalized overfitting.

Furthermore, the algorithms perform best at identifying those malware applications tagged by several AV engines, which suggests that either low detection count malware samples are not always malware or that the meta-data is more predictive when more AV engines agree on their detection. In addition, the permissions-only approach is clearly inferior to this system, that does not use almost any permission hash.

4.4.3 Robustness of the Models

One of the potential weaknesses of this approach is the possibility that issuers and developers use new accounts to reduce their reputation and avoid detection. However, reputations are not the only good features for detection and, therefore, the proposed model could detect malware samples coming from a good reputation account since the classifiers are built using more than these two variables.

Indeed, the proposed 15 features represent a very good trade-off, being more unnecessary, as demonstrated by the flat behavior of Fig. 4.4. Despite, no less features should be selected either, since many of them contribute to the redundancy required to make the system robust. This is very important, specially for cases when

TABLE 4.4: Performance metrics of the algorithms for each dataset in the benchmark.

Logistic Regression (train/test)				
Malware	NDet	F-score	Precision	Recall
2%	1	0.82/0.1	0.76/0.06	0.89/0.24
25%	1	0.89/0.57	0.87/0.46	0.91/0.73
50%	1	0.93/0.79	0.94/0.82	0.93/0.77
2%	2	0.8/0.18	0.74/0.12	0.88/0.33
25%	2	0.9/0.68	0.89/0.59	0.9/0.79
50%	2	0.94/0.82	0.95/0.78	0.93/0.86
2%	4	0.81/0.27	0.75/0.19	0.89/0.47
25%	4	0.91/0.73	0.9/0.65	0.91/0.83
50%	4	0.95/0.84	0.97/0.79	0.94/0.89
Suport Vector Machine (train/test)				
Malware	NDet	F-score	Precision	Recall
2%	1	0.85/0.08	0.76/0.05	0.96/0.23
25%	1	0.93/0.68	0.92/0.69	0.93/0.67
50%	1	0.96/0.82	0.96/0.87	0.95/0.77
2%	2	0.82/0.16	0.72/0.1	0.95/0.35
25%	2	0.93/0.73	0.93/0.7	0.93/0.76
50%	2	0.96/0.84	0.97/0.9	0.94/0.8
2%	4	0.81/0.26	0.7/0.17	0.97/0.54
25%	4	0.94/0.77	0.94/0.72	0.93/0.83
50%	4	0.96/0.87	0.98/0.89	0.95/0.84
Random Forest (train/test)				
Malware	NDet	F-score	Precision	Recall
2%	1	0.99/0.12	0.99/0.07	0.99/0.33
25%	1	0.99/0.73	0.99/0.7	0.99/0.77
50%	1	0.99/0.84	0.99/0.88	0.99/0.8
2%	2	0.99/0.22	0.99/0.15	0.99/0.46
25%	2	0.99/0.77	0.99/0.73	0.99/0.83
50%	2	0.99/0.87	0.99/0.89	0.99/0.86
2%	4	0.99/0.32	0.99/0.22	0.99/0.59
25%	4	0.99/0.81	0.99/0.76	0.99/0.87
50%	4	0.99/0.89	0.99/0.88	0.99/0.9

some features are corrupted or unavailable, like for instance when the developer has changed accounts.

For further illustration, Table 4.5 shows the F-score results of the RF algorithm over different subsets of features, selected by a *sliding window* procedure. The first column shows the same train/test F-score values as in Table 4.4 since both use the same top-15 features. The second column shows the values for features from 3 to 17 in Fig. 4.4, which represent top-17 features without developerRep and issuerRep. In this case and the following ones (5-19, 7-21 and 9-23) F-scores are worse than before, but still reach an acceptable performance that drops once top-7 features are not used.

TABLE 4.5: Random Forest F-score for different feature subsets based on an incremental sliding window.

F-score Random Forest (train/test)					
NDet	1-15 feats.	3-17 feats.	5-19 feats	7-21 feats	9-23 feats
1 AV	0.99/0.84	0.99/0.86	0.99/0.84	0.99/0.74	0.96/0.72
2 AV	0.99/0.87	0.99/0.87	0.99/0.86	0.99/0.79	0.96/0.75
4 AV	0.99/0.89	0.99/0.88	0.99/0.87	0.99/0.80	0.96/0.77

4.4.4 Performance and Computational Time

In the light of the obtained results, the proposed solution could be implemented as an early detection system that rapidly inspects meta-data upon application upload to any market. In this case, the time consumed by the system and specifically the delays introduced are important, specially regarding the impact on market submission systems.

To assess such impact, we conducted an experiment measuring train, test and prediction times over an Intel Xeon E5-2630 server with 24 cores and 190 GB of RAM memory. Results are displayed below and suggest that this system would be fast and scalable:

- **Logistic Regression**:
  - *Prediction time*: 0.1 μs
  - *Train and test time*: 46 ms
  - *Train* with Hyper-parameter tuning: 2 m 6 s
- **Support Vector Machines**:
  - *Prediction time*: 16 ms
  - *Train and test time*: 2.4 s
  - *Train* with Hyper-parameter tuning: 8h 20m
- **Random Forest**:

- *Prediction time*: 32 μs
- *Train and test time*: 3.5 s
- *Train* with Hyper-parameter tuning: 47 m 32 s

Prediction times are fast for all models, even though SVMs are slightly slower. Moreover, training times enable daily model computation, which would be enough for model validity. As a result, the algorithm proposed in this section, that relies on meta-data to detect malware is fast and scalable and could be applied satisfactorily for market-scale malware detection.

4.5 Summary and Conclusions

For most developers, meta-data is important, as it presents application functionalities and features inside any Android market, where meta-data appears naturally connected to their apps in order to give transparency to potential users.

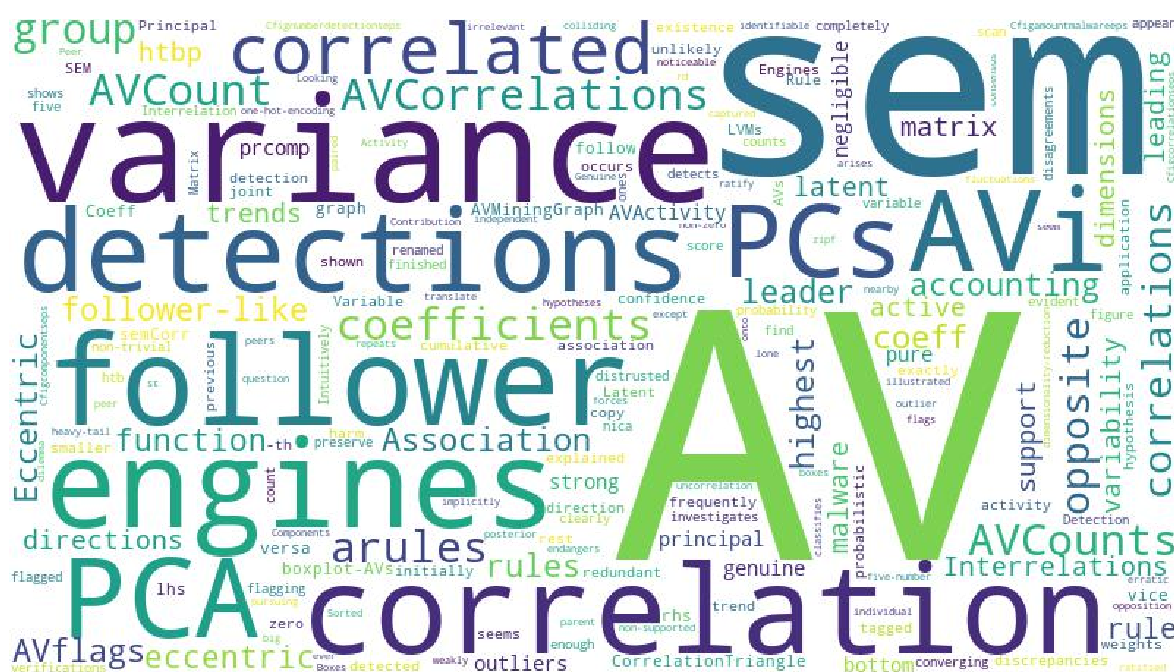
Through this chapter, we have analyzed meta-data as a malware indicator and applied it for malware detection. Early in Section 4.3, we have focused on the analysis of quantitative meta-data features to understand which ones are more related to malware itself and how each of them behaves in malware and gooware. In fact, it has been proved that developer and certificate issuer reputation are the most important features separating what is malware and what is not.

Then, Section 4.4 conducts an extensive experiment involving a collection of benchmark datasets with different malware configurations and machine learning algorithms. The aim has been to study the viability of using meta-data in a large-scale malware detection system using detections from important multi-scanner vendors as ground truth. Results have shown that such system is possible, robust, fast and easily scalable. Moreover, permissions have been demonstrated not to be the most efficient predictor.

In conclusion, this chapter has demonstrated and analyzed how meta-data can be used for malware detection in the context of Android applications. Relying on malware detections from different AV engines, we have proposed a ML system that identifies malware with an 89% F-score using solely meta-data and can work fast at market scale.

Chapter 5

Data-driven Interrelation Analysis of AV engines



In the last chapter (Chapter 4), we performed ML-powered malware detection using as ground truth a large list of multi-scanner detections. Solving such problem unveiled that multi-scanner labels offer distrusted detection information due to the disagreements among engines on what is malware and what is not. Indeed, one of the most interesting and conflicting observations in the previous chapter is that no single application is flagged as malware by every single AV engine, accounting the mostly detected application for just 53 detections out of a total of 61. In fact, the average number of detections in our application collection is 3.135.

In this light, this chapter investigates such discrepancies and relations of AV engines from a data analytics perspective. Intuitively, AV engines should eventually agree on evident malware applications, even though our observations show different behavior patterns not converging to this assumption. In this chapter, we leverage different data analysis tools to understand these behaviors and identify AV trends.

Recall that whenever a malware engine detects a malware application, it returns a detection signature containing some additional data, like last scan date or malware class identifier. Since such detections are not always consistent, they become controversial and makes the identification of malware a non-trivial task. For completeness, we will include in the analysis any sample that has been tagged by at least one AV engine.

For this chapter, we use an application collection containing 82,866 multi-scanner outputs coming from suspicious applications and provided by the same tool as in previous Chapter 4: Tacyt (developed by Telefónica). In contrast to the previous chapter, only AV engine detection reports accounting for a total of 259,608 positive detections with an average of 3.13 detections per application are considered. To preserve privacy, AV engines have been renamed to AV1 to AV61 consistently along the thesis.

First, we begin by describing the dataset and the process to extract its *Detection matrix* along with some insights in Section 5.1. Afterwards, in section 5.2, we describe our initial hypothesis of *follower* and *eccentric* engines and look for verification through the analysis of the detection matrix using PCA, correlations and rule mining. Finally, we present in Section 5.3 a statistical approach based on latent variable models to estimate the risk of a sample being malware. The chapter is finished in Section 5.4 providing the most relevant conclusions.

5.1 Detection Matrix and Dataset Insights

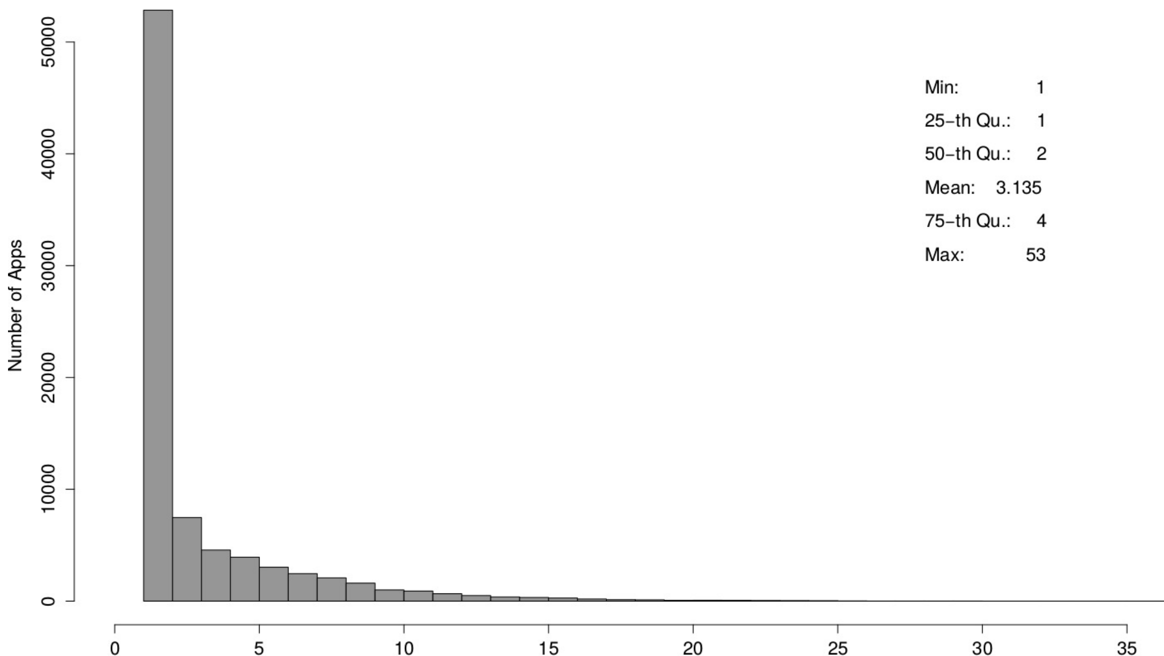


FIGURE 5.1: Number of AV flags per application

Let A be an $82,866 \times 61$ matrix which elements $A_{ij} \in \{0,1\}$ are equal to 1 if the i_{th} app is considered malware by the j_{th} engine and 0 otherwise. This matrix is called the *Detection Matrix* of the collection and indicates which AVs have tagged each application. Each of the rows in the detection matrix A is called the *Detection Vector* of each application sample and is a *one-hot-encoding* vector of all the detections any application sample may have.

The detection matrix is very sparse with roughly 5% non-zero entries. On average, each application is detected by 3.135 ± 3.46 engines, showing enormous fluctuations that depend on each specific application. Above all, six engines (AV27, AV58, AV7, AV2, AV30 and AV32) are the most active ones, with detection counts above $10,000$ applications.

Fig. 5.1 represents the the amount of applications for different numbers of detections in matrix A. The data clearly follows a heavy-tail zipf distribution where most malware applications are detected by a small number of AV engines and larger counts rare. Indeed, single-detection applications account for a total of $38,933$ (46.9%), almost half of the application collection. Interestingly, the lack of consensus is so prominent that no single application has been detected by all the 61 engines, being the highest detection count at 53 engines.

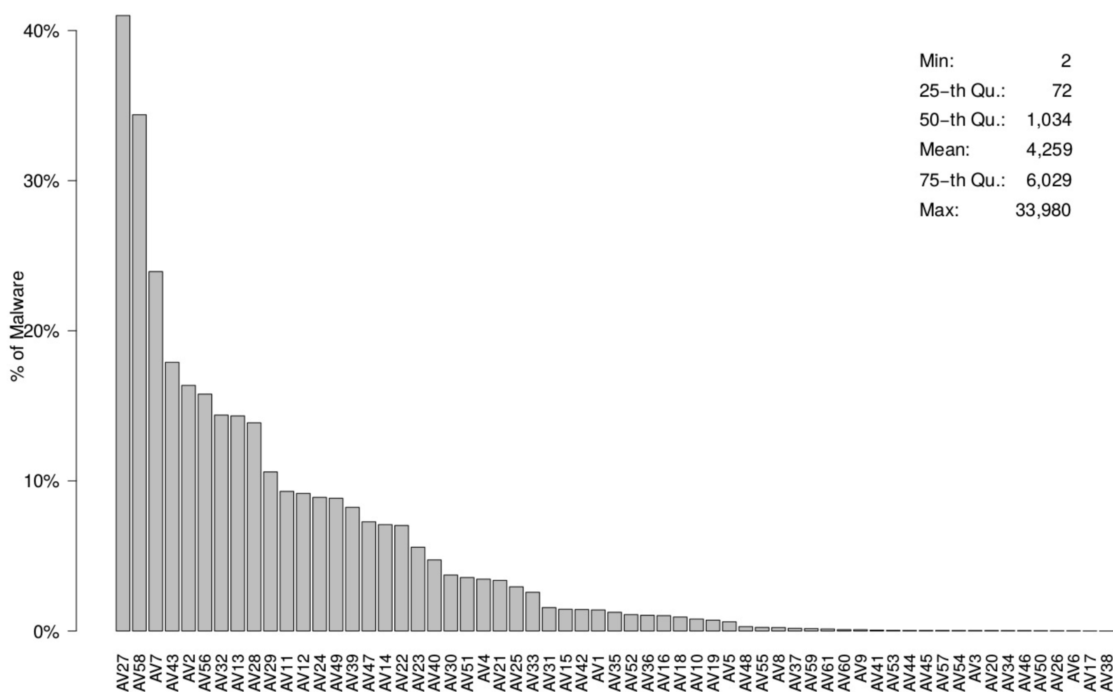


FIGURE 5.2: Activity of AV engines: Sorted AV engines by their individual detection rate

In addition, Fig. 5.2 shows the AV activity histogram, where the number of detections per engine is illustrated in relative terms. There, the most active engine (AV27) performs more than $33,975$ exposures, which is 40% of the dataset. At the same time, other engines are less active and do not reach even 50 application detections each (AV53 to AV38). The figure displays as well the five-number summary of the number of uncoverings per engine.

These figures raise a big dilemma, specially to malware analysts: While one or two engines are detecting malware in a sample, there are as many as 59-60 that find no harm in the analyzed application, which endangers the trustworthiness of the prior positive detections. In fact, any detection could be considered enough to mark a sample as malware, but in that case, the question arises: *Why do the rest of engines do not find any harm in these samples?* To solve this, Section 5.2 will initially review this insight and will analyze AV patterns and behaviors.

Furthermore, we assume two types of detections based on the previous observations: *genuine* and *supported* detections. *Genuine* detections correspond to lone detections, those where an AV engine classifies something as malware on its own. On the contrary, *supported* detections are those where some sort of agreement between engines is shown.

5.2 Peer Relations among AV Engines

5.2.1 Followers and Eccentrics

AV engines perform detections according to certain policies and parameters designed by their parent companies. Such policies are aimed to improve their detection rates and better prevent malware in AV client devices. In spite of aiming towards the common objective of fast and efficient detection of malware applications, different engines follow different policies to target malware that usually translate in different detection patterns.

Indeed, we identify two potential groups of AV engines that show completely opposite behaviors: *Followers* and *Eccentrics*. Such groups aggregate the two main trends we have discussed above:

- **Follower engines** usually perform detections paired with other engines, meaning that samples detected by one follower engine, which would be a *leading engine*, will easily have exposures from other followers as well. This pattern implicitly gives peer support to follower engines: AV detections that are supported or ratified by other engines, even within a group, are more likely to be indicating malware more accurately than non-supported detections. Following the previous debate on engine patterns, the engines in this group would mainly perform *supported* detections.
- **Eccentric engines** do exactly the opposite of followers. They show erratic patterns that no other engine is supporting, hardly ever colliding with others in their detections. The behavior in this group matches to those engines performing *genuine* detections.

These concepts are theoretical and have not been demonstrated yet. However, through the rest of this section we will try to find traces of these patterns and ratify the previous hypotheses from a data analysis perspective.

5.2.2 Principal Component Analysis

PCA seeks for the highest variance direction within the dataset and projects all data onto such direction. Then, it repeats the process with orthogonal directions for as many directions as features within the dataset. These directions or linear combinations of features are called Principal Components (PCs), and they typically summarize the contributions of different variables to the dataset variance (See Section 3.2 for reference).

Using PCA, it is possible to determine the amount of variability in engine detection, as the smaller the variability, the easier for PCA to reduce dimensions and therefore remove AV engines, which in turn points to irrelevant or redundant engines.

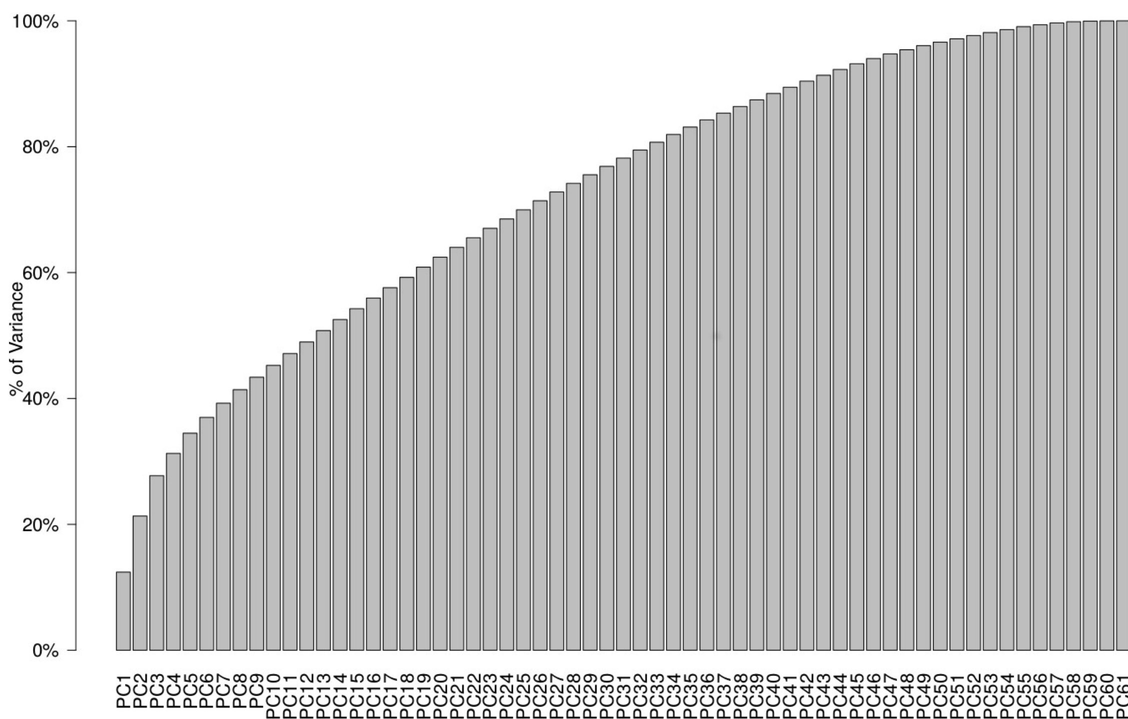


FIGURE 5.3: Contribution of principal components to variance. The variance depicted is cumulative, so posterior components include previous contributions.

Fig. 5.3 shows the percentage of cumulative variance captured by each one of the principal components. As shown, 50% of the variance in the dataset is explained with the first 13 PCs, 75% of the variance is explained with the first 29 PCs whilst reaching 95% of variance explainability requires up to 48 PCs.

As a result, engines seem not to be so alike, which forces a dimensionality-reduction algorithm such as PCA to compress 61 dimensions into 48 potential ones. This demonstrates just the contrary of expectations: AV engines are not redundant, as they do not detect the same applications and, therefore, there are not many engines providing the same detection information that could be potentially removed.

5.2.3 Correlation between Engines

Since the goal of all AV engines is aligned, AV detections should be highly correlated, specially in those cases of real malware. Nonetheless, correlation analysis manifests the exact opposite: the correlation matrix of the detection matrix A shows that AV engines are weakly correlated in general terms, except for some small subsets of engines, which potentially indicate follower-like behaviors.

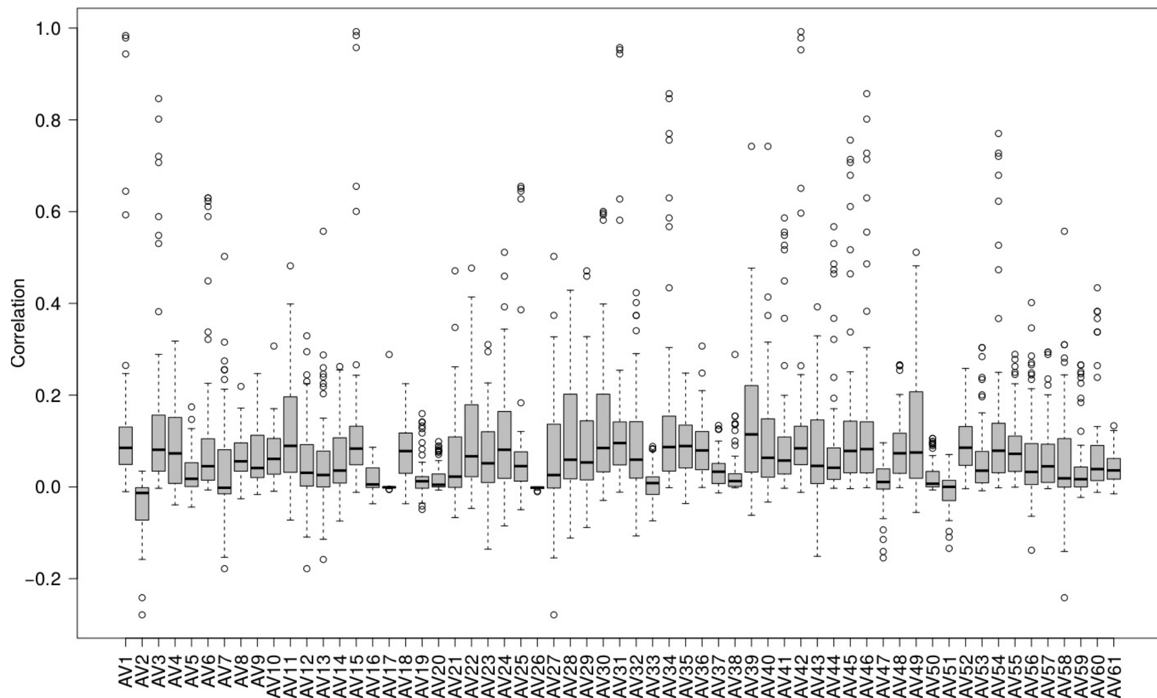


FIGURE 5.4: Pairwise correlations of all AV engines. Boxes represent standard 1st, 2nd and 3rd quantiles.

Fig. 5.4 depicts the correlation boxplots for each AV engine with respect to all their peers, which contains most boxes nearby zero, indicating weak or negligible correlation values albeit there is a significant number of outlier correlation pairs. Such outliers appear strongly correlated with others, reaching values over 0.8, which indicates very similar detection patterns.

Looking at the groups of correlated AVs, the most relevant one contains AV1, AV15, AV31 and AV42 with a very strong correlation one another. Furthermore, other AVs that show noticeable correlations as well are AV3, AV45, AV39 or AV11.

In opposition, other engines retain no correlation to any other, not even negative. These seem to perform detections completely independent of the rest, like AV17 or AV26. In Fig. 5.4 they are easily identifiable through their flat boxplot around zero and almost no variability. The most obvious ones are AV17 and AV26, which indicate a purely eccentric behavior, but some others, like AV2, AV38 or AV19 show very small correlation value distributions as well, with values relatively close to zero.

Additionally, Fig. 5.5 depicts the most relevant engines for the correlation analysis on a one to one basis. It is worth noting the large correlation evinced in all cases

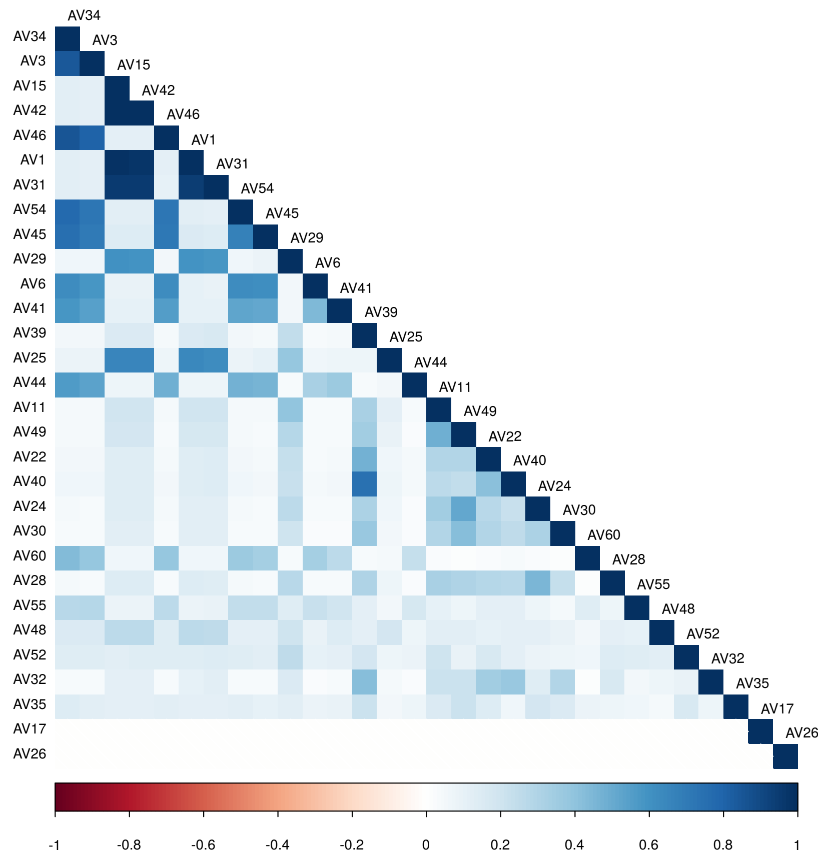


FIGURE 5.5: Most relevant correlations between AV engines: The top ones are the most correlated ones while the two bottom ones are the pure eccentrics

among AV1, AV15, AV31 and AV42 as well as the strong correlations between AV3 and AV34. Other AVs, such as AV46, AV54 or AV45 are also highly correlated to the latter group. Finally, AV17 and AV26 at the bottom of the graph show themselves as pure eccentrics with a correlation close to zero with the rest.

5.2.4 Association Rule Learning

Association Rule Mining can help identifying AV engines that frequently appear together revealing whether they are followers or *leaders*, that is, follower-like engines that lead the detections (for reference regarding rule mining, see Section 3.1). We leverage the R package *arules* [201] for this experiment.

Indeed, Table 5.1 shows a list of the most relevant association rules sorted by support. For instance, rule no. 1 $\{AV58\} \rightarrow \{AV27\}$ reveals that, when AV58 detects a malware application (which occurs in 21.3% of the cases), then 62% of the times AV27 tags the same application as malware too. The opposite occurs in 52% of the cases as indicated in rule no. 2.

Support values are not so high, as they correspond to the joint frequency of all

TABLE 5.1: This table shows the 40 most relevant association rules for AV engines.

No	lhs	rhs	support	conf.	No	lhs	rhs	support	conf.
1	AV58	AV27	0.213	0.620	21	AV58	AV56	0.096	0.280
2	AV27	AV58	0.213	0.520	22	AV28,AV58	AV27	0.087	0.884
3	AV7	AV27	0.203	0.850	23	AV27,AV28	AV58	0.087	0.801
4	AV27	AV7	0.203	0.496	24	AV27,AV58	AV28	0.087	0.408
5	AV32	AV27	0.123	0.858	25	AV32	AV58	0.086	0.599
6	AV27	AV32	0.123	0.301	26	AV58	AV32	0.086	0.250
7	AV43	AV27	0.118	0.660	27	AV43	AV58	0.083	0.467
8	AV27	AV43	0.118	0.288	28	AV56,AV58	AV27	0.082	0.854
9	AV7	AV58	0.117	0.490	29	AV27,AV56	AV58	0.082	0.733
10	AV58	AV7	0.117	0.341	30	AV27,AV58	AV56	0.082	0.386
11	AV56	AV27	0.112	0.712	31	AV49	AV27	0.081	0.926
12	AV27	AV56	0.112	0.274	32	AV32,AV58	AV27	0.077	0.895
13	AV28	AV27	0.108	0.784	33	AV27,AV32	AV58	0.077	0.625
14	AV27	AV28	0.108	0.265	34	AV27,AV58	AV32	0.077	0.361
15	AV58,AV7	AV27	0.106	0.908	35	AV32	AV7	0.075	0.525
16	AV27,AV7	AV58	0.106	0.523	36	AV7	AV32	0.075	0.315
17	AV27,AV58	AV7	0.106	0.499	37	AV56	AV7	0.074	0.470
18	AV28	AV58	0.098	0.710	38	AV7	AV56	0.074	0.309
19	AV58	AV28	0.098	0.286	39	AV56	AV32	0.074	0.469
20	AV56	AV58	0.096	0.611	40	AV32	AV56	0.074	0.515

AV engines performing the same detection, which we have shown to be low in this dataset. In fact, top joint detections account for roughly over 20% of the samples in the dataset. Nevertheless, confidence is high, specially in top rules, which suggests there are several engines showing agreements in a large amount of detections.

Actually, Fig. 5.6 illustrates the graph obtained when using the support of the obtained rules as adjacency matrix. The graph shows two clearly defined groups: The one formed by AV1, AV15, AV25, AV31 and AV42 and the other one containing the rest. The smaller group seems to comprise highly correlated AV engines that also appear frequently enough together in the detections. They are related with all the rest in the group, but it is clear in the graph that they they follow a slightly different trend with respect to them. On the other side, the larger group seems to be a bunch of AV engines that detect altogether much more frequently and that follows in some way the engines in the previous group.

This clearly shows that the follower group is a broad scope group that includes both *leader* and *follower* engines. Indeed, this graph strengthens the previous hypothesis whilst indicating the existence of an additional smaller group that could be leading detections. The other engines appear pure followers, that is, those who look for some trendy engine to copy, possibly in the leader group. Furthermore, the extracted rules completely lack those engines identified as eccentrics in Fig 5.4 according to their correlations.

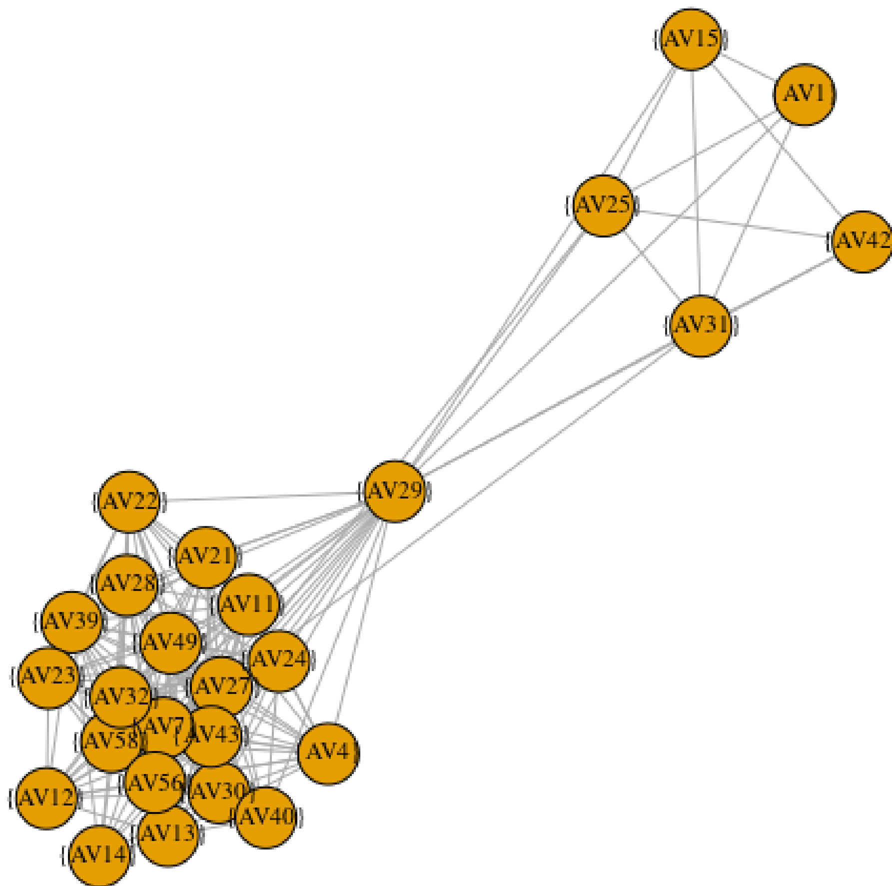


FIGURE 5.6: AV engine follower graph computed from the rules computed by the apriori algorithm. The support of rules is used as adjacency matrix of the graph.

5.3 Latent Variable Modeling for Malware Risk Assessment

So far, we have exposed different AV behaviors when detecting malware. Specifically, we have found three types of AV engines attending to their detection patterns: Leaders, followers and eccentrics. Although some AV engines are correlated, most of them are not and therefore, there are severe discrepancies to determine engine to be trusted in malware decisions. Whenever a malware sample is detected by a majority of engines it should be considered malware; despite, such situation seems unlikely in the light of previous Fig. 5.1 that associated the majority of samples with one or two detections at most.

In the literature, any predetermined number of AV detections is considered a

good enough criterion to assign the malware label, such as five or more. Nonetheless, the problem is that even though five engines are saying that a sample is malware, the remaining 56 engines are stating it is not. Besides, depending on the specific five AVs triggering detection, the confidence on the final decision could be affected, as we have shown many AV engines just follow others.

In this light, this section focuses on defining a new metric to provide a *malwarish* score to each application, considering the underlying relations of engines and their disagreements. To approach this metric, we leverage the well-known Latent Variable Models (LVMs) to assess actual malware risk. There are many probabilistic methodologies to address LVMs and we have selected *Structural Equation Models* (SEM) given their simplicity and extended use. SEMs model relations assuming a linear relation where the latent variable is the outcome:

$$SEM = \sum_{i=1}^{61} \alpha_i \times X_{AVi} \tag{5.1}$$

where the observed variables denoting the AV votes on an application (X_{AVi}) contribute to the latent Z_{sem} malware variable; coefficients α_i are fitted by the algorithm using the available data. The R library *lavaan* [202] has been used to fit the coefficients and function parameters. The results of this experiment are shown in Table 5.2.

TABLE 5.2: SEM Coefficients for each AV engine in the dataset.

X_{AVi}	α_i	X_{AVi}	Coeff. α_i	X_{AVi}	Coeff. α_i
AV1	1.000	AV22	0.297	AV43	0.353
AV2	0.039	AV23	0.119	AV44	0.012
AV3	0.017	AV24	0.336	AV45	0.024
AV4	0.284	AV25	0.956	AV46	0.016
AV5	0.035	AV26	0.000	AV47	0.203
AV6	0.010	AV27	0.287	AV48	0.125
AV7	0.241	AV28	0.389	AV49	0.446
AV8	0.023	AV29	0.387	AV50	0.002
AV9	0.062	AV30	0.984	AV51	0.003
AV10	0.067	AV31	1.025	AV52	0.118
AV11	0.498	AV32	0.341	AV53	0.007
AV12	0.078	AV33	0.030	AV54	0.019
AV13	0.065	AV34	0.017	AV55	0.036
AV14	0.175	AV35	0.110	AV56	0.201
AV15	1.029	AV36	0.107	AV57	0.008
AV16	0.064	AV37	0.017	AV58	0.205
AV17	0.000	AV38	0.001	AV59	0.013
AV18	0.064	AV39	0.367	AV60	0.018
AV19	0.009	AV40	0.256	AV61	0.017
AV20	0.001	AV41	0.023		
AV21	0.172	AV42	1.018		

In the table, coefficients range between 0.001 to 1.029. Interestingly, eccentric

engines match those engines the model assigns negligible weights to, follower engines are scored with weights in the range 0.15 to 0.4 and all the highest weights are assigned exactly to the set of leader engines discovered above (AV1, AV15, AV25, AV30, AV31 and AV42).

Additionally, those engines with smaller detection counts (below 100 samples tagged) get smaller α coefficient values, as they are lacking too many detections to be considered as important as other more active engines.

To better estimate malware probability, we constrain Z_{sem} score between 0 and 1 into a probabilistic value by using the well known logistic (aka logit) function:

$$Z_{sem} = \frac{p}{1-p} \quad \Rightarrow \quad p = \frac{e^{Z_{sem}}}{1+e^{Z_{sem}}} \tag{5.2}$$

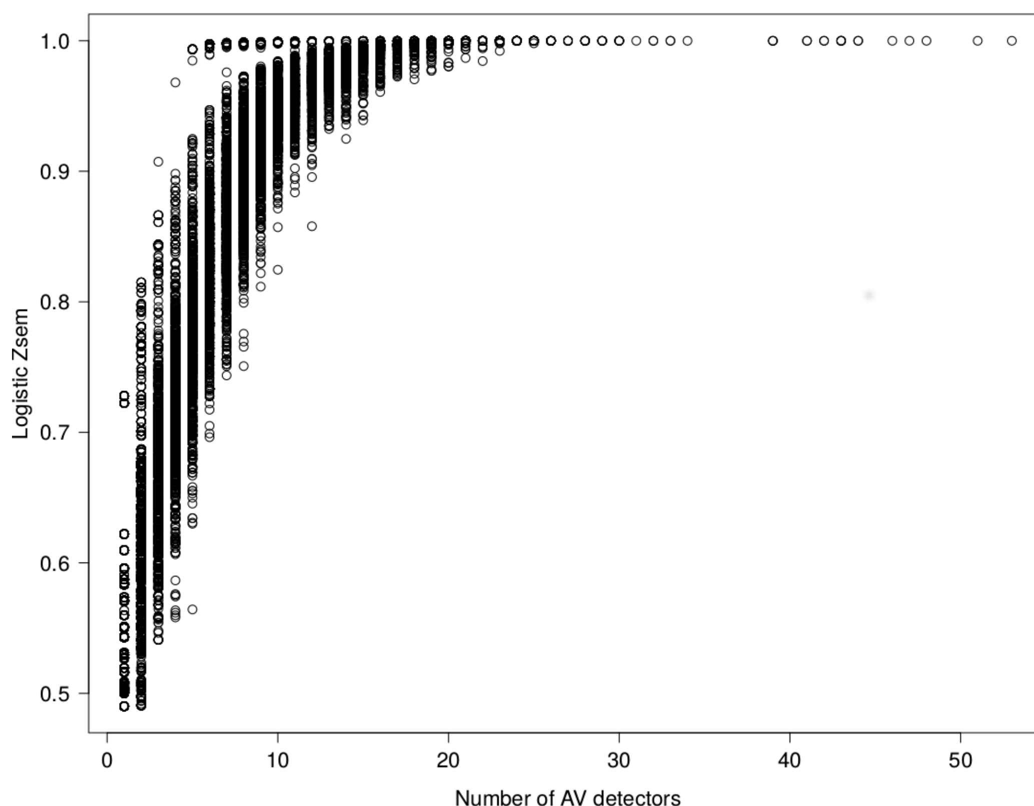


FIGURE 5.7: Logistic Z_{sem} as opposed to AVCount.

Fig. 5.7 shows a scatter plot of Z_{sem} values against the number of AVs that tagged each apk as malware (AV count) for the total 82K apps. Clearly, applications identified as malware by many AVs also typically score high in Z_{sem} and vice versa, showing 0.92 correlation between AV count and Z_{sem}. Indeed, for AVCounts larger than 10, the probability of true malware is greater than 0.8. While majority voting may not be the best reference to compare to is still a good proxy, since is the only available and should be correlated: the more AV engines detect a threat, the more likely such threat is real.

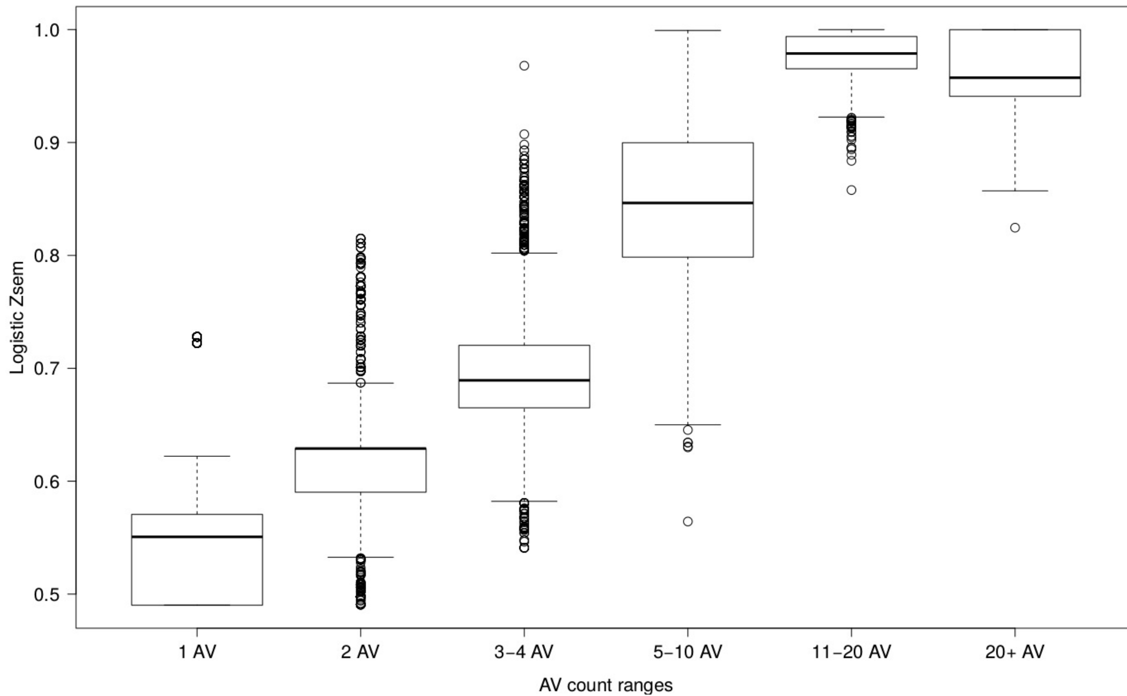


FIGURE 5.8: Boxplot of Z_{sem} scores for aggregated detection counts

Similarly, Fig. 5.8 shows Z_{SEM} as a function of constrained AVCounts, showing the trend for each AVCount or group of them. Actually, the figure clearly shows an increasing trend where probability values are likely to be incremented as the AVCount is higher. However, we can observe that even in cases of large AVCount, the score can be low and vice versa, as noted by the outliers shown in both parts of the figure.

As an example, consider application number 2,257, that has 73% probability of being malware and has been spotted by a single powerful engine: AV30. This application scores higher than almost everyone accounting for two detections and many than those with three-four detections. Oppositely, application 69,284 has a similar Z_{sem} value of 73% but has been flagged by three less powerful AV engines, namely: AV49, AV24 and AV56 (see Table 5.2).

5.4 Summary and Conclusions

Through the inspection of malware detections performed by different engines over a collection for 82,866 application samples, this chapter has investigated the different types and inter-relations of AV engines using the detection matrix A. Indeed, we initially advanced the existence of *follower* and *eccentric* engines, that show completely opposite detection patterns. These types of AV engines have been demonstrated empirically through data-driven analysis of malware detections.

In fact, AV engines' uncoverings have been observed surprisingly uncorrelated, suggesting the counter-intuitive idea that many engines are needed to justify the entire set of applications, as suggested by the 48 dimensions required to obtain 95% of

variance in the PCA analysis of AV detections. Nevertheless, rule mining has shown that a good number of engines follow or copy very often their patterns, clearly separating follower engines into pure followers and leaders.

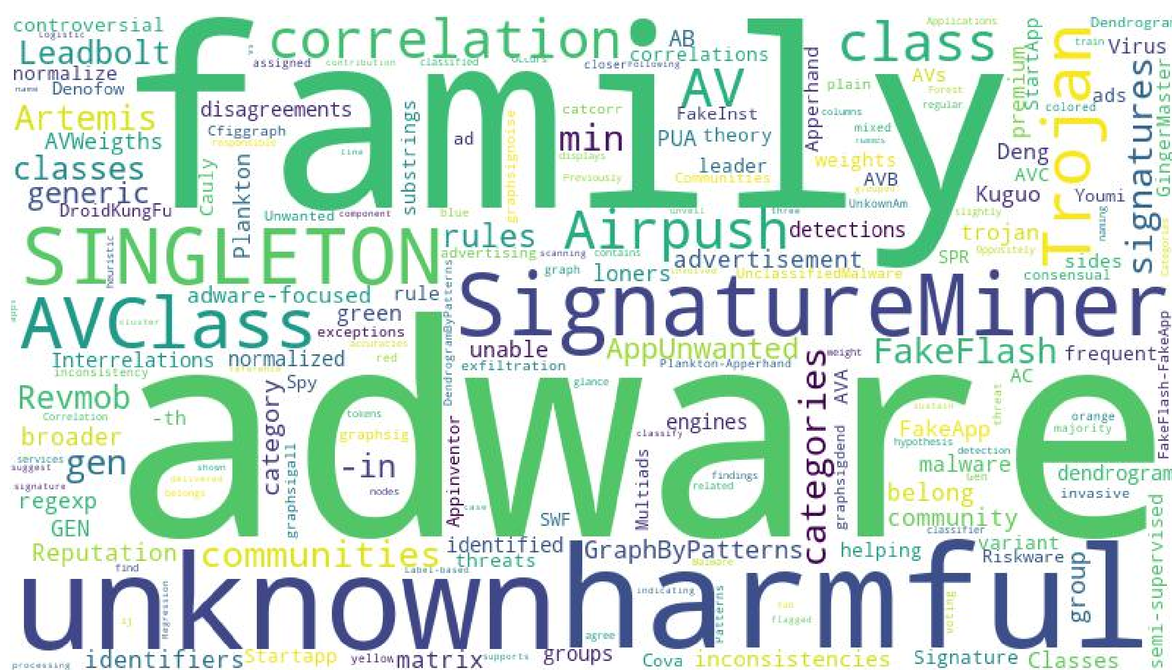
As a result, we have disclosed three groups of engines that fit the afore-defined scheme: (i) *Leader* engines, including AV1, AV15, AV25, AV31 and AV42, which present a follower-like behavior, but more independent than other follower engines suggesting they are the leading detectors among all engines; (ii) *Follower* engines, which comprise a large subset of engines with very high correlations and consistent detections with leaders and one another themselves and, (iii) *Eccentric* engines, that have almost no correlation with others and follow their own detection patterns, like AV17 or AV26.

Finally, we have attempted to assess application *malware risk* through a combination of Structural Equation Models and the logistic function that has been validated through its consistency with AVCounts (i.e. the detection account of each sample). In this process, engines have been implicitly weighted by the SEM model and produced weights consistent with the observed group scheme of leaders, followers and eccentrics.

The next chapter will continue with the analysis of AV engines by focusing on other key element of multi-scanner outputs: detection signatures, that is, the string tokens AV engines produce to identify the detected threat together with the detection result.

Chapter 6

AV Label-based Analysis of Malware Families



Following Chapter 5, AV engines have been shown to sustain three different detection patterns, namely: *Leaders*, *followers* and *eccentrics*. Such patterns have been found using the detection matrix of the dataset which contains whether an app is flagged or not by each engine from a one-hot binary encoding matrix. However, multi-scanner AV detections (a.k.a. signatures) typically include more information related to the detection, such as the scanning dates or identifiers of different malware classes.

In any case, such malware identifiers are delivered as textual references with no structure or format supporting machine processing. Actually, malware is not the same altogether: malware families or classes are used to differentiate among malware according to their behaviors and actions. According to the literature many engines present large naming inconsistencies when determining the family each sample belongs to, that is, AV engines do not agree on what is malware and neither on what malware family each sample belongs to.

In this light, this chapter continues AV engine analysis from a different perspective than in Chapter 5: we now focus on the patterns of engine interrelations with respect to malware families. Again, we consider the same suspicious application collection of Chapter 5, that comprises 82,866 applications that have produced 259,608 free-text signatures that must be normalized and inspected to provide insights of malware classes or families.

Through this chapter such analysis is structured as follows: we begin in Section 6.1 detailing our *SignatureMiner* approach to homogenize and normalize different AV signatures. In Section 6.2 we leverage SignatureMiner to find a normalization scheme for the nearly 260K signatures in the collection and categorize the resulting families into three broader categories, namely: *adware*, *harmful* and *unknown*. Then, in Section 6.3 we perform family-based pattern detection supported by correlations and graph theory algorithms aiming to unveil potential naming inconsistencies. Afterwards, the most relevant advancement is presented in Section 6.4, where an ML classifier of malware samples into adware or harmful class is trained and evaluated. Finally, Section 6.5 summarizes this chapter findings and remarks the most relevant conclusions.

6.1 SignatureMiner: A fast Anti-Virus Signature Intelligence Tool

To enable cross-engine analysis in multi-scanner detections, we have developed *SignatureMiner*, a lightweight application for the semi-supervised extraction of malware information encoded inside AV identifiers. Once adjusted, SignatureMiner can return a *normalized* malware family that is consistent across all engines.

To this aim, SignatureMiner enables semi-supervised training by helping users to create a set of rules to be applied to existing AV solutions. This process is based on the nature of malware detection signatures, that contain common substrings indicating details from each threat. For instance, signatures can contain parts of tokens like *"Trojan"*, *"GingerMaster"* or *"FakeFlash"*, which provide useful information regarding the type of malware each sample contains. Consider as an specific example an adware sample detected by three different AV engines, namely: AVA, AVB and AVC. When asked, each engine detecting malware returns a different detection signature like the following ones:

- AVA: *a variant of Android/AdDisplay.Startapp.B*
- AVB: *Adware/Startapp.A*
- AVC: *Adware.AndroidOS.Youmi.Startapp (v)*

which agree on labeling the malware sample as a variant of adware from the *Startapp* family. While this process is simple and obvious for the human reader, it is a challenging task for modern computers and devices.

FIGURE 6.1: Wordcloud of the observed AV signatures in the dataset. The size of each signature is consistent to its frequency in the dataset.

As an illustration, Fig 6.1 depicts a wordcloud of the signatures inside the detection collection. The size of each signature is proportional to the frequency of appearance, showing at first glance that adware is the most frequent type of malware in this dataset. Besides, different signatures containing the same families appear, such as *StartApp*, *RevMob* or *Leadbolt*.

To overcome these differences, SignatureMiner separates detection identifiers into a subset of *identifier tokens* that represent the items found in different detections. Starting by applying conventional text-cleaning techniques, SignatureMiner strips raw strings of weird characters, applies lower-casing, punctuation removal and text tokenization by dots. Then, SignatureMiner leverages min-hashing to put similar tokens together for manual inspection (recall min-hashing is detailed in Section 3.1).

Similar substrings are aggregated under the same min-hashes and presented to the human user so they can create detection rules based on Python's regular expressions. For example, *htmlframe* and *htmliframe* tokens or *yomi* and *youmi*, would easily obtain the same min-hash, helping the user to find a rule that targets them together. This process can be repeated as much as needed using at each step the rules already defined at each point, which enables systematic improving and refinement.

Moreover, this SignatureMiner mining component can help users to define alternative directives that produce other signature merging schemes, like neglecting certain families or defining broader family schemes. At the end, the resulting rules are used by SignatureMiner's classification component to produce the normalized class defined in the rules upon receiving a new signature from any AV. As an additional component, SM supports majority voting to directly output a single classification output. SignatureMiner source source code is available in Github [203].

6.1.1 SignatureMiner Performance

In order to assess SignatureMiner performance and capabilities, it has been compared to the most relevant work in the literature: *AVClass*; a detection signature normalization system that is trained over large datasets to extract the most important malware classes and find them in newly incoming signatures [110]. Since the code for AVClass is available online, we downloaded and executed it over our dataset and compared results with those of SignatureMiner.

AVClass repository contains a pre-trained version, which is the one used for the experiments following the suggestions of AVClass authors. At the same time, we trained SignatureMiner by elaborating a set of rules over the malware applications of the dataset and assigning each sample a unique class through the majority voting feature of SignatureMiner. Following AVClass terminology, any uncertain outcome, like ties in majority voting, is referred to as *SINGLETON*.

SignatureMiner does only generate 9,348 SINGLETON cases, which is roughly 11.5% of total data. In contrast, AVClass is unable to accurately classify 48,743 (52.1%) malware samples from the dataset, generating as many SINGLETON labels. When combined, both approaches are unable to determine a class for only 5,504 samples, which is the number of samples that receive SINGLETON label by both SignatureMiner and AVClass. Moreover, both approaches agree on the class of 28,330 samples and lack consensus on 1,932 samples. Neglecting SINGLETON cases on any of the two sides, the actual coincidence by both approaches is 93.6%, which is very high.

Actually, many disagreements are a consequence of the *SINGLETON* class label in any of the two sides, being the 10 most frequent mismatches due to SINGLETONs. Of those, AVClass is responsible for 9, whereas SignatureMiner is responsible for only one. In general, completely different class assignments are rare: the most frequent case, which is *adware* (SM) vs *adwo* (AC), occurs only in 152 cases. Oppositely, the most frequent mismatch is *adware* (SN) vs *SINGLETON* (AC) and occurs 12,290 times.

Finally, it is worth noting that SignatureMiner requires only as many samples as those available at classification time. In contrast, AVClass authors warn that AVClass "requires as input the AV labels for a large set of samples, e.g. several million samples".

6.2 Malware Categorization and Classes using SignatureMiner

Signature detections in the analysis dataset can be homogenized thanks to the SignatureMiner solution. During the training process of SignatureMiner, 41 distinct malware families have been identified from the samples in the dataset. Using the SM classification component, a normalized class is assigned to each sample (classes and SM rules are detailed in Table 6.1 below).

Malware Classes identified in the dataset have been scrutinized to comprise a robust normalization scheme and further inspected to unveil three broader categories in which malware families are grouped:

1. Applications looking into fast monetary gain using too many ads or a maluse of them (in what follows *adware* type).
2. Applications attempting intrusive and harmful techniques such as information leakage (in what follows *harmful*).
3. Applications AV engines detect as threats but cannot produce an accurate class (in what follows *unknown*).

Table 6.1 provides a summary of each of the 41 ($S1, \ldots, S41$) rules obtained through SignatureMiner along with their malware family and other details. Concisely, the table contains the predicate for each rule in regular expression syntax, family name of the malware class assigned and its associated broader malware category (i.e. adware, harmful or unknown), along with the number of cumulative detections, applications and AV engines from each class respectively. In what follows, a detailed summary of categories, their classes and specific details of each threat is presented.

Adware

Adware class labels are present in 60, 538 applications, which is a very large penetration of 73% of the dataset. Since this application collection comes from Google Play, these figures suggest a clear preeminence of adware-related apps in this market.

- *Leadbolt*, *Revmob*, *Startapp*, *WAPSX*, *Dowgin/dogwin*, *Cauly*, *Modwin* and *Apperhand/Counterclank* are well-known advertisement networks that can be used to perform full screen and invasive advertising malciously.
- *Kuguo*, is an advertisement library which has been known due to the abuses and maluses of its developer community.
- *Youmi* and *DroidKungFu* are advertising services that have been involved in data exfiltration incidents.
- *Airpush* is an advertisement network known for the abuse of adbar pushing notifications.
- *Fraud/osoneclick* is a fraudulent adware piece that attempts to increase number of ad clicks by stealthily displaying advertisements behind interactive applications.
- *Adware (gen)* tag is a generic reference assigned to those samples that engines detect solely as adware. In addition, other AVs mark as *Multiads* applications that contain different advertisement libraries known for the display of invasive ads.

TABLE 6.1: Malware classes, their figures and their SignatureMiner rules

#	Regexp rule	Family	Cat.	Det.	Apps	AVs
S1	.*a[ir]*push?.*	Airpush	Adware	35,850	12,802	26
S2	.*leadbolt.*	Leadbolt	Adware	17,414	4,045	21
S3	.*revmob.*	Revmob	Adware	38,693	13,680	18
S4	.*startapp.*	StartApp	Adware	29,443	11,963	13
S5	[os]*apperhand.* | .*counterclank.*	Apperhand	Adware	1,606	716	12
S6	.*kuguo.*	Kuguo	Adware	2,127	1,893	23
S7	wapsx?	WAPS	Adware	1,546	344	6
S8	.*dowgin.* | dogwin	Dogwin	Adware	1,098	421	23
S9	.*cauly.*	Cauly	Adware	1,143	626	3
S10	[os]*wooboo	Wooboo	Adware	220	120	14
S11	[os]*mobwin	Mobwin	Adware	1,284	249	3
S12	.*droidkungfu.*	DroidKungFu	Adware	105	54	3
S13	.*plankton.*	Plankton	Adware	4,557	741	25
S14	[os]*you?mi	Youmi	Adware	1,472	370	22
S15	[osoneclick]*fraud	Fraud	Adware	736	382	19
S16	multiads	Multiads	Adware	560	555	3
S17	.*adware.* | ad.+	Adware (gen)	Adware	33,133	24,515	46
S18	riskware	Riskware	Harmful	1841	1353	14
S19	spr	SPR	Harmful	1,789	1,789	2
S20	.*deng.*	Deng	Harmful	2,926	2,926	1
S21	.*smsreg	SMSreg	Harmful	649	440	16
S22	[os]*covav?	Cova	Harmful	1,564	1,296	5
S23	.*denofow.*	Denofow	Harmful	1,224	610	11
S24	[os]*fakeflash	FakeFlash	Harmful	1,381	510	15
S25	.*fakeapp.*	FakeApp	Harmful	518	420	14
S26	.*fakeinst.*	FakeInst	Harmful	493	401	22
S27	.*appinventor.*	Appinventor	Harmful	4,025	3,113	6
S28	.*swf.*	SWF	Harmful	4,651	4,566	10
S29	.*troj.*	Trojan (gen)	Harmful	23,775	16,851	49
S30	.*mobi.*	Mobidash	Harmful	981	796	16
S31	.*spy.*	Spy	Harmful	1483	1,221	26
S32	.*gin[ger]*master	Gingermaster	Harmful	58	36	10
S33	unclassifiedmalware	UnclassifiedMalware	Unknown	857	855	1
S34	.*virus.*	Virus	Unknown	959	896	15
S35	.*heur.*	Heur	Unknown	182	179	15
S36	.*gen.*	GEN	Unknown	9,827	9,118	25
S37	[osgen]*pua	PUA	Unknown	1,249	1,152	2
S38	[ws]*reputation	Reputation	Unknown	2,886	2,885	1
S39	.*applicunwnt.*	AppUnwanted	Unknown	4,863	4,860	1
S40	.*artemi.*	Artemis	Unknown	9,662	6,175	2
S41	.* (Default Case)	Other	Unknown	10,778	7,880	57
	TOTAL			259,608		

Recall that adware is a very controversial type of malware detection, since it is generally a legitimate way for developers to obtain a benefit for the applications they offer for free. This practice is widely accepted and therefore, cannot be considered malware in all cases. Consequently, the line between good and malicious use of advertisement is thin and different engine brands develop distinct policies that become inconsistencies and disagreements on a multi-AV environment.

Harmful

This category is reserved for more dangerous applications that perform actions directly harming users, such as premium service enrollment or data exfiltration. The total number of applications with at least one harmful label is 29, 675.

- *Deng*, *SPR (Security and Privacy Risk)* and *Riskware* are given to flag applications that require too many *harmful* permissions or threaten user privacy.
- *Denofow* and *Cova* are trojan programs that subscribe users to premium SMS services.
- *SMSReg* is a generic indicator for applications that require SMS-related permissions for data exfiltration or premium subscriptions.
- *FakeFlash*, *FakeInst* or *Fakeapp* are names for applications that look like or even replicate popular apps to lure users into installing their malicious code.
- *Appinventor* is a developer platform that has been very popular for malware developers.
- *SWF* stands for different versions of Shockwave Flash Player exploits.
- *Trojan (gen)* is the generic reference for trojan applications as detected by engines.
- *GingerMaster* is a well-known family of rooting exploits.
- *Spy* is a generic reference for spyware malware.

Unknown

This category includes AVs which detections include generic class-related information, including generic detection signatures and those signatures that do not fit any other rule. There are 23, 915 applications getting at least one of these classes.

- *UnclassifiedMalware*, *Virus*, *Heur* (from heuristics), *GEN* (Generic Malware), *PUA* (Potentially Unwanted Application), *Reputation*, *AppUnwanted* (Application Unwanted) and *Artemis* are generic identifiers given by different engines in order to indicate applications that are detected as harm without any further details.

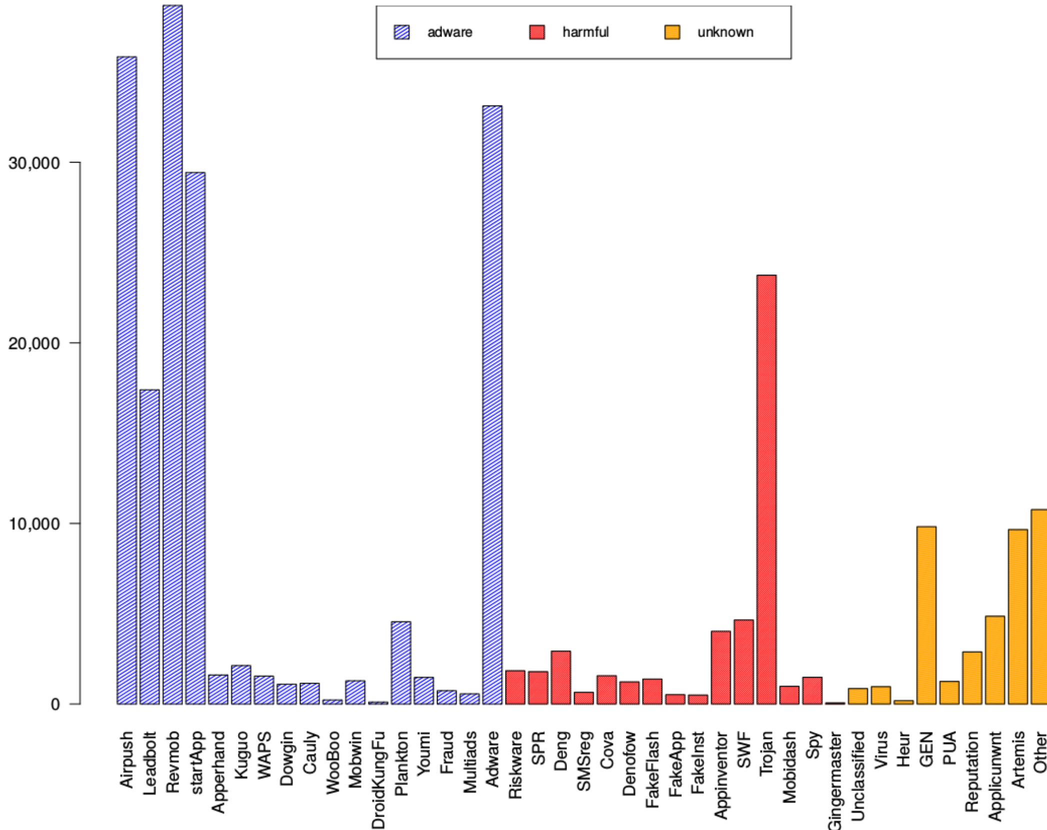


FIGURE 6.2: Frequency of detections per malware class. The colors in the figure are consistent with the assigned categories.

- *Other* includes the applications not assigned to any other family by SM.

Altogether, *Revmob*, *Airpush* and *Adware* are the most popular signatures involving many AV different engines and detections. *Trojan* detections are also very popular in the harmful category and identified by up to 49 different engines. Indeed, Fig. 6.2 depicts the numbers in Table 6.2 by category where adware families popularity is clearly displayed.

In general, family classes are spotted by more than one engine, with some exceptions, usually in the *unknown* category, where family classes like *Reputation* or *AppUnwanted* are flagged by a single AV engine. Additionally, the most active engines indicated in Chapter 5 are also very active in the detection of adware samples, further supporting the adware popularity in Google Play.

Family matrix

Using such normalized classes, let us define a matrix B of size $82,866 \times 41$ where the elements $B_{ij} \in \{0, 1\}$ have value 1 if the i-th app has been identified to belong to the j-th family malware category or 0 otherwise. Such matrix is named *Family Matrix* and each of the vectors will be referenced individually as *Family Vector* of the application.

Out of the 43,933 applications with more than one detection, 63.26% are assigned to more than one malware family by different AVs,reaching some applications up to 12 distinct malware families. These results evince that not only do AV engines fail in agreeing on malware detection, but also in the specific family or category a sample may be related to. The following sections deepen on this observation and provide additional insights.

6.3 Malware Family Classes and Categories Interdependences

Recall there are 27,781 applications in the collection which family classifications are controversial, that is, more than one single family. This discovery is in line with the literature [103, 102], as one third of applications lack consensual detections.

In what follows, multi-class applications will be analyzed and explored in order to identify whether they correspond to name duplicities where the same malware is given different names or just plain engine disagreements.

6.3.1 Correlation of Malware Categories

Previously, each of the 41 malware classes has been assigned to one of three possible categories, namely *adware*, *harmful* and *unknown*. For categorical analysis, we define the *Category Matrix* C as an $82,866 \times 3$ matrix where C_{ij} accounts for the number of times the i-th application has received a detection of each category. The correlation of the columns of matrix C can indicate and approximation on how frequently each pair of categories are assigned to the same application. Table 6.2 illustrates the correlation matrix of C.

TABLE 6.2: Correlation of matrix C (Malware Categories)

	Adware	Harmful	Unknown
Adware	1	0.06	0.3
Harmful	0.06	1	0.44
Unknown	0.3	0.44	1

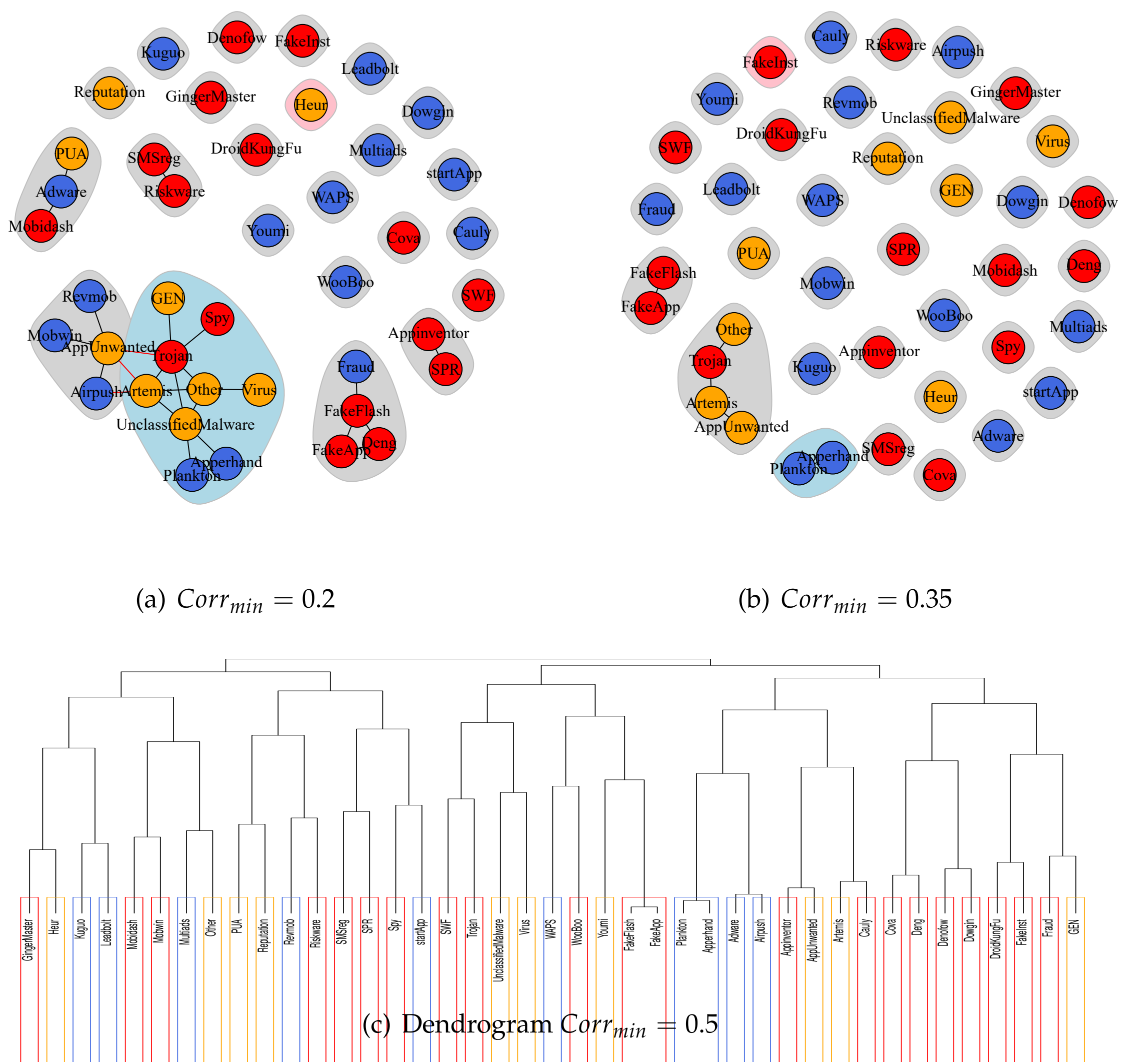
In the table, harmful and adware categories show really weak correlation values (0.06), indicating that AV engines are usually clear on whether some app is adware or harmful. Oppositely, the unknown category shows higher correlation values, both with adware (0.3) and harmful categories (0.44).

In this light, many AV inconsistencies seem to be present around the unknown category. Indeed, the unknown class corresponds to a set of samples AV engines are unable to correctly categorize, but still recognize them as threats and assign a *synthetic* family identifier. Anyway, correlations suggest such ambiguities should belong to any of the other two categories, being the harmful class slightly more involved.

6.3.2 Graph Community Clustering to Detect Class Redundancies

Starting from the correlation matrix of family matrix B, it is possible to build and explore a graph using network theory. For that, we compute the correlation matrix $Corr(B)$ and create a graph using it as adjacency matrix. The resulting graph has 41 nodes (as many as malware classes) separated according to their pairwise class correlations.

With the help of *edge betweenness* algorithms for community search (See Section 3.3 for reference), nodes are grouped into high correlation clusters to find related malware classes according to the discrepancies of AV engines. To prevent noise from building insignificant communities, we set a threshold, $Corr_{min}$, below which correlation values are set to zero.



(a) $Corr_{min} = 0.2$ (b) $Corr_{min} = 0.35$

(c) Dendrogram $Corr_{min} = 0.5$

FIGURE 6.3: Communities of malware classes for different correlation threshold values

Fig. 6.3 displays two different communities for $Corr_{min}$ equal to 0.2 and 0.35 along with the dendrogram for the community groups with $Corr_{min} = 0.5$ where the evolution of groups according to increasing correlation thresholds is shown. In

Fig. 6.3(a) many groups are clearly defined, even though their intra-community correlation is not very high. Interestingly, most communities mainly involve families from a single category, notwithstanding the concurrence with unknown samples. Actually, raising the correlation slightly, as Fig. 6.3(b) shows, eliminates almost all groups leaving only three survivor groups, which members belong to the same category.

Hence, the most interesting communities are three: one larger community formed by three unknown signatures (*AppUnwanted*, *Artemis* and *Other*) and one harmful threat (the generic trojan family) and two smaller communities, *FakeFlash-FakeApp* and *Plankton-Apperhand*.

The case of the last two communities, namely *FakeFlash-FakeApp* and *Plankton-Apperhand*, is comprised by two moderately correlated pairs (0.61 and 0.72 respectively) and suggest inconsistencies according to family names and relations. First, the case of FakeFlash and FakeApp could belong to a minor name inconsistency, since FakeFlash could be considered an specific case of FakeApp. Then, the case of Plankton and Apperhand the problem is less obvious and seems a plain nomenclature inconsistency where different AVs give different names to the same threat.

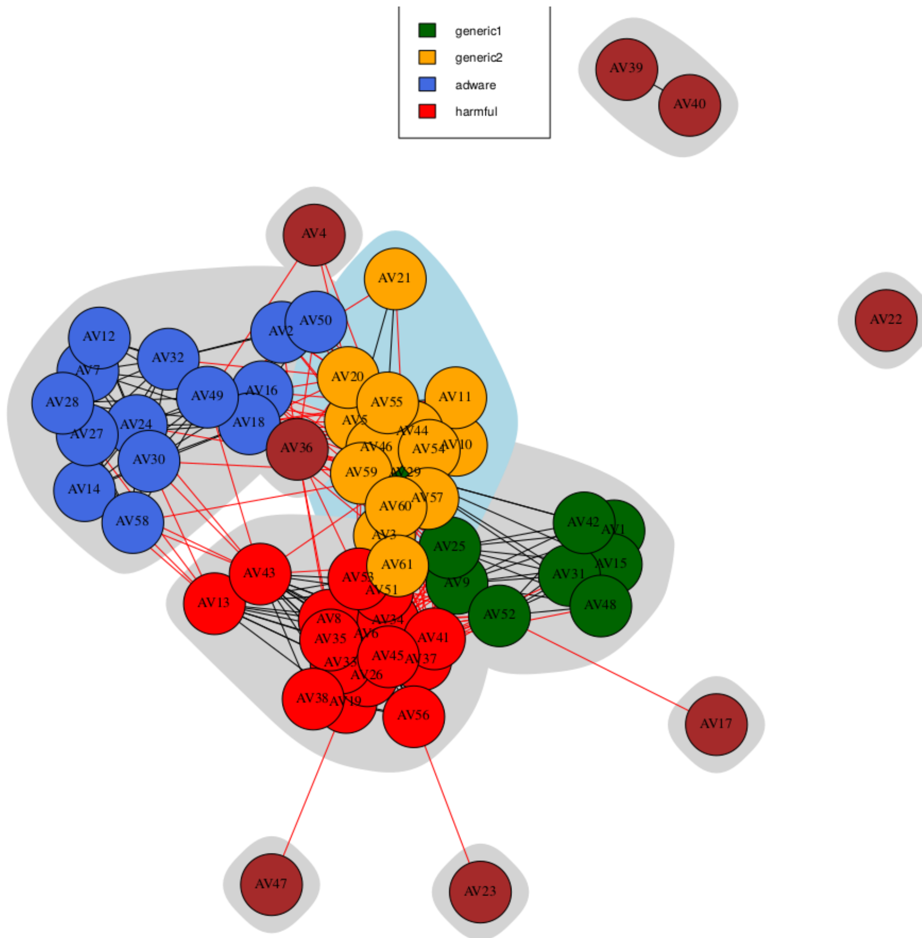
Finally, the larger community members show smaller correlation values than the previous ones. Actually, the presence of a harmful member could be hinting a slightly larger lookalike to harmful rather than adware categories for those unknown families. Indeed, the dendrogram in Fig. 6.3(c) does show this third group disappearing when $Corr_{min}$ reaches 0.5. Eventually, same-category classes tend to aggregate together with the exception of the unknown class, but keep some inner structure among them, sometimes relating to opposing classes.

In conclusion, most of the signatures obtained through SignatureMiner are robust as not many others are detecting the same malware. There are only two clear exceptions described along this section that will be merged for the rest of the chapter. In addition, unknown classes have shown overlapping patterns with both adware and harmful categories, suggesting they may be part of them inherently. These observations show that unknown classes might be classified into their correct category, as will be described in Section 6.4.

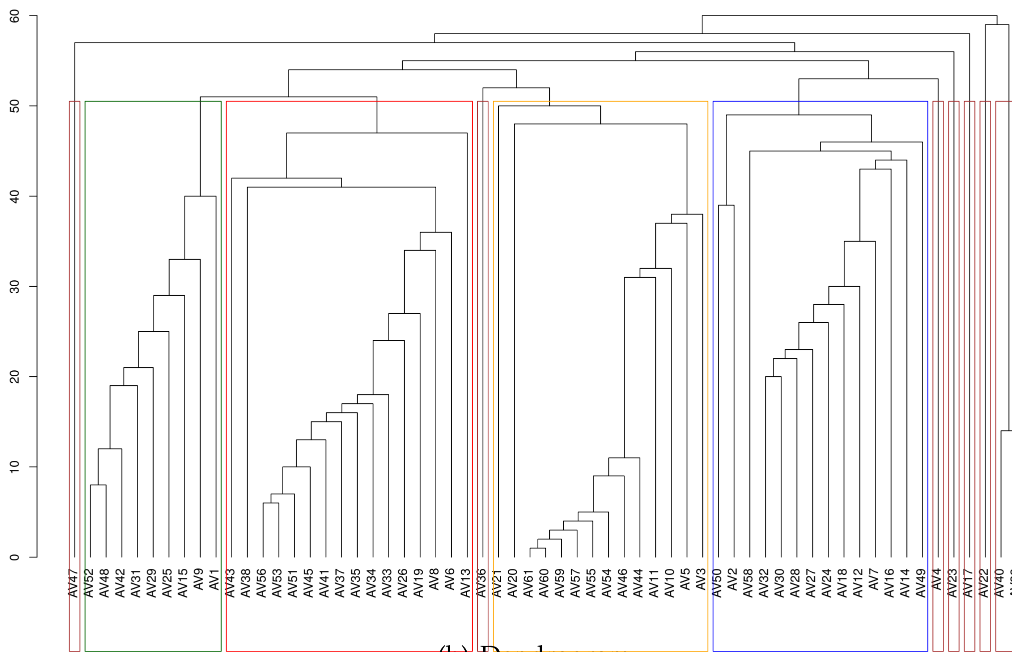
6.3.3 Grouping AVs by their Detection Schemes

To investigate how AV engines relate according to the families they detect, we leverage a derivation of the detection and the family matrix, which represents at each element ab_{ij} the number of detections performed by AV engine j (columns) of class i (rows). We call this matrix the *correspondence matrix* AB and leverage graph theory again using $Corr(AB)$ as adjacency matrix.

In this case, Fig. 6.4(a) shows the AV-based graph obtained for engine relations containing nodes colored according to their group ($Corr_{min} = 0.35$). In the community plot, most AV engines are grouped into a certain community while a few others, matching with the eccentrics discovered in Chapter 5, appear as loners. These AVs



(a) Graph groups



(b) Dendrogram

FIGURE 6.4: Community cluster of AV engines according to the classes they detect

are AV22 and AV39-AV40 that have no relation at all and AV4, AV47, AV23 and AV17 with subtle relations to some others in groups.

Besides, there are four main groups or communities: an adware-focused group (blue), which most frequent detections are *Revmob*, *Adware*, *Airpush* or *startApp*; a harmful-oriented group of AVs (red), with top families such as *Trojan*, *Airpush*, *Gen* or *Kuguo* and two mixed groups: The first (green) shows mainly *Plankton*, *Adware* or *Trojan* detections whilst the second one (orange) contains detections on *Other*, *Deng*, *Airpush* or *Leadbolt*.

Fig. 6.4(b) depicts the dendrogram of matrix AB, where the same patterns are observed: loners are way too far from the rest as compared to other engines and therefore connected very high in the dendrogram. Other groups show internal differences, suggesting they could be clustered in more fine-grained groups, but still make sense as groups.

Furthermore, the position of each group in the dendrogram is very interesting as well: the harmful group is in between both mixed groups and away of the adware related group, indicating a marked difference in their detections despite they somehow converge in the yellow group. Additionally, most loners are closer to adware, which could be a sign of them being more adware focused. On the other side, the green group is closer to harmful and matches very much with the *leader group* in Chapter 5, that evinces leaders catch mainly harmful threats, even though not losing most relevant adware samples, which strengthens their leadership hypothesis and positions them as suitable candidates for effective malware labeling.

These groups further demonstrate how engines often incur in similar detection patterns at the family level, being category-oriented detections very frequent. It is worth noting that the yellow group contains many unknown categories along with top engines from harmful categories while, at the same time, the green cluster is more associated with broad-focus engines targeting similarly adware and harmful categories.

6.4 Identifying Unknown Malware

Previously, malware families within the *unknown* category have been observed to be more related to *harmful* than to *adware* category according to their correlations (0.44 to 0.3). This section attempts to unveil the specific malware category (either harmful or adware) of malware families from the *unknown* category using Machine Learning for classification.

For this purpose, we developed a machine learning classifier to predict the malware category of a given sample using as input features the detection vector of each application. This classifier has a twofold objective: (i) provide a fast categorical assignment using solely AV detections with no family information, and (ii) further identify which AV engines are more powerful for either adware, harmful or both detections. Hence, we propose two different classifier algorithms *Logistic Regression* for optimized probability estimation and interpretability and *Random Forest* for performance (these and other ML methods are reviewed in Section 3.2).

The Logistic Regression algorithm has been regularized to improve its performance and analyze AV engine contribution, forcing less biased AVs towards a category to be have near-zero weight while other engines are associated with a weight according to their relevance for harmful detection (positive contribution) or adware (negative contribution). Lasso regularization is used due to its well-known performance over binary feature-sets.

To train and validate both algorithms we use the samples in the adware and harmful categories first, assuming as label their majority voting category facilitated by SignatureMiner. Hyper-parameter tuning is performed using 10-fold cross-validation.

Table 6.3 displays the performance results of both ML algorithms during training and validation, showing F-score values above 0.75 for Logistic Regression and 0.84 for Random Forest. As expected, RFs outperform LR in terms of accuracy, showing outstanding results (0.92 accuracy test). The optimal hyper-parameter values are also shown in the table.

TABLE 6.3: Train and validation scores for the different models.

Algorithm	Acc_{train}	Acc_{test}	F_{train}	F_{test}
Logistic Regression	0.895	0.894	0.758	0.757
Random Forest	0.935	0.927	0.859	0.841

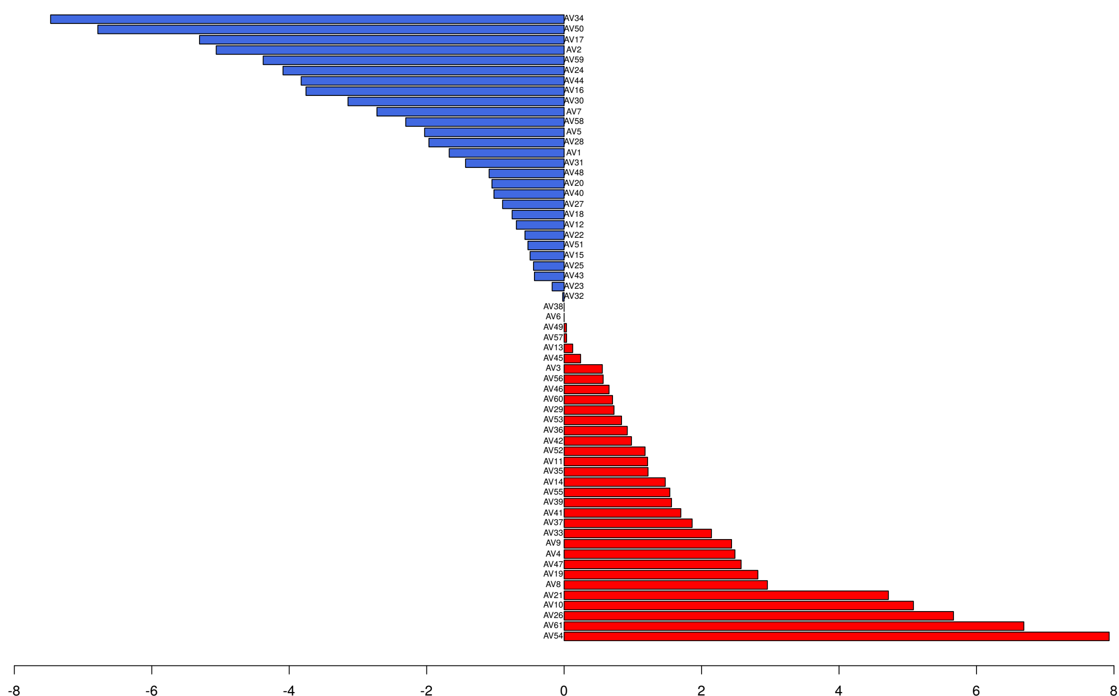


FIGURE 6.5: Computed weights after ℓ_1 logistic regression.

Regarding AV engines, Fig. 6.5 displays the logistic regression weights assigned to each engine and colored consistently with the category they contribute more, namely adware (blue) vs harmful (red). Recall that positive weights indicate engines detecting more harmful malware and negative weights are assigned to more

adware-focused engines. At a first glance, there are slightly more harmful-aware AVs (31) than adware-aware ones (26). This suggest that most engines are focused on detecting all types of malware, even though, there is a considerable amount of mostly adware-focused engines.

The results observed in Fig. 6.5 are consistent with Fig. 6.4(a), where most AVs contributing to adware belong to the adware class in the dendrogram of Fig. 6.4(a). Indeed, examples include AV50 and AV2 in the adware category and AV8 and AV26 in the harmful class.

Zero contribution engines are interesting as well, since they appear to be performing detections for both categories consistently. In fact, leader engines have weights near zero, indicating that they detect both categories without sharp specifications, i.e. their detections target both adware and harmful and thus are not useful for ML classification. This is consistent with their grouping patterns, as this group has been shown to detect both adware and harmful samples. On the contrary, the orange group, that had the larger proportion of unknown category samples, contains some of the best weighted engines for the harmful category, like AV54 and AV61, which indicates certain harmful tendency.

Finally, we have used the trained RF model to classify the apps in the unknown malware category. As a result, we observe that 51.5% of the samples are classified as harmful while the remaining 48.5% belong to adware. This result is in-line with the correlation experiments of Section 5.2 which already pointed to a slightly larger amount of unknown samples belonging to the harmful category.

TABLE 6.4: Amount of harmful samples detected at each family in the unknown category

Family	Virus	Heur	GEN	PUA	Reputation	Artemis	Other	SINGLETON
Harmful	83.48%	47.87%	34.63%	50%	61.8%	41.66%	39.98%	38.77%

Moreover, Table 6.4 summarizes the classification results for each malware family in the unknown category separately. Concisely, the percentage of apps that are identified as harmful by the classifier are displayed. As shown, most apps tagged as *virus* fall in the harmful category, whereas the *gen* family class is closer to adware. Within *reputation* more applications seem to belong to harmful while *artemis*, *SINGLETON* and *other* are closer to adware. There are two signatures missing: *applicunwt*, which is not selected as target family in any case by majority voting and *unclassifiedMalware*, which appears only once and is classified as adware.

6.5 Summary and Conclusions

Following the analysis of the 82,866 suspicious samples dataset and their multi-scan signatures started in Chapter 5, this chapter focuses on malware categorization and family assignment. In this chapter, the previous disagreements among engines are also observed in malware classification, incurring in extended name inconsistencies and categorization disagreements.

Initially, SignatureMiner has been presented. SignatureMiner is a semi-supervised threat intelligence tool based on min-hashing that helps users to *mine* information regarding Antivirus detections. The tool has been used to normalize the 259, 608 signatures associated to each of the different applications into 41 malware families which has enabled further categorization into three broader categories, namely adware, harmful and unknown.

Then, classes and categories have been analyzed and inspected through correlation and graph community detection showing interesting patterns, in line with the findings of Chapter 5 regarding followers and leaders. For instance, leader engines in Chapter 5 belong to the same generalist darkgreen detection cluster in Fig 6.4 (Section 6.3).

Furthermore, correlation analysis has showed that the unknown category is typically present in controversial detections, that is, those where more than one category is involved. This finding has motivated the hypothesis that unknown samples should mainly belong to either adware or harmful and as a result, we have proposed in Section 6.4 a ML classifier to determine whether any suspicious sample belongs to adware or harmful based on the AVs detecting such sample.

By training the classifier with harmful and adware samples, we validate a model with F-score up to 0.84, that has been later used to classify the aforementioned unknown samples. This classifier has further demonstrated that unknown malware classes are more likely to represent real threats to the users rather than regular adware.

The results from this chapter and the previous one have set a broad set of rules and methods to detect AV engine trustworthiness. In our case, we have detected a small group of AV engines formed by AV1, AV15, AV31 and AV42 that shows the most consensual and successful detection scheme including all types of malware samples and families.

Chapter 7

Android Meta-data for Repackaging Detection

In this chapter, we present CloneSpot, our tool for repackaging detection based on the clustering of similar textual meta-data from applications. Initially, Section 7.1 details the crawling methodology and methods. In Section 7.2 the CloneSpot methodology, capable of detecting application duplicates, is presented together with its web application and validation in Section 7.3. Finally, Section 7.4 concludes and wraps the chapter with a summary of the most relevant conclusions and achievements.

7.1 Meta-data Application Collection

In this chapter, we consider again the meta-data from applications collected from Google Play. In this case, we develop a massive crawler that downloads application information from the web version of Google Play market. This system follows the next steps:

1. Google Play home page is downloaded to identify and recover as many application links as possible.

2. Starting from the obtained links, this process is recursively repeated until no new application link is obtained. This way, after some hours, the links for a large part of the market are obtained.

3. Using a custom multi-machine parallel crawler capable of downloading, parsing and storing each app meta-data, we collected nearly 1.3 Million apps meta-information in a few days (using in the process an Intel Xeon E5-2630 server with 24 cores 190 GB RAM memory). As of 2018, this number represents 61% of the Google Play Store.

4. After removing different application versions and duplicates, the size of the collection is reduced to exactly $1,288,643$ applications.

We developed the above crawler with a twofold objective: (i) quickly retrieving a list of URL links for a large proportion of applications in Google Play recursively and (ii) downloading all the meta-information (not the APK packages) contained in each application URL. The resulting dataset contains four elements of mainly qualitative nature: application title, description, category and developer name.

Table 7.1 summarizes these meta-data items, their description and type of variable (either categorical or free text). These features require a different approach than that of Chapter 4, since they do not contain any numerical value to be used in an ML setting. Moreover, the dataset lacks any type of detection label, which makes supervised learning impossible.

TABLE 7.1: Summary of the features collected from Google Play. The date of collection is 2017.

	Name	Description	Value
1	Title	The application title	Free Text
2	Category Name	Assigned Google Play Category	Categoric
3	Description	A textual description of the application and its features. It can be written in different languages and has a variable extension	Free Text
4	Developer Name	The associated local name of the developer account within the studied market	Categorical

7.2 CloneSpot: Fast Detection of Application Duplicates

By design, the Java code in Android applications can be approximated through decompilation, modified with new functionalities, re-compiled and uploaded to any

application market. This design facilitates the generation of duplicated application versions with altered behaviors very fast and with little effort. This process is known as *repackaging* and has become the most popular method to hinder and distribute malware applications within markets, including Google Play [204].

At present, repackaging is a very trending problem in the Android community; for example, a repackaged version exists for 80% out of the top 50 most downloaded applications in Google Play [205].

As a result the ability of cloning or repackaging any Android application from a given market is within reach of anyone. It is worth noting that malware is not the only incentive for repackaging: monetizing third-party applications by means of introducing advertisement or simply plagiarizing application's code code are important motivations to perform application cloning.

The main attractive for repackaging apps is that rather than requiring application development from scratch, they can be used instead to introduce malicious code components in existing applications and upload the newly created app to any market. To this end, the application must be disguised as the legitimate app itself, with the nearly same functionalities and *meta-data* then the original victim application. Actually, meta-data features are the best presentation letter of an application to its potential users.

Consequently, most repackagers should be reluctant to undertake major modifications to application's meta-information that would reduce their chances to be mistaken for the real application. Indeed, many cloners stick to minor changes to one or more of these meta-data fields, such as changing some pixels in logos or subtly modifying titles or descriptions in a way that clones are identical for the human eye to make users install them believing they are installing the legitimate *victim application*.

Thus, the most straightforward procedure to detect clones would be a brute-force approach that computes the pairwise similarity of serialized meta-data entries and rank them accordingly for manual inspection. However, this solution is computationally costly and unfeasible to do over a 1.3M application collection in a reasonable time.

In this light, we propose *CloneSpot*, a comparative solution that relies on similarity clustering to aggregate alike meta-data applications together into *app-sets* that contain potentially cloned application candidates. Once within their app-sets, pairwise similarity is possible, since the number of pairwise comparisons is drastically reduced to the app-set members. The similarity clustering algorithm selected is min-hashing, due to its versatility, velocity and efficiency (see Section 3.1 for further details).

7.2.1 Clustering Qualitative Meta-data through Min-hashing

The CloneSpot-based system works as follows: application meta-data is serialized by joining each application's *title*, *category*, *description* and *developer name* fields with regular blank spaces. The resulting string is clustered using min-hashing with a

shingling key of $k = 7$ and the resulting min-hashes are used as keys for aggregating applications into app-sets.

Single application app-sets are considered out of repackaging risk and therefore removed. Besides, app-sets which applications belong to the same developer are removed under the assumption that a developer has no malicious incentive for repackaging his own applications, even though he might be reusing the same base code for different apps. Finally, too large app-sets make no sense, as their similarity must be based on frequent words (i.e. stopwords). Consequently, app-sets containing more than 100 applications are removed from the analysis of duplicates as well.

7.2.2 Market-scale Detection of Application Duplicates

The CloneSpot methodology can be used to analyze entire application markets in less than a day. The collected applications, contain app-sets with an average of 18.9 members, although the median value is two and the third quantile (75% percentile) is 8. Therefore, app-set sizes are preeminently small, being those containing less than 10 applications around 79% of the total. On the other side, there are $1,273$ groups containing more than 100 applications that have been removed.

As a result, there is a total of $54,944$ app-sets containing an overall amount of $419,830$ applications, which directly represent 32% of the original collection. The remaining applications have fallen into single size app-sets and have thus been filtered out. At this point, application inspection is easier at local level, even though dataset-wide manual inspection is not scalable yet.

Instead, group-wide pairwise operations can be considered, since the potential intra-group operations required are much less. In fact, performing all within app-set comparisons requires roughly 1.5 million operations[1], as compared to the 1.6 trillion operations required to complete the brute-force pairwise comparison of the entire collection.

7.2.3 Intra-group Detection Scoring

Generally, some degree of meta-data plagiarism could be tolerated, even though it is very unlikely that nearly-identical fields happen within two applications by chance. Hence, app-set pairwise similarity metrics and experiments are designed following the assumption that higher similarity scores are indicative of repackaging at each app-set.

To score app-sets, we consider two metrics involving each a distinct field from each application's meta-data: title and description. Following, *edit distance* for application titles and *cosine distance* for descriptions are detailed and combined to create the *Application Similarity Index* local scoring metric (references for the distance metrics used can be found in Section 3.1).

[1] Assuming 8 apps per app-set: $\binom{8}{2} \times 54,944$

Edit Distance of Application Titles

Subtle changes in repackaged application titles are a common practice of repackagers to avoid detection by non-expert users. These changes typically range from capitalization alterations to the addition of tiny or unnoticeable characters like dots, which are really difficult to detect, even to the expert eye, and can be effective unless the user is prevented.

Therefore, any two applications in the same app-set showing almost the same title is a very clear indicator of potentially cloned or fraudulent application. To detect such changes systematically, we compute the *edit distance* among the titles in the same app-set. Recall that the *Edit distance* between two strings a and b accounts the number of modifications to be made to a until it becomes b (for further reference on edit distance the reader can return to Section

Cosine Similarity of Application Descriptions

Repackaged samples have a tendency to paraphrase or directly copy the description in their victims' meta-data. Thus, description similarity is another indicator of repackaged applications that can be sharper in some cases: When the descriptions of two applications directly match, there is almost no doubt that both applications are clones. To avoid detection, some clones may try to modify their cloned description by altering the order of sentences and paragraphs in such a way the application still makes sense, which is important to select an appropriate mechanism for plagiarism.

In this light and assuming descriptions are sequences of free text, which are lists of words, we simplify them by transforming their text into *Bag of Words* (BoW) representations. Then, *cosine similarity* is computed over each pair or application descriptions' BoW to produce an orderless and language-independent similarity score for application descriptions.

The BoW representation of each description is obtained by *tokenizing* texts into words, separating them using blank spaces and removing trailing characters. Afterwards, each token frequency of appearance is computed and BoW are converted into vectors containing all the tokens appearing in any of the two applications. The *Cosine Similarity* is computed with these vectors, being the value for each term dimension the number of times such word appears in their respective description.

In this case, Cosine Similarity is bounded between 0 and 1 and produces a broadly accepted estimation of the *proximity* between two text descriptions. Moreover, thanks to the BoW approach this similarity is language independent and robust against the alteration of sentence or paragraph collocations.

Application Similarity Index

Intra app-set similarities of application titles and descriptions have been previously proposed as indicators of application cloning. Both schemes can be merged together

into a single measurement that balances description similarity with proximate titles in terms of edit distance.

For this purpose, we define the *Application Similarity Index* (ASI) between any two applications a and b as the quotient between median cosine similarity of descriptions and the average title edit distance:

$$ASI1(a,b) = \frac{median(CS(\vec{a},\vec{b}))}{\varepsilon(ED(\vec{a},\vec{b}))} \quad (7.1)$$

where CS is the cosine similarity between vectorized BoW a and b and ED refers to the edit distance metric. In short, this metric modulates application description similarity by their title similarity: the more alike two titles are, the closer the ASI is to the cosine similarity of descriptions. This way, near equal titles are very important, even capable of neglecting a similar description provided titles are not similar.

Additionally, ASI could be modified to be more effective in detecting clones in larger size app-sets. For that, the modulating edit distance would be changed to the minimum between any two apps in the app-set:

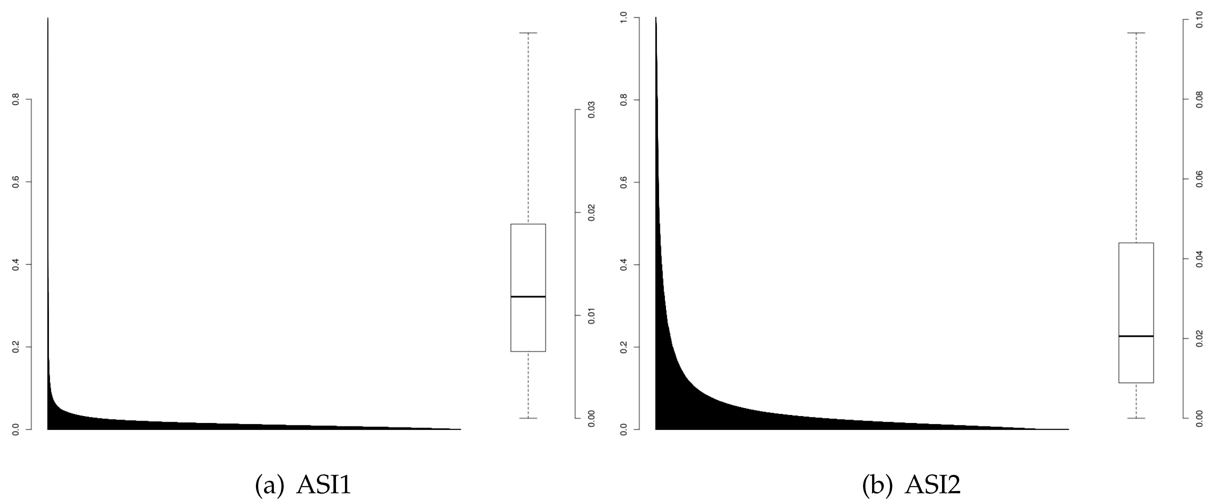
$$ASI2(a,b) = \frac{median(CS(\vec{a},\vec{b}))}{min(ED(\vec{a},\vec{b}))} \quad (7.2)$$

Both equations trade-off similarity in titles and descriptions to get a high ASI score. In both cases, different descriptions reduce the ASI accordingly, while too different titles drastically decrease ASI near to zero. Nonetheless, the first proposal (Eq. 7.1) is more focused towards small sized app-sets which applications are all clones while the second one (Eq 7.2) looks for a single pair of clones within any app-set.

Anyway, the intuition behind these formulations is that two almost-identical applications are more likely to be clones than any other not so similar pair. Recall that this metric can be computed locally over each app-set, enabling parallelization to speed up the process. As a result, any of the ASI metrics defined above can be used to sort app-sets into a ranked list of *potentially cloned applications* for malware analysts to inspect. Thanks to this ranking, CloneSpot is not only capable of serving potentially repackaged applications together, but also including a quantitative approximation to estimate how copied two applications are.

Fig. 7.1 provides a visualization of both ASI scoring distributions that shows in both cases very little applications with high scores and larger and homogeneous collections of low-score app-sets. Both figures clearly show the large impact title distances have on the final score. Furthermore, this zipf-like pattern is driven by many two-sized app-sets on top on the ranking with a high similarity (and thus a very large likelihood of being clones) which increases its score.

In fact, the figure clearly shows that using the minimum in ASI2 (Fig. 7.1(b)) yields larger ASI values than using the mean in ASI1, suggesting that some larger app-sets include a tuple of cloned applications among other similar but not-cloned applications.



(a) ASI1 (b) ASI2

FIGURE 7.1: Histogram of the distributions for both ASI scoring systems

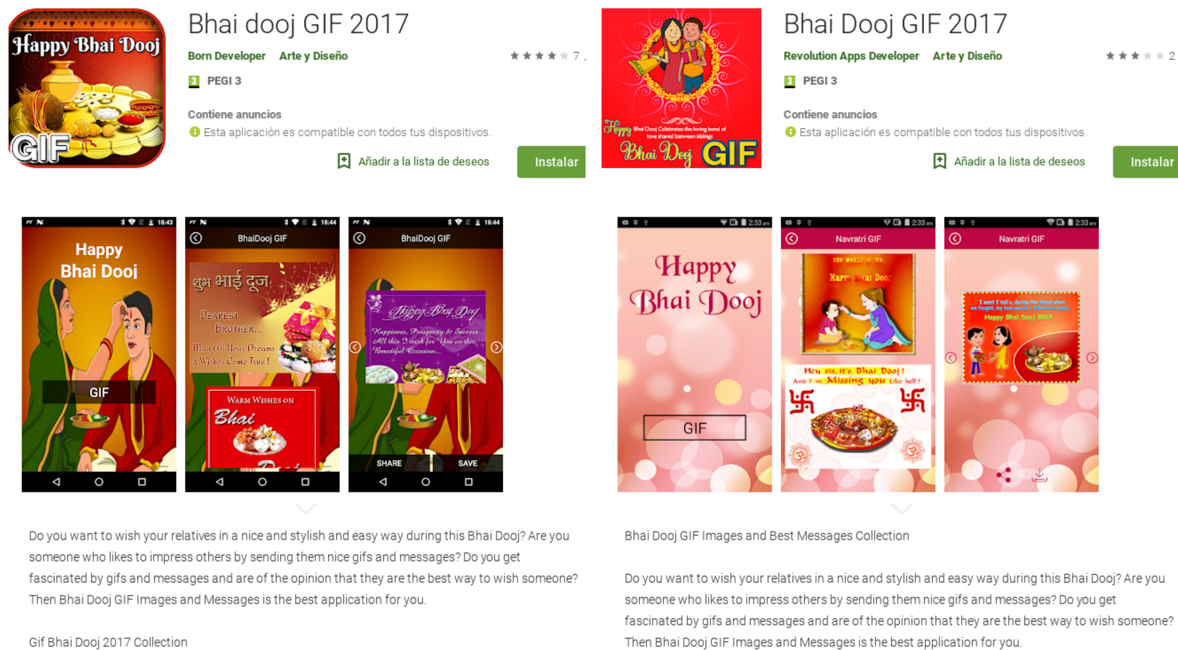


FIGURE 7.2: Screen-shot of the top application in $ASI1$ ranking

As an example, consider app-set number one in the ASI1 ranking: *Bhai dooj GIF 2017* and *Bhai Dooj GIF 2017*, which score 0.9979 in both ASI metrics, 1 in edit distance and 0.9979 in cosine similarity. Since it only contains two applications, $ASI1$ and $ASI2$ are exactly the same, even though the app is down to the 32nd position in the $ASI2$ ranking. Fig. 7.2 shows the Google Play home pages of both applications.

This example illustrates very well a repackaging case for plagiarism. The only difference is the capitalization of the word "Dooj" and both applications share the exact same description with a small add-on. Moreover, the UIs of both applications can be observed to follow the same structure, even though the images have been changed. In spite of the differences and the possibility of them sharing the subject,

the suspicious degree of lookalike in title and description along with the structural correspondence in UI structure indicates a fairly clear case of repackaging, probably used for monetary gain through swapping the advertisement account.

CloneSpot workflow is described in Alg. 1, which provides a methodological summary of the aggregation and ranking of app-sets from raw applications to the final result.

```
Data: CloneSpot: Google Play meta-data from 1.3M applications
Result: Ranked List of app-sets
1. Min-Hashing Clustering;
forall app in allApplications do
    Split in Shingles k = 7;
    Compute Min-Hash;
    Send app to Min-Hash Bucket;
end
2. Clean app-set list and Compute Local ASI;
forall app-set in groups do
    if Single-sized app-set or Single developer app-set then
        Remove app-set;
    end
    else
        Compute Title Edit distance;
        Compute BoWs and Cosine Similarity;
        Compute ASI;
    end
end
3. Sort app-sets according to ASI;
```

Algorithm 1: Summary of the repackaging detection pipeline

Next section validates the ability of CloneSpot to detect and rank clones in Google Play and presents a PoC application to illustrate how this approach can be used in real implementations to improve security and help analysts, application markets and developers.

7.3 CloneSpot service: Fast Retrieval of Potential Clones

After applying CloneSpot methodology to our 1.3M dataset, there are a total of $54,944$ ranked app-sets containing potential victims or clones ordered according to their local pairwise similarity. In this section the CloneSpot approach is validated through the following experiments:

- Number of applications in app-sets that have been removed from Google Play market: We measure the number of app-sets containing at least one application that has been removed from market between September 2017, May 2018 and October 2018.

- *CloneSpot*, a proof-of-concept web-service to demonstrate the applicability of this approach through a real-time database of meta-data from 1.3 Million applications in Google Play.

7.3.1 Application Removal in Google Play

Generally, application markets and more specifically Google Play, perform continuous and extensive reviews of their in-house applications that might end in market removal for different reasons. Such causes range from application withdrawal by its developer to flagrant cases of breaking market policies, for instance, plagiarism. Whenever an application is removed from Google Play neither the application nor its meta-data is available nor downloadable from the application market anymore.

Altogether, any of these reasons involve rare and uncommon behaviors for application developers that makes them untrustworthy. In the case of policy noncompliance or plagiarism, it is straightforward as the application was doing something not tolerated by the market. Even in the case of developer removal, it seems strange for any developer to remove their own applications on purpose without further re-upload.

Hence, application removal is considered a proxy for CloneSpot validation. In essence, application removal is a clear sign of suspiciousness and those app-sets which contain removed application some time afterwards will have been able to detect rare applications that probably should not be in the market.

To verify application removal, we downloaded again the pages from applications falling into suspicious app-sets to check their market availability. Whenever an application is not found in Google Play, an HTTP 404 (not found) error code is returned, so by comparing the correct (code 200) vs not found responses from Google Play, we can easily infer which applications have been removed since September 2017.

We perform this experiment twice, in May 2018 and in October 2018 to keep track of application removal rates over time. As a result, out of $419,830$ applications discovered by CloneSpot in September 2017, we observe that $78,164$ have been removed from the market by May 2018, being $341,666$ potential clones left at that time. Nevertheless, by October 2018, the number of removed applications had risen to $218,621$, leaving just a total of $198,651$ applications. This last figure indicates that about one half of the applications detected by CloneSpot had been removed from Google Play after a year, showing that CloneSpot was able to detect many conflicting applications that have been eventually detected by Google Play afterwards.

Besides, it is worth noting that while application count has reduced to half its size, the number of app-sets containing at least an application is still large, confirming that the removal of applications has occurred to different app-sets.

Actually, the number of applications per app-set has been drastically reduced, to mainly one or two applications as compared to the three applications per app-set of September 2017 and May 2018. Fig. 7.3 illustrates the boxplots for app-set size that clearly shows this decrease by October 2018. Removed applications distribution is

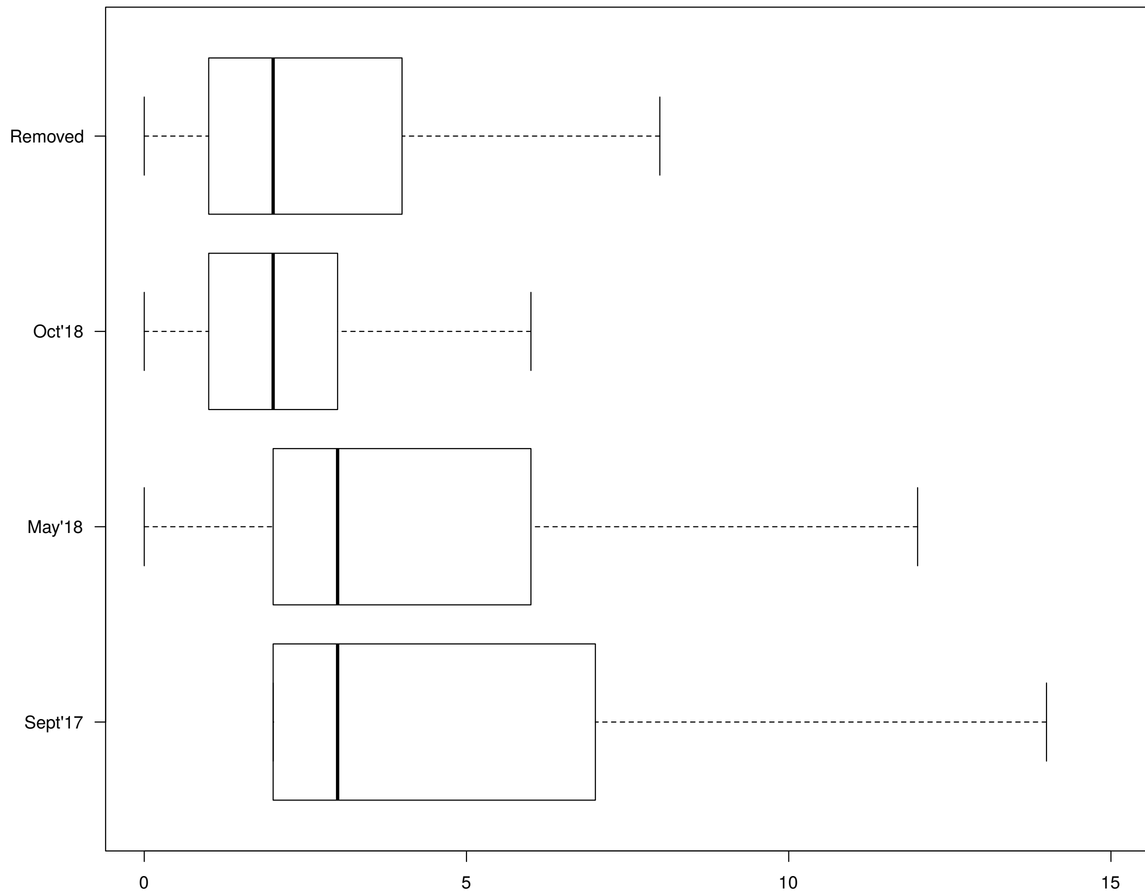


FIGURE 7.3: Boxplots of the distribution of app-set sizes. It can be observed a small reduction in May 2018 that becomes bigger as of October 2018.

depicted as well, showing that most repackaged applications belong to small app-sets (2-3 applications size).

In addition, approximately 50% of the app-sets (precisely 29,145 groups out of 54,944) have suffered application removal by May 2018 and even more, 47,887 (87%) by October 2018. These figures point to a removal trend that leaves legitimate applications alone within their app-sets as the clones are eliminated.

Indeed, the regular outcome for legitimate applications is to be left alone within their app-sets while potential clones or plagiarized copies slowly disappear from the market. Although application removal cannot be directly associated to plagiarism or repackaging, it can be observed that CloneSpot finds conflicting applications.

As an additional example, consider Fig. 7.4 which depicts another *"potential clone pair"* (Their Min-Hash value is $0x21a62ee1184c08b3d8a9efba058338808cedaac$), in this case, it is clear that both applications are exact copies which have just changed application logo and made slight modifications to the description and titles.

Indeed, the example belongs to a four application app-set which already lacked two applications at the time of writing. Its $ASI1$ score is 0.223, its edit distance is 4 and its cosine similarity 0.892. This application is still in the top 100, at position

FIGURE 7.4: Example of two similar applications detected by Min-Hashing

76. Furthermore, this example illustrates how the min-hashing approach is able to detect clones in different languages with no additional requirements.

Recall that results must still be manually inspected for verification, even though the min-hashing approach is capable of grouping similar applications fast and accurately. For further validation, we performed manual inspection over the unremoved cases in the top-100 applications in the $ASI1$ ranking. Out of 100 potential clone groups, 71 have lost at least one application whereas 29 app-sets remain unaltered. Hence, we performed visual inspection over the latter 29, finding 17 to look very much like clones attending to similarities in UIs, application structures, sizes and permissions. In other words, only 12 of the initial top-100 apps spotted by CloneSpot appear false positives in the detection.

As a result, CloneSpot is able to quickly detect, rank and sort accurately potential clones for manual inspection enhancing the analysis process by providing together and orderly victims and clones in the same application groups.

7.3.2 Real-time Repackaging Detection through the CloneSpot Service

The previous section has focused on the search for repackaged applications within an entire market such as Google Play using only qualitative meta-data fields. Even though this process is optimized and fast, it is not enough for online real-time application analysis, not to mention the requirements in computing resources to complete such operation.

However, the actual computation time of a min-hash, which is the basic requirement to index a single application, is really small, as the result of performing some operations over a few kilobytes of data. Since market analysis can be effectively

undertaken separately in an offline fashion and stored in a highly efficient database indexed by min-hash, it is possible to develop a near real-time clone retrieval system.

Consequently, CloneSpot can be converted into an application duplicate search engine that seeks for copies of incoming applications from a database that can be updated and extended in the meantime. In this light, *CloneSpot Poc* is a real-time querying engine of potential clones over their basic meta-data. When given any application title, description, category and developer name, CloneSpot can compute its min-hash and query the database by index to retrieve which applications, if any, are potential clones. Such application can be used by different actors within the Android ecosystem, for instance:

- From a market perspective, this PoC can be used to test newly updated applications at upload time and even require developers more details if meta-data analysis retrieves other applications.
- For analysts, the CloneSpot PoC can serve as a fast recommendation engine for applications to compare given a new sample.
- From an application developer perspective, CloneSpot PoC is a useful tool to keep track of similar applications and whether any application is being copied.

Implementation Details

The CloneSpot PoC is a REST API service built on top of Java Servlets. The underlying database is MongoDB due to its speed and high performance in largely loaded scenarios. The service is designed to query applications to the database using each app-set min-hash as an index for fast retrieval. The system is developed to return applications by directly receiving the min-hash value or, instead, the required meta-data fields to compute the min-hashing by itself. The methods implemented in the system are the following:

- *getDuplicates*: HTTP GET method that receives a min-Hash value in the request url and returns the app-set corresponding to that min-Hash.
- *analyze*: HTTP POST method that takes as input a JSON-encoded representation of application meta-data (title, description, category and developer name) and returns the app-set associated to the computed min-Hash value (if any).

In addition, the service provides an *info* method containing the user guide and usage considerations. The reader should bear in mind that this service is a Proof of Concept of the CloneSpot approach and is not prepared for full-load operation. Anyway, the service is publicly available for use over the dataset presented in this chapter [206]

7.4 Summary and Conclusions

In this chapter, we have deepened in the utilization of meta-data for Android application repackaging detection by proposing a more specific usage of two meta-information features: title and description. Using advanced textual clustering methods, namely min-Hashing, we have grouped applications with similar meta-data and ranked the resulting groups according to pairwise title and description similarity indexes.

This approach, that we have called *CloneSpot*, has been presented in Sections 7.2 and 7.3. CloneSpot is capable of grouping applications according to the similarity of their meta-data, so malware analysts can tackle their inspection in the correct context. Furthermore, aiming at a larger contribution to the community, we have opened the CloneSpot methodology to the Internet through a web service that given an application meta-data returns its potential clones from Google Play.

In sum, this chapter has extended the potential capabilities and uses of Android application meta-data in the context of malware as initiated in Chapter 4. Together, both chapters show Android meta-data advantages and potentialities in the field of malware detection that could improve Android security at scale.

Chapter 8

Optical WDM Networks Configuration from an ML perspective

From well-known classification algorithms capable of mimicking traditional protocol behavior, to most state-of-the-art developments like DL that have the potential of reinventing protocols and algorithms, AI technologies can be leveraged into more efficient state-aware network management and planning systems.

In this chapter we overview the application of AI technologies to optical WDM networks by methodologically defining an application framework with three main steps: (i) data collection, generation and labeling; (ii) problem solution, typically based on network protocol approximation with ML; and (iii) network implementation by means of novel networking technologies such as SDN.

Data collection and acquisition to support AI is a complex and difficult task in any area. Quality data is a scarce resource that many organizations have unavailable or are unable to manage adequately. In the case of networking data, data streams are only available to some few players, typically network operators and equipment vendors, who have not gained interest on its collection until very recently. Thus,

correctly acquiring, labeling and using networking data is a complex step that needs an inexistent systematization hitherto.

Once enough quality data has been amassed, network problems can be tackled from a data analysis perspective. Usual ML approaches are very restricted to classification and regression and, therefore, require domain expertise and advanced modeling capabilities to properly emulate network protocols. Besides, many networking problems are profoundly complex and require much effort just to define the correct classification/regression target along with performance metrics and expectations.

Finally, after modeling and training, ML models must be implemented into networks so they can actually be useful for the networking community. In this sense, the advent of technologies such as SDN and NFV has propitiated the use of ML to support quick decision-making and network modification at runtime. Hence, the natural evolution for a validated ML model is the implementation via these technologies.

Each of these phases presents its own problems and challenges. Through this Chapter, we provide PoC solutions to the two first modeling steps, that is, data generation and model definition. In Section 8.1 we introduce Netgen, our solution for generating and labeling network information at scale that relies on the well-known network planner *Net2Plan*. Then in Section 8.2 we propose a DL-based approach to solve well-known *Routing and Wavelength Assignment* (RWA) problem that can be applied to small networks. Section 8.3 details the results, specially in time and network efficiency of these approaches and Section 8.4 summarizes finding and remarks the most relevant conclusions.

8.1 Netgen: Automated Tool for Network Data Generation

The first step for applying ML to network problems consists on the acquisition and labeling of training datasets. Examples of useful data might include network topology observations, link loads, hop counts or Traffic Matrices (TMs) in any networking context. Such data can be either collected from real networks or synthetically generated following a mathematical traffic model. In any case, all these observations require a label to be useful which depends on the specific details of the network protocol or problem.

To generate these labels, there exist different tools supporting network modeling and ILP/heuristics solving. Regardless, most of these tools are conceived to be used by a human expert that needs to plan and manage the resources of a network using estimates based on expectation models. Due to this, many of them are not prepared to deal with the solutions of tons of combinations of traffic matrices and network topologies. In this light, we propose *Netgen* to overcome this problem and enable the generation of quality labeled data. Netgen is conceived a wrapper management system of a very well-known planning tool: *Net2Plan*.

8.1.1 Net2Plan: An Open-source Network Planner

Net2Plan [157] is a Java-based open-source tool designed for planning, optimization and performance evaluation of communication networks that provides a singular framework to solve many popular algorithms, including different types of routing, path protection or even RWA algorithms. Such framework facilitates the definition optimization routines over different network elements, the management of well-known solvers, such as GLPK or CPLEX, or even sub-optimal approximations by means of heuristic methods.

In its Graphical User Interface (GUI) version, Net2plan provides users with tools that aid in the design, modification and storage of network representations including topologies, traffic matrices, characteristics, etcetera. In addition, the tool ships with a *Command Line Interface* of the program to be interacted directly by a computer.

Given a topology and traffic matrix, Net2Plan can solve a number of classical networking problems, such as IP routing, Routing and Wavelength Allocation (RWA) in IP over optical WDM networks or optimal configuration of physical-layer in wireless networks, either using ILP formulations assisted by different well-known implementations or even custom-made heuristic algorithms developed with the help of the tool. This way, Net2Plan supports the development of customized optimization solutions that are either ILP-based or heuristic-based.

Net2Plan can be fed with topology and TM information through .n2p files following a schema-like XML syntax. Once in the application, solutions to the required algorithm are computed and results can be stored as a new n2p or an excel file.

8.1.2 Netgen Architecture

Netgen is developed around the command line interface of Net2Plan: Net2Plan-CLI, which enables the execution of multiple instances for solving TMs at scale. Fig 8.1 drafts the basic Netgen workflow and components. In what follows, each of these components will be described and summarized. Netgen source code is publicly available at Github [207].

Traffic Matrix Generation

The first Netgen module is the *Traffic Matrix generator*, which is coded in the python script *TMGenerator.py*. This script takes as input a *canonical* Net2plan file, containing the network topology and demands, and generates random traffic matrices with different features to be applied on such network topology. The module corresponds to the initial phase (Data generation) in the workflow diagram in Fig 8.1.

When invoked, the program reads the topology and traffic matrix from the input file, modifies the canonical matrix as many times as required and stores the resulting matrices in a new n2p file interpretable by Net2Plan. Random noise is introduced through statistical distributions centered in the given canonical value and

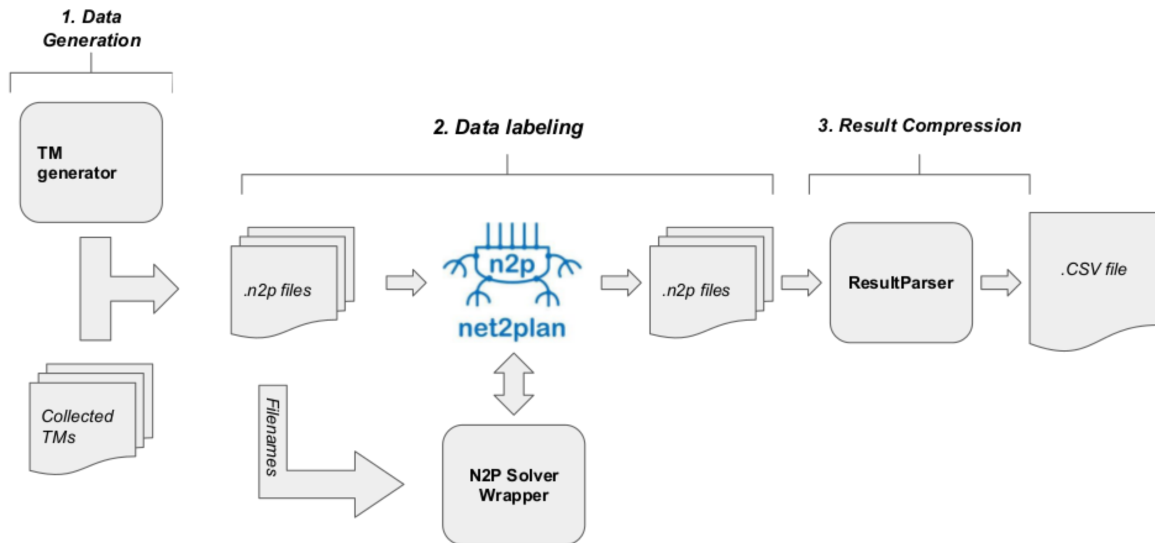


FIGURE 8.1: Netgen diagram overview: Data generation, labeling and result compression

configurable parameters. At present Gaussian and Uniform noise distributions are supported.

Netgen also provides an interface to load user-specific traffic matrices in Comma Separated Value (CSV) format. This component is encoded in the *TMLoader.py* script file.

Net2Plan Solver Wrapper

The Net2plan Wrapper manages a pool of Net2Plan instances that are invoked upon need to solve sample traffic matrices passed as input. Every time a TM is solved, it is stored into a new n2p file in the output directory. Specific algorithmic parameters required by Net2Plan are specified in a separate configuration file.

In fact, this module has the ability to invoke several instances of Net2plan in parallel to speed up algorithm resolution. The parallelization parameters of Net2plan are configurable in the program depending on the computing capabilities of the host system. The python script is *N2PSolveMulti.py*.

Net2Plan Result Parser

After resolution, Net2Plan results are stored in a folder in *n2p* file format. The final stage of Netgen comprises the collection of all output .n2p files and production of a tabular dataset more usable in ML algorithms. This is accomplished by the *ResultParser.py* script. Essentially, this python module collects the input and output n2p files and combines them into a single CSV file which columns include the input features (Traffic Matrix, etc) along with the output values depending on the specific networking algorithm executed. Each row in the file refers to a different n2p file.

Each solver requires its own parser due to the particularities of the underlying algorithm. For instance, if the algorithm executed in the second phase is the classical RWA algorithm, then the parser should collect the wavelength and sequence of links traversed by each lightpath.

The current version of Netgen includes parsers for three Net2Plan algorithms: *Routing and Wavelength Allocation*, *Path Formulations* (Routing) and *Link Disjoint Path Protection routing*.

8.2 Modeling RWA as an ML classifier

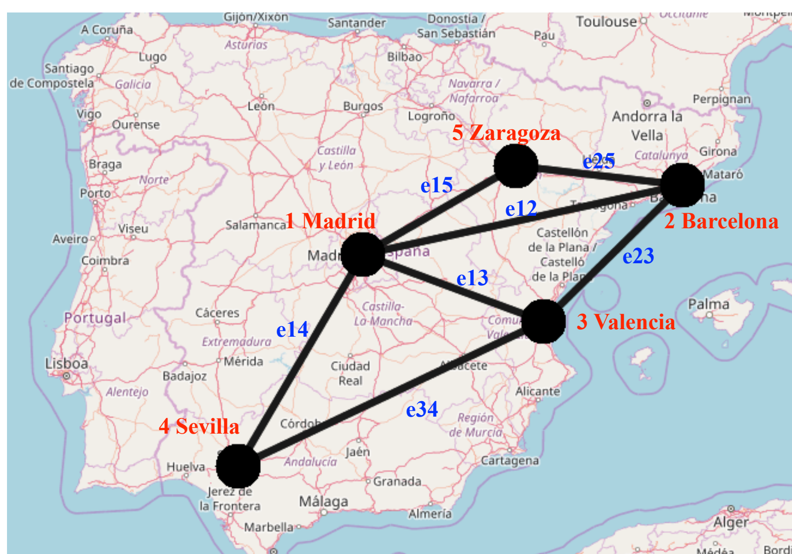


FIGURE 8.2: Depiction of the basic 5 Node network used for this problem

In this section we demonstrate the second phase of our framework by the example. Concisely we focus on a very complex problem: The *Routing and Wavelength Allocation* problem (RWA for short), which attempts to optimally assign lightpaths (routes and wavelengths) to each of the demands within a network.

Routing and Wavelength Assignment in optical networks is typically modeled as a multi-commodity flow problem, where a directed graph is used to represent the optical network topology. Given a set of demands in traffic matrices (TMs) for the given topology, the resolution of the RWA consists on determining the optimal route and wavelength combination that to satisfies every given demand.

Generally, RWA formulations include a set of constraints the resolution must be subject to, usually regarding *flow conservation*, *capacity constraints* or *wavelength usage*, notwithstanding other elements like *wavelength continuity* or *maximum reach*.

Typical objective functions are varied, ranging from minimizing the number of wavelengths used, the number of transponders or the energy consumption. In the

past, RWA has been solved either through complex, non-scalable exact mathematical models based on Integer Linear Programming (ILP) or different heuristic algorithms proposed to speed up resolutions specially in large networks. ILPs have been proved NP-hard, meaning the problem is computationally costly to solve and can take way too long. To alleviate this, heuristic functions that provide sub-optimal solutions have been designed. For a more comprehensive overview of RWA solution approaches, the reader may refer to [208].

In this light, we propose an ML approach that relies on ILP-solved data to train a sub-optimal heuristic-like model capable of learning from ILP patterns, minimizing the suboptimality of heuristic approaches whilst reducing the computation times of both ILP and heuristic approaches. For this problem, we consider a version of RWA with no wavelength conversion and unconstrained route length. Concerning the optimization target for the ILP, we attempt to reduce overall network cost, that mainly takes into account network transponders.

8.2.1 Methodology

For illustration purposes, consider the 5-node network topology of Fig. 8.2 with five nodes and seven bidirectional links and assume a number of W wavelengths per link, each one operating at C Gb/s (i.e. W lambdas @ C Gb/s).

Let us also consider a 5×4 traffic matrix collecting the traffic demands d_{ij} (in Gb/s) from source i to destination node j ($i \neq j$). The RWA algorithm maps each traffic demand d_{ij} to a list of link sequences (routes) and wavelengths (lambdas). For example the network could contain a formulation like the following:

$$
\begin{aligned}
d_{12} &\rightarrow (e_{12}, \lambda_1) \\
d_{13} &\rightarrow (e_{13}, \lambda_1) \\
&\vdots \\
d_{24} &\rightarrow (e_{23}, e_{34}, \lambda_2) \\
&\vdots \\
d_{54} &\rightarrow (e_{52}, e_{23}, e_{34}, \lambda_3)
\end{aligned}
\tag{8.1}
$$

which provides different network routes and wavelengths for each demand. In this case, demand d_{12}, going from source node 1 to destination node 2 and is provisioned with direct link e_{12} and the first wavelength λ_1. Also, demand d_{54} goes through the route defined by links $e_{52} - e_{23} - e_{34}$ and uses the third wavelength λ_3. We call this formulation a *Routing and Wavelength Configuration* (RWC) and is suited for a particular demand matrix. In other words, the RWA solver receives as input a serialized traffic matrix with all source-destination traffic demand requests and outputs its optimal RWC list.

When the ILP is given a sample traffic matrix (TM_1), it will produce a similar set of links, $y_1 = RWC_1$. Similarly, upon another new matrix TM_2, the ILP will

produce a different configuration $y_2 = RWC_2$, and so on for all available matrices. Consequently, the RWA problem can be transformed into a multi-class classification problem where the input variables are the elements of a serialized traffic matrix and the output labels are network *RWCs* obtained from solving the ILP.

In many cases, a given *RWC* can correctly satisfy different TM demands at the expense of a minimal penalty on some link loads and hop counts producing slightly suboptimal results. Additionally, reducing the total number of classes involved in a classification problem is key to boost performance. Hence, aiming at the reduction of complexity, a threshold of 30% average hop count will be introduced to facilitate the reduction of classes. For that purpose, we combine TMs and *RWCs* over a forward-pass over the latter to compress the number of classes involved in the solutions.

To perform such pass, we verify that the *RWCs* assigned to new TMs comply with the previously defined tolerance condition as well as with producing a *feasible* *RWC* for the new TM. In this context, any *RWC* is considered *feasible* for any TM provided the following conditions are met:

- All traffic demands are allocated meeting the wavelength continuity constraint, that is, every lightpath traverses the network in the optical domain, without changing the wavelength
- All link occupations (link loads) are possible, that means having an occupation of 100% at most.
- Each slot formed by a link and a wavelength is used only once.

The outcome of these transformations may be used to train a classical supervised ML that learns how to replicate the ILP behavior forming a heuristic algorithm that is trained over optimal and reliable output data to make faster predictions over new TMs. Furthermore, in case no ILP solution was available, such a model could also learn from *RWC* datasets solved using heuristics, since the class scheme would be similar.

8.2.2 Dataset Generation and Labeling

The experiments in this section are carried over the two networks, a simple 5-node Spanish topology as a the proof of concept and the well-known Abilene network topology to illustrate real-data and scalability.

The 5-network topology proposed in Fig. 8.2 is composed by 5 nodes and 7 bidirectional links (14 links in total). Transponders are the assumed equal in all nodes and the number and capacity of wavelengths is the same. Each node is located in one of the biggest cities in Spain, namely Madrid, Barcelona, Valencia, Zaragoza and Sevilla; the distance (in kilometers) is computed from the real coordinates and the population associated to each node is that of their home cities. We have generated $10,000$ TM samples for this network following a well-known traffic model: population-distance.

The population-distance model (aka gravity) assumes that the traffic demands between each two cities depend on their size and distance [209]. This base model is altered with random white Gaussian noise centered at 100 times the average population distance model μ_{dist} and with a standard deviation of 0.15 times such average for generating the required TMs. In short:

$$Noise \approx \mathcal{N}(\mu_{dist}, 0.15 \times \mu_{dist}) \tag{8.2}$$

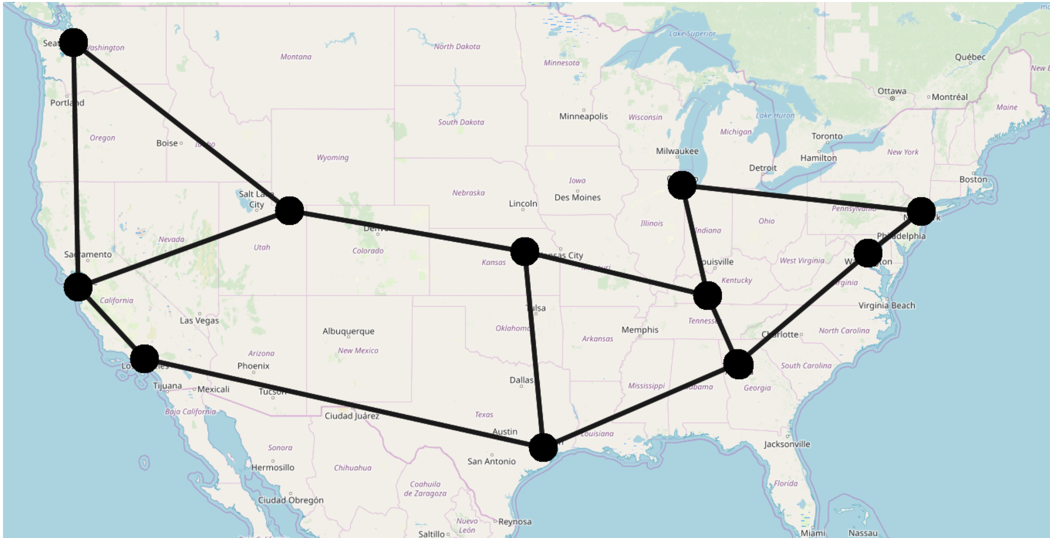


FIGURE 8.3: Schematic of the Abilene Network

In addition, we consider the Abilene network topology aiming at validating the model with real network data and and verifying the scalability of the approach. The Abilene topology, depicted in Fig 8.3, is composed by 12 nodes and 15 bidirectional links (30 in total), which makes it a larger and more complex network setting for the ILP. Furthermore, we make use of 48, 096 real traffic matrices collected in a timespan of six months and available online through the SNDlib project [159].

Then, we solve the available traffic matrices using Netgen for both topologies with different configurations. Concisely in the 5-node topology, we consider four different transponder configurations, namely 8 and 10 wavelengths at 40 Gbps and 5 wavelengths at both 100Gbps and 400Gbps (namely, *5@400, 5@100, 8@40 and 10@40* datasets).

In the case of Abilene, the network configurations are 20 wavelengths at 400 Gbps and 20 and 40 wavelengths at 100 Gbps (that is, *20@400, 40@100 and 20@100* datasets). In addition, we solve the same cases of Abilene for the heuristic algorithm provided with Net2Plan, which is a frist-fit algorithm. As ILP solver, we have configured Net2Plan to work with IBM's *CPLEX* implementation [210].

8.2.3 Machine Learning Classification Models

Classification algorithms might be used to estimate the probability of an *RWC* to be feasible for an specific TM. This way, rather than predicting a single *RWC*, we can create a ranking for most likely suited *RWCs* given a TM and keep the 10 topmost to minimize the number of unfeasible *RWCs* at a limited time cost. In other words, ML models are set up to produce class probability estimates and consider the top 10 most likely *RWCs*. In the case where no result provides a feasible solution for a TM as defined above, that particular sample is marked as unfeasible.

This way, ML algorithms will predict the scores of the 10 most probable *RWCs* and the resulting system will choose from the list in order the first feasible case. In this light, we choose Logistic regression and Feed-forward Deep Neural Networks as classifiers, as they are both good at estimating probabilities. Moreover, the LR classifier can deliver a baseline performance estimation whereas the DNN can work on classification with state-of-the-art performance.

Logistic Regression Architecture

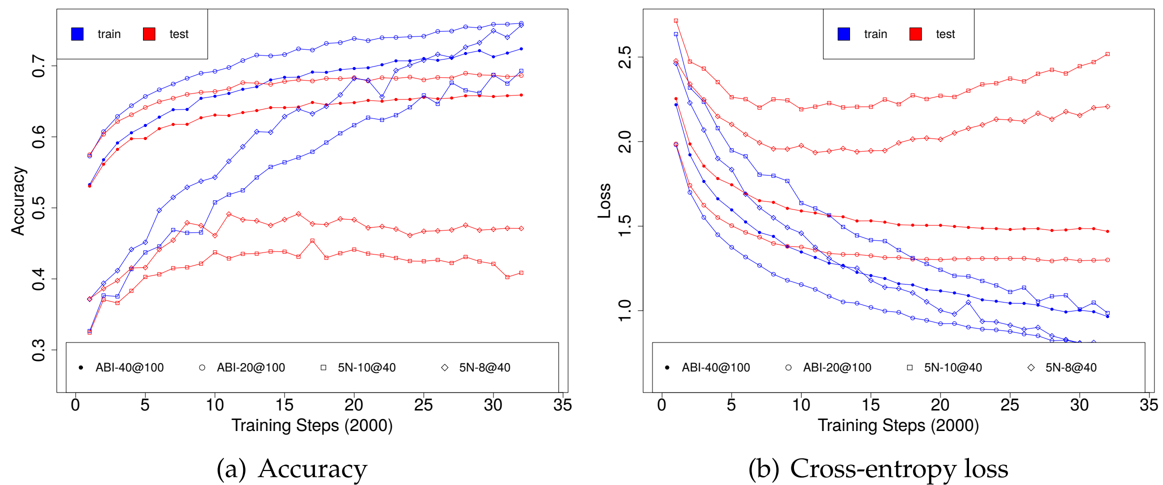
Logistic Regression classifiers are very fast to train and interpret but prone to underfitting due to their simple linear nature. We train two logistic regression models with different regularization schemes each: lasso (ℓ_1) and ridge (ℓ_2). The regularization constant (C) of each model is determined with the help of 10-fold cross-validation, where train data is split into 10 chunks and each is used once as testing set with different candidate parameters. A *hold-out validation* set is kept aside for final validation.

Deep Neural Net Architecture

The proposed DNN configuration comprises six fully connected layers, with *dropout* in the first and fourth layers and ℓ_2 *regularization* in the third and fifth layers. Activation is performed using the well-known rectified linear unit (*ReLU*) and hyperbolic tangent (*tanh*) functions. The model has been trained using the Tensorflow framework, concisely the Stochastic Gradient Descent implementation.

The optimization target is the minimization of the output's cross-entropy over $16,000$ training steps that perform an optimization batch of 400 datapoints randomly sampled. The learning rate is 0.02. In this case, regularization, dropout and the rest of hyper-parameters have been set by heuristics, validated through a hold-out set reported separately (validation). In addition, we have studied the number of steps and overall architecture validation by testing the training of the model through several steps.

Indeed, Fig. 8.4 depicts the evolution of accuracy and cross-entropy over training steps for training and test sets for the two network scenarios (5-node and Abilene). Each of the points in the horizontal axis represents a group of $2,000$ training steps The figure demonstrates how the test curve plateaus in between $13,000$ and $19,000$



(a) Accuracy (b) Cross-entropy loss

FIGURE 8.4: Accuracy and Cross-entropy loss for the 5-node and Abilene network topologies.

steps while the training accuracy curve continues growing, suggesting that using more steps would incur in overfitting.

This figure further shows that the proposed DNN is valid and useful for both topologies and all the proposed configurations, which in turn suggests good generalization capabilities for the case of RWA.

Performance Assessment

Throughout experiments, F-score values are collected for train, test and validation sets to evaluate performance in terms of ML classification metrics. However, the RWA problem is not strict and different solutions may be applied to different TMs provided they satisfy demands and requirements correctly.

Thus, the account of *feasible* results which may not be necessarily identical to the given result, is collected. Furthermore, the average link load and hop count of the proposed solution is also registered for reference. This way, each model's generalization abilities can be measured in terms of ML and also its specific network parameters. At the end, network performance metrics are useful to determine the capabilities of classification algorithms to produce a similar *RWC* to the one computed by the ILP that leverages ML for better approximations than heuristics.

8.3 Results

8.3.1 Dataset Generation Efficiency of Netgen

Using Netgen, six 100, 000 sample datasets have been solved for the three networking problems currently supported by Netgen, namely (i) conventional shortest path IP Routing, (ii) shortest path IP routing with 1+1 link-disjoint protection, and (ii)

RWA. For completeness, we compute such values in two networks: (i) a small 5-node Spanish topology and (ii) the bigger NSFnet topology. Table 8.1 shows time figures separated by component[1]. These results have been measured from the execution of Netgen in an Intel Xeon E5-2630 server with 24 cores and 190 GB of RAM memory.

TABLE 8.1: Netgen in action: Time performance for labeling datasets

Algorithm Topology	IP Routing NSFNet	1+1 Prot NSFNet	RWA NSFNet	IP Routing 5Spain	1+1 Prot 5Spain	RWA 5Spain
Data Generation	454 secs	454 secs	454 secs	180 secs	180 secs	180 secs
Data Labeling	~15 hours	~94 hours	~10 hours	~4 hours	~14 hours	~3 hours
Result Compression	126 secs	250 secs	178 secs	19 secs	22 secs	39 secs
Average per sample	0.54 secs	3.38 secs	0.35 secs	0.12 secs	0.91 secs	0.109 secs

As shown, most of the time spent in the generation of the dataset corresponds to the actual optimization solution of the algorithm, which takes several seconds for complex formulations and large network topologies.

8.3.2 ML-based Heuristic to Solve RWA (Spanish 5 Node topology)

TABLE 8.2: Ml and networking metrics of the proposed solution for the different datasets and algorithms

DATASET	d1: 5@400									
Algo.	$Train_{Fsc}$	$Test_{Fsc}$	Val_{Fsc}	$\bar{\rho}$	max_{ρ}	$\bar{hops}$	max_{hop}	F_{train}	F_{val}	RWC
ILP	-	-	-	0.1302	0.5097	2.05	4	100	-	-
ℓ_1	0.613	0.604	0.606	0.1288	0.5092	2.05	4	99.9	99.9	15
ℓ_2	0.616	0.600	0.604	0.1292	0.5097	2.05	4	99.9	99.9	15
NN	0.799	0.809	-	0.130	0.510	2.05	4	100	100	15
DATASET	d2: 5@100									
Algo.	$Train_{Fsc}$	$Test_{Fsc}$	Val_{Fsc}	$\bar{\rho}$	max_{ρ}	$\bar{hops}$	max_{hop}	F_{train}	F_{val}	RWC
ILP	-	-	-	0.358	1	1.775	3	100	-	-
ℓ_1	0.549	0.540	0.542	0.316	1	1.6	3	99.05	99.30	69
ℓ_2	0.545	0.511	0.542	0.369	1	1.775	3	62.22	61.30	69
DNN	0.954	0.949	0.952	0.36	1	1.77	3	99.98	99.88	69
DATASET	d3: 10@40									
Algo.	$Train_{Fsc}$	$Test_{Fsc}$	Val_{Fsc}	$\bar{\rho}$	max_{ρ}	$\bar{hops}$	max_{hop}	F_{train}	F_{val}	RWC
ILP	-	-	-	0.503	1	1.895	4	100	-	-
ℓ_1	0.399	0.327	0.361	0.496	1	1.87	3	85.37	83.80	215
ℓ_2	0.395	0.318	0.295	0.4925	1	1.855	4	80.10	81.00	215
DNN	0.815	0.801	0.808	0.49	1	1.86	3	99.45	97.16	215
DATASET	d4: 8@40									
Algo.	$Train_{Fsc}$	$Test_{Fsc}$	Val_{Fsc}	$\bar{\rho}$	max_{ρ}	$\bar{hops}$	max_{hop}	F_{train}	F_{val}	RWC
ILP	-	-	-	0.554	1	1.648	3	100	-	-
ℓ_1	0.400	0.330	0.350	0.450	1	1.871	3	81.45	82.00	197
ℓ_2	0.390	0.329	0.319	0.506	1	1.777	3	85.32	85.80	197
DNN	0.851	0.837	0.854	0.52	1	1.63	3	99.63	96.88	197

[1]The resulting datasets are publicly available at: https://osf.io/et2ga/

Tab. 8.2 provides a detailed summary of experimental results for each dataset in the 5-node network. The Net2Plan ILP result metrics are reported as well. Train, test and validation F-score values are given along with networking metrics such as average link load ($\bar{\rho}$) and hops ($\bar{hops}$), bottleneck link load (max_{ρ}) and maximum hop count (max_{hop}) and the amount of feasible solutions produced for the training (F_{train}) and validation (F_{val}) sets.

As expected, machine learning scores are not optimal, with the exception of the neural network, which provides acceptable F-score values, being very good in some cases. Anyway, network metrics indicate very performance in terms of feasibility, as most of the assigned *RWCs* are possible even in many cases where the classification fails in a strict sense. Here, the DNN is also the clear winner, showing very high feasibility and with comparable link load and hop count with respect to the optimal solutions from the ILP. Indeed, neural networks achieve almost complete feasibility, reaching above 95% in all cases.

On the other side, LR results are worse, specially in terms of ML scores (F-score). Despite, feasibility is high as well, with ℓ_1 regularization usually beating ℓ_2. This results suggest that RWA can be approximated with ML, being more complex models better than simple formulations.

8.3.3 ML-based Heuristic to Solve RWA (Abilene topology)

TABLE 8.3: Numerical results obtained for the Abilene network topology

DATASET	d5: 20@400									
Algo.	$Train_{Fsc}$	$Test_{Fsc}$	Val_{Fsc}	$\bar{\rho}$	max_{ρ}	$\bar{hops}$	max_{hop}	F_{train}	F_{val}	RWC
ILP	-	-	-	0.025	0.421	2.64	6	100	-	-
heur	-	-	-	0.029	0.422	3.007	8	100	-	-
DNN_{ILP}	0.886	0.883	0.884	0.025	0.416	2.643	6	97.77	98.08	1,470
DNN_{Heur}	0.987	0.984	0.979	0.028	1	3.007	8	99.92	99.86	133
DATASET	d6:40@100									
Algo.	$Train_{Fsc}$	$Test_{Fsc}$	Val_{Fsc}	$\bar{\rho}$	max_{ρ}	$\bar{hops}$	max_{hop}	F_{train}	F_{val}	RWC
ILP	-	-	-	0.059	1	3.064	7	100	-	-
heur	-	-	-	0.047	1	2.503	5	97.25	-	-
DNN_{ILP}	0.748	0.706	0.712	0.057	1	3.05	7	94.67	94.65	1,775
DNN_{Heur}	0.456	0.423	0.422	0.047	1	2.503	5	85.07	84.38	5,868
DATASET	d7:20@100									
Algo.	$Train_{Fsc}$	$Test_{Fsc}$	Val_{Fsc}	$\bar{\rho}$	max_{ρ}	$\bar{hops}$	max_{hop}	F_{train}	F_{val}	RWC
ILP	-	-	-	0.111	1	3	6	100	-	-
heur	-	-	-	0.093	1	2.503	5	84.33	-	-
DNN_{ILP}	0.771	0.731	0.725	0.110	1	3.003	6	95.55	94.87	1,462
DNN_{Heur}	0.425	0.389	0.389	0.09	1	2.503	5	71.33	71.41	11,128

Table. 8.3 shows the results obtained for the DNN in the Abilene network topology assuming different capacity and wavelength configurations. In this case, we illustrate the possibility of training a DNN with the output result of both optimal ILP and first-fit heuristic algorithms. We observe that in all ILP cases, the number of *RWC* classes is about $1,500$ for $48,096$ training points, and the results are very good

both in terms of F-score and network-related metrics (link load and hop count). Concerning feasibility, values around 95% are obtained in all DNN_{ILP} cases.

Therefore, results illustrate how using ILP-based data as training ground truth is better than using heuristic-based data, since the solutions achieved by the former are more compact than those of the latter. Interestingly, learning from heuristic-based solutions in harder, since the number of classes is not easily reduced and therefore the correct training of the DNN is not easy. Nonetheless, feasibility is still high and above 70% in all cases, indicating than any of the two approaches can yield acceptable performance.

In conclusion, this experiment shows that DNN models can learn from both data sources, even though ILP-based solutions appear to be more appropriate for obtaining improved results. However, it is worth noting that the final number of classes is relevant and affects the requirements for data samples.

8.3.4 Complexity and Time Impact of ML Predictions

The time complexity and impact of the proposed solution is highly dependent on the classification time, the comparison with the ILP and the number of attempts each TM requires to get the correct solution. Whenever any ML solution is not feasible and cannot be used, it is necessary to extract the *RWC* from an alternative method, such as the ILP or increasing the top ranking.

Hence, aiming at estimating the time impact of the system, we compare ML completion times with the ILP computation times, using as ML times upon failure the completion time of the ILP. This way, when the ML solution fails, it gets penalized to the full time required to obtain the ILP solution.

As a result, this system comprises a hybrid ML-ILP solution that always assigns an appropriate solution. Average completion time of an ML estimation is below 15 *ms* for any model, whilst the time required by Net2Plan solver for the ILP or even the first-fit heuristic accounts for hundreds of ms on average. Then, the time improvement over the ILP solution can be computed as follows:

$$\text{Impr} = \frac{t_{\text{ILP}} - (N_{\text{feasible}} * t_{\text{ML}} + N_{\text{unfeasible}} * t_{\text{ILP}})}{t_{\text{ILP}}} \tag{8.3}$$

where t_{ILP} and t_{ML} correspond to the time used by the ILP and the ML classifier respectively and N_{feasible} and $N_{unfeasible}$ determine the amount of unfeasible cases.

Table 8.4 illustrates the specific completion times for model training and prediction, along with an estimate of the time cut produced by ML. Besides, the time proportion imputed on average to feasible (ML) and unfeasible (ILP) solutions is displayed as well. Figures in the table do show that higher link load network settings take more time to train and predict. Additionally, the reader should note that more complex ML models are more accurate, but slower, even though they are still way faster than traditional heuristic of ILP approaches.

The table indicates that LR models are fast to train and query for prediction,

TABLE 8.4: Training and prediction times for the ML solution. In addition, the improvement of a hybrid ML-ILP solution with respect to pure ILP is shown as well.

5-Node ℓ_1 LR				
Dataset	Train.	t_{ML}	t_{ILP}	Impr.(ILP)
5@400	2.0 *s*	0.6 μs	478 *ms*	99.89%
5@100	8.4 *s*	1.6 μs	393 *ms*	99.29%
10@40	21.0 *s*	2.6 μs	436 *ms*	83.79%
8@40	19.6 *s*	2.3 μs	470 *ms*	81.99%
5-Node ℓ_2 LR				
Dataset	Train.	t_{ML}	t_{ILP}	Impr.(ILP)
5@400	2.2 *s*	0.7 μs	478 *ms*	99.89%
5@100	2.8 *s*	8.1 μs	393 *ms*	61.22%
10@40	2.8 *s*	3.7 μs	436 *ms*	80.99%
8@40	3.7 *s*	2.2 μs	470 *ms*	85.79%
5-Node ILP DNN				
Dataset	Train.	t_{ML}	t_{ILP}	Impr.(ILP)
5@400	15.1 *mins*	1.1 *ms*	478 *ms*	99.81%
5@100	15.5 *mins*	1.6 *ms*	393 *ms*	99.63%
10@40	16.5 *mins*	3.3 *ms*	436 *ms*	96.74%
8@40	16.3 *mins*	1.8 *ms*	470 *ms*	96.65%
Abilene ILP DNN				
Dataset	Train.	t_{ML}	t_{ILP}	Impr.(ILP)
20@400	17.2 *mins*	8.8 *ms*	861.8 *ms*	97.08%
40@100	16.5 *mins*	10 *ms*	1382.1 *ms*	93.96%
20@100	18.2 *mins*	3.5 *ms*	1899.2 *ms*	94.69%
Abilene Heuristic DNN				
Dataset	Train.	t_{ML}	t_{heur}	Impr.(heur)
20@400	12.3 *mins*	2.6 *ms*	256 *ms*	98.90%
40@100	21.5 *mins*	5.9 *ms*	237 *ms*	82.28%
20@100	33.6 *mins*	14 *ms*	262 *ms*	67.60%

while the DNN models require some minutes to train and milliseconds for prediction. Still, both approaches are below the hundreds of milliseconds required by the ILP to optimally solve a given traffic matrix. Moreover, linear ML models may be faster than DNN but they often report more unfeasible cases that require a full ILP cycle for a feasible *RWC* penalizing the models' time performance. Hence, DNN models achieve better results mainly due to their higher feasibility scores.

Specifically, LR produces results with feasibility around 70-80% and a potential time cut with respect to the ILP of 80% whilst the DNN is able to achieve complete feasibility in almost every case and producing time improvements of, at worst, 94% with respect to the ILP. In the case of Abilene topology, linear models have not been attempted due to the increase in complexity of the network, which would produce worse performance.

Finally, it is worth noting that the proposed ML solution is also faster than the reported times of the first-fit heuristic, making it an interesting replacement for both alternatives. In fact, while ML can be trained from any of the solutions, it is clear that ILP-based ML is better in terms of feasibility. Thus, an appropriate approach to produce ML-based RWA would be to use ILP as often as possible and using the heuristic-based training when ILP solution is not available or computationally feasible.

8.4 Summary and Conclusions

Through this chapter, we propose and validate an ML application framework in the context of optical WDM networks driven by an example problem: Routing and Wavelength configuration assignment. The framework for ML application is formed by three main stages: *data generation and labeling*, *modeling and validation* and *system implementation*. This chapter focuses on the development of the former two, while the latter is future work currently undergoing as a collaboration of our UC3M team with *Politecnico de Milano*.

Moreover, we have investigated solutions for the data collection and model validation phases as proofs of concept. For data generation and labeling, we presented Netgen which interfaces the Net2plan network planner tool to generate or label training datasets for networking. Regarding modeling and validation, we attempted the approximation of the complex problem of Routing and Wavelength Allocation through ML. Modeling was developed to adjust the problem statement to a classification problem where entire routing and wavelength configurations (*RWCs*) are classes to be assigned from different TMs as input.

Concisely, we used logistic regression and Deep Neural Networks with a ground truth dataset of thousands of traffic matrices and their associated RWA solutions provided by the RWA ILP or first-fit heuristic. Results indicate great potential for this approach, specially in terms of time reduction and improved resource management.

In particular, the proposed DNN architecture has shown useful learning nonlinear structures successful in identifying the relations between TMs and *RWCs*. Actually, the feasibility provided by DNNs is over 90% for the ILP-based cases and over 70% with heuristic-based data. Moreover, the algorithm achieves a time cut of at least 93% with respect to classic ILP approaches. The data used in the analysis has been synthetic in the case of the 5-node network and collected in the case of Abilene network.

In sum, this chapter has demonstrated the viability of approximating network protocols like RWA using ML to create ILP-mimicking algorithms that provide suboptimal solutions better than those of heuristics and much faster than both alternatives.

Chapter 9

Conclusions and Future Work

This chapter recaps the main contributions and advancements of the thesis and concludes the work summarizing conclusions, lessons learned and future work. In Section 9.1 we summarize conclusions and results from the different contributions from each chapter while in Section 9.2 the main future lines are enumerated. Finally, in Section 9.3 we facilitate a detailed list of the publications arisen during the PhD period.

9.1 Summary of Main Contributions

The aim of this thesis is the advancement in the adoption of AI and ML through applying techniques from these technologies into two distinct fields related to the Internet, namely cybersecurity and pptical WDM networks. Along different lines and incremental contributions, this thesis has proposed various enhancements, typically software components, which improve the detection of Android malware, the

knowledge and combination of multi-scanner AV detections and the usage of ML in optical networks.

In Chapter 4 we have presented meta-data, that is, that information not present in binaries but describing application functionalities and features. Meta-data has been proposed in the chapter as a detection vector for Android malware. Using different meta-information fields, we have developed malware detection systems using Machine Learning over labeled Android samples. This classification has been successful, achieving up to 0.89 F-score in general malware detection. Specifically, this chapter reaches the following conclusions:

- The analysis of application permissions in the context of Android malware detection is useful, but can only offer moderate results.
- Other features publicly available in application markets, such as developer or issuer names, are more relevant to detect malware, being reputation-based features the most effective ones.
- Using meta-data alone it is possible to create compact and efficient classifiers to detect Android malware from each app market information.

In the light of the problems arising from AV engine detections in malware analysis, we studied in Chapter 5 the outputs of multi-scanner tools coming from a large collection of suspicious applications with the aim of clarifying the concept of malware. This analysis demonstrated different behaviors observable in engines with a clear detection pattern structure and resulted in contributions useful for the enhancement of malware risk assessment. In detail, the main contributions from this chapter are the following:

- We hypothesized and unveiled through data-driven analysis three patterns that can be found in AV engines performing detections, namely *leaders*, *followers* and *eccentrics*.
- AV engines have been shown to have surprisingly low correlation values and, therefore, many engines have been needed to inspect all possible behaviors and patterns.
- A new method to estimate malware risk of a sample based on each sample's AV detection was proposed. Such method relies on SEM and the logistic function to weight AV engines according to their inter-relations.

Moreover in Chapter 6, the *SignatureMiner* tool has been proposed for detection signature normalization. SignatureMiner has enabled further family-based analysis leading to a category classifier solely relying on AV binary detections. In sum, Chapter 6 has contributed to this thesis with the following:

- The SignatureMiner tool for normalizing AV engine detection signatures has been developed and shows performance similar to that of its competitors in performance whilst lightweight and versatile. The tool has been shared publicly and is available at Github.

- Using SignatureMiner, the most relevant malware families have been extracted from a collection of 80K applications and their detection signatures. From there, applications have been aggregated into three possible categories, namely: *adware*, *harmful* and *unknown*. Further analysis on families and categories has provided inter-relation insights and indicators of consistent category information to unknown samples.

- Using the above scheme, an ML classifier is proposed to determine whether a sample application belongs to adware or harmful categories. This classifier uses a random forest classifier over AV detections and achieve an F-score of 0.92. This classifier has been used to unveil the actual category of the *unknown* category samples.

The last contribution regarding cybersecurity, in Chapter 7, has demonstrated how meta-data can be used to unveil repackaged applications by solely using application titles and descriptions. Concisely, CloneSpot has been able to identify nearly 420K application clones out of 1.3M apps. From these 420K applications approximately 200K have disappeared from Google Play a year after analysis following a removal of clones pattern. The main remarks from this chapter are the following:

- The CloneSpot methodology to analyze applications' similarity at market level is proposed and utilized. Such methodology relies on min-hashing over application's meta-data to cluster similar applications together.

- CloneSpot performance is evaluated through removal statistics, showing that a year after 50% of the applications detected by CloneSpot have been removed from Google Play.

- As a Proof of Concept, we produced CloneSpot PoC, a service that is capable of returning potential application duplicates on real time using the CloneSpot methodology. This application is running at the time of writing and can be accessed online.

On the network side, Chapter 8 proposes and explains in detail a three-stage framework for the application of ML to network problems that comprises *labeled data generation*, *modeling and validation* and *system implementation* by means of novel technologies such as SDN. Concisely in this chapter, we focus on the first two stages. For the first stage, we developed the Netgen system for data generation and labeling at scale.

For the second stage, we addressed the classical problem of Routing and Wavelength Assignment in optical networks as a PoC. While the proposed system is not perfect in terms of ML, it shows an impressive performance by approximating the RWA optimal ILP and producing *RWCs* that may may fit many input TMs at once showing comparable performance. The last stage is left as future work and will require the implementation of previously trained models into real networks. To sum up, Chapter 8 contributions are listed below:

- The proposal of a system-wide framework for the application of ML in computer networks, specifically in the context of optical WDM networks.

- We develop a tool for data generation called Netgen, which leverages the planning tool Net2Plan to generate labeled networking datasets at scale.
- An ML-powered heuristic solution for RWL is proposed. This algorithm works by emulating the optimal solution provided by an ILP using ML. Experiments show such heuristic is comparable to ILP and well-known heuristics while much faster to compute.

Contributions have demonstrated that AI has a large potential in these fields and showed how difficult and complex problems can be simplified through AI and, specially how AI technologies provide the tools to propose out-of-the-box solutions different to anything seen before. Finally, part of the methodology applied throughout different parts of this thesis can be used as the initial building block to apply AI technologies similarly in other fields.

9.2 Future Work

Broadly, the main line of work for this thesis is to continue with the design and development of intelligent components to apply AI and ML into these and other fields. Regarding the specific contributions in the thesis, each one may be continued from two perspectives: (i) incremental extension using larger datasets or more complex models and (ii) functional extensions specific to each component. In this section we briefly explore possible lines of work from both perspectives.

Meta-data has been proven useful in the detection of Android malware in Chapter 4 and the identification of potential repackaged applications in Chapter 7. In summary, the main lines for future work on meta-data could be summarized as follows:

- In Chapter 4 more complex models, such as neural networks, could be used provided an increase in the size of analysis datasets.
- In Chapter 7 cross-market repackaging could be investigated, together with an assessment of the quality of a Market according to the clones available in there (along with other relevant observations).
- In both chapters, other meta-data fields could be inspected to obtain alternative or complementary items, useful for the aims of each of the chapters. Furthermore, including more applications, either from Google Play or any other market is another possibility.

The AV engine detection analysis proposed in Chapter 5 and Chapter 6 could be extended with new analytic approaches, even supervised approaches provided ground truth data was obtained. More specifically, contributions could be extended as follows:

- Obtain ground truth for suspicious applications that would be later labeled by multi-scanner tools. For instance, manual inspection of applications is a possibility, although hardly scalable.
- Increase the collection of AVs by including different multi-scanner systems together with other expert systems in each application malware analysis.

Then, regarding optical WDM networks, the results from Chapter 8 encourage different possibilities to extend the proposed framework and include new intelligent network protocols. Concisely:

- The development of network protocol approximations as *intelligent heuristics* that emulate optimal solutions through ML and thus, are faster and yield better results than traditional heuristics.
- Future work should also complete the third stage of the proposed framework and contribute to the development of ML-powered SDN protocols that can leverage the flexibility and capabilities of the combination of both disciplines.

Finally, some of the lessons learned together with the methodological approaches and procedures conform with other related works the basis for a new paradigm in the AI/ML field: *AIaaS* or *MLaaS* (AI or ML as a Service respectively). Such paradigms should work towards the design, definition and development of AI tools, methodologies and components that can help inexperienced users from different backgrounds to systematically leverage the advantages of AI into their expertise areas.

9.3 List of Publications during Thesis Period

These dissertation results and conclusions are supported by a set of articles published in different conferences and journals during the thesis period. In what follows, academic publications supporting this thesis are listed together with non-academic contributions and other relevant merits achieved during the PhD:

- **Chapter 4**. Android Meta-data for Malware Detection.
  - C1 **Android malware detection from Google Play meta-data: selection of important features**; *A. Muñoz, I. Martín, A. Guzmán, J. A. Hernández* in IEEE Conf. Communications and Network Security (CNS'15). Florence, Italy. Sep 2015; doi: `https://doi.org/10.1109/CNS.2015.7346893`
  - J1 **Android Malware Characterization using Metadata and Machine Learning Techniques**; *I. Martín, A. Muñoz J. A. Hernández, A. Guzmán* in Security and Communication Networks, Hindawi, June 2018, vol 2018; doi: `https://doi.org/10.1155/2018/5749481`
- **Chapter 5**. Data-driven Interrelation Analysis of AV Engines.

C2 **Insights of Antivirus Relationships when Detecting Android Malware: A Data Analytics Approach**; *I. Martín, J. A. Hernández, S. Santos, A. Guzmán* in ACM Conf. Computer and Communications Security (CCS'16). Viena, Austria. Oct 2016, pages 1778-178; doi: https://doi.org/10.1145/2976749.2989038

C3 **Analysis and Evaluation of Antivirus Engines in Detecting Android Malware: A Data Analytics Approach**, *I. Martín, J. A. Hernández, S. Santos*, in European Intelligence and Security Informatics Conference (EISIC'18). Karlskrona Sweden, Oct 2018

- **Chapter 6**. AV Label-based Analysis of Malware Families.

C4 **SignatureMiner: A fast Anti-Virus signature intelligence tool**; *I. Martín, J. A. Hernández, S. Santos* in IEEE Conf. Communications and Network Security (CNS'18). Beijing, China. Apr 2018 **Best Poster Award**; doi: https://doi.org/10.1109/CNS.2018.8433141

J2 **Machine Learning based analysis and Classification of Android Malware Signatures**; *I. Martín, J. A. Hernández, S. Santos*, in Elsevier Future Generation Computer Systems, August 2019, vol 97, pages 295-305; doi: https://doi.org/10.1016/j.future.2019.03.006

- **Chapter 7**. Android Meta-data for Repackaging Detection.

J3 **CloneSpot: Fast detection of Android Repackages**; *I. Martín, J. A. Hernández*, in Elsevier Future Generation Computer Systems, May 2019, vol 94, pages 740-748, doi: https://doi.org/10.1016/j.future.2018.12.050

- **Chapter 8**. Optical WDM Networks Configuration from an ML perspective.

C5 **Is Machine Learning Suitable for Solving RWA Problems in Optical Networks?** *I. Martín, J. A. Hernández, S. Troia, F. Musumeci, G. Maier, O. González de Dios*, in European Conference on Optical Communications (ECOC'18). Rome, Italy, Sept 2018; doi: https://doi.org/10.1109/ECOC.2018.8535562

C6 **Netgen: A Fast and Scalable Tool for the Generation and Labeling of Networking Datasets**, *I. Martín, J. A. Hernández, O. González de Dios*, under review.

J4 **Machine-Learning-Based Routing and Wavelength Assignment in Software-Defined Optical Networks**, *I. Martín, S. Troia, J. A. Hernández, Alberto Rodríguez, F. Musumeci, G. Maier, Rodolfo Alvizu, O. González de Dios*, under review.

- **Other Publications**

J5 **Salary Prediction in the IT Job Market with Few High-Dimensional Samples: A Spanish Case Study**; *I. Martín, A. Mariello, R. Batitti, J. A. Hernández* in International Journal of Computational Intelligence Systems, Atlantic Press, June 2018, vol 11, Issue 1, pages 1192-1209, doi: https://doi.org/10.2991/ijcis.11.1.90

C7 **Machine-Learning-Assisted Routing in SDN-based Optical Networks** *S. Troia, A.Rodríguez, I. Martín, J. A. Hernández, O. González de Dios, R. Alvizu, F. Musumeci, G. Maier*, in European Conference on Optical Communications (ECOC'18). Rome, Italy, Sept 2018; doi: https://doi.org/10.1109/ECOC.2018.8535437

J6 **Meeting the traffic requirements of residential users in the next decade with current FTTH standards: how much? how long?**; *J. A. Hernández, R. Sánchez, I. Martín, D. Larrabeiti*, in IEEE Communications Magazine, 2018, early access, pages 2-7, doi: https://doi.org/10.1109/mcom.2018.1800173

C8 **Machine Learning-assisted Planning and Provisioning for SDN/NFV-enabled Metropolitan Networks**, *S. Troia, D. Eugui, I. Martín, L. M. Moreira, Guido Maier (1) , J. A. Hernández, O. González de Dios, M. Garrich, J. L. Romero-Gázquez, F.J. Moreno-Muro, P. Pavón, R. Casellas*, in European Conference on Networks and Communications (EUCNC'19), Valencia, Spain, June 2019.

J7 **Expanding the Measurement of Human Culture**, *N. Obradovich, O. Özak, I. Martín, I. Ortuño-Ortín, E. Awad, M. Cebrián, R. Cuevas, K. Desmet, I. Rahwan, and A. Cuevas*, under review.

- **Non-academic publications**

G1 **SignatureMiner** [Github repository]. Available at: https://github.com/ignmarti/SignatureMiner

G2 **Netgen** [Github repository]. Available at: https://github.com/ignmarti/netgen

P1 [PRESS] **Un estudio sitúa a Cataluña como la quinta comunidad menos semejante al resto** [Spanish]. Available at https://www.efe.com/efe/espana/sociedad/un-estudio-situa-a-cataluna-como-la-quinta-comunidad-menos-semejante-al-resto/10004-3469622

P2 [PRESS] **SignatureMiner, nuestra herramienta de análisis y la homogeneización de firmas de antivirus** [Spanish]. Available at: https://blog.elevenpaths.com/2018/06/signatureminer-herramienta-analisis-antivirus-ciberseguridad.html

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] A. Graves, A.-r. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, 2013, pp. 6645–6649.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, “Mastering the game of go with deep neural networks and tree search,” *nature*, vol. 529, no. 7587, p. 484, 2016.

[4] P. Vinod, R. Jaipur, V. Laxmi, and M. Gaur, “Survey on malware detection methods,” in *Proceedings of the 3rd Hackers’ Workshop on Computer and Internet Security (IITKHACK’09)*, 2009, pp. 74–79.

[5] E. Gandotra, D. Bansal, and S. Sofat, “Malware analysis and classification: A survey,” *Journal of Information Security*, vol. 5, no. 02, p. 56, 2014.

[6] R. Mohandas, “Hacking the malware–a reverse-engineer’s analysis,” 2007.

[7] A. Epishkina and S. Zapechnikov, “A syllabus on data mining and machine learning with applications to cybersecurity,” in *Digital Information Processing, Data Mining, and Wireless Communications (DIPDMWC), 2016 Third International Conference on*. IEEE, 2016, pp. 194–199.

[8] T. Mahmood and U. Afzal, “Security analytics: Big data analytics for cybersecurity: A review of trends, techniques and tools,” in *2013 2nd National Conference on Information Assurance (NCIA)*, Dec 2013, pp. 129–134.

[9] O. Lysne, *Static Detection of Malware*. Cham: Springer International Publishing, 2018, pp. 57–66. [Online]. Available: https://doi.org/10.1007/978-3-319-74950-1_7

[10] A. Moser, C. Kruegel, and E. Kirda, “Limits of static analysis for malware detection,” in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, Dec 2007, pp. 421–430.

[11] J. Devesa, I. Santos, X. Cantero, Y. K. Penya, and P. G. Bringas, “Automatic behaviour-based analysis and classification system for malware detection.” in *ICEIS (2)*, 2010, pp. 395–399.

[12] C. Willems, T. Holz, and F. Freiling, “Toward automated dynamic malware analysis using cwsandbox,” *IEEE Security Privacy*, vol. 5, no. 2, pp. 32–39, March 2007.

[13] M. Vasilescu, L. Gheorghe, and N. Tapus, "Practical malware analysis based on sandboxing," in *2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*, Sept 2014, pp. 1–6.

[14] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, "The cuckoo sandbox," 2012.

[15] "Meet virustotal droidy, our new android sandbox," http://blog.virustotal.com/2018/04/meet-virustotal-droidy-our-new-android.html, accessed: April 2019.

[16] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of android malware," in *Proceedings of the Seventh European Workshop on System Security*, ser. EuroSec '14. New York, NY, USA: ACM, 2014, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2592791.2592796

[17] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1–12, 2017.

[18] B. Baskaran and A. Ralescu, "A study of android malware detection techniques and machine learning," 2016.

[19] D. Gavrilut, M. Cimpoesu, D. Anton, and L. Ciortuz, "Malware detection using machine learning," in *Computer Science and Information Technology, 2009. IMCSIT '09. International Multiconference on*, Oct 2009, pp. 735–741.

[20] "Vx heaven," http://83.133.184.251/virensimulation.org/, accessed: April 2019.

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

[22] S. Y. Yerima, S. Sezer, and G. McWilliams, "Analysis of bayesian classification-based approaches for android malware detection," *IET Information Security*, vol. 8, no. 1, pp. 25–36, 2014.

[23] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for android malware detection," *Computers & Security*, vol. 49, pp. 255 – 273, 2015.

[24] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "Androdialysis: Analysis of android intent effectiveness in malware detection," *Computers & Security*, vol. 65, pp. 121 – 134, 2017.

[25] S. Huda, J. Abawajy, M. Alazab, M. Abdollalihian, R. Islam, and J. Yearwood, "Hybrids of support vector machine wrapper and filter based framework for malware detection," *Future Generation Computer Systems*, vol. 55, pp. 376 – 390, 2016.

[26] N. Peiravian and X. Zhu, "Machine learning for android malware detection using permission and api calls," in *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, Nov 2013, pp. 300–305.

[27] S. Y. Yerima, S. Sezer, and I. Muttik, "High accuracy android malware detection using ensemble learning," *IET Information Security*, vol. 9, no. 6, pp. 313–320, 2015.

[28] H. Fereidooni, V. Moonsamy, M. Conti, and L. Batina, "Efficient classification of android malware in the wild using robust static features," *Protecting Mobile Networks and Devices: Challenges and Solutions*, vol. 1, pp. 181–209, 2016.

[29] H.-S. Ham and M.-J. Choi, "Analysis of android malware detection performance using machine learning classifiers," in *ICT Convergence (ICTC), 2013 International Conference on*, Oct 2013, pp. 490–495.

[30] M. Mas'ud, S. Sahib, M. Abdollah, S. Selamat, and R. Yusof, "Analysis of features selection and machine learning classifier in android malware detection," in *Information Science and Applications (ICISA), 2014 International Conference on*, May 2014, pp. 1–5.

[31] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis: Android malware under the magnifying glass," *Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001*, 2014.

[32] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer, "Andrubis – 1,000,000 apps later: A view on current android malware behaviors," in *2014 3 Int. Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, Sept 2014, pp. 3–17.

[33] Z. Wang, C. Li, Z. Yuan, Y. Guan, and Y. Xue, "Droidchain: A novel android malware detection method based on behavior chains," *Pervasive and Mobile Computing*, vol. 32, pp. 3 – 14, 2016, mobile Security, Privacy and Forensics.

[34] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," *Proc. of Symp. Network and Distributed System Security*, 2014.

[35] S. N. Hanumanthegowda, "Automated machine learning-based detection of malicious Android applications using Google Play Metadata," Master's thesis, Northeastern University, Illinois, USA, 2013.

[36] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. Bringas, and G. Álvarez, "PUMA: Permission usage to detect malware in Android," in *Proc. Int. Conference CISIS'12-ICEUTE'12-SOCO'12*, ser. Advances in Intelligent Systems and Computing, 2013, vol. 189, pp. 289–298.

[37] A. Aswini and P. Vinod, "Droid permission miner: Mining prominent permissions for android malware analysis," in *Applications of Digital Information and Web Technologies (ICADIWT), 2014 Fifth International Conference on the*, Feb 2014, pp. 81–86.

[38] C. L. P. M. Hein, "Permission based malware protection model for android application," in *Proceedings of International Conference on Advances in Engineering and Technology (ICAET'2014)(March 2014). doi*, vol. 10, 2014.

[39] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 73–84.

[40] V. Moonsamy, J. Rong, and S. Liu, "Mining permission patterns for contrasting clean and malicious android applications," *Future Generation Computer Systems*, vol. 36, pp. 122–132, 2014.

[41] Z. Aung and W. Zaw, "Permission-based android malware detection," *Int. J. Scientific and Technology Research*, vol. 2, no. 3, pp. 228–234, 2013.

[42] A. Egners, U. Meyer, and B. Marschollek, "Messing with android's permission model," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*. IEEE, 2012, pp. 505–514.

[43] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 627–638.

[44] S. Rastogi, K. Bhushan, and B. Gupta, "Android applications repackaging detection techniques for smartphone devices," *Procedia Computer Science*, vol. 78, pp. 26–32, 2016.

[45] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[46] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of semantically similar android applications," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 182–199.

[47] Q. Guan, H. Huang, W. Luo, and S. Zhu, "Semantics-based repackaging detection for mobile apps," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016, pp. 89–105.

[48] H. Gonzalez, A. A. Kadir, N. Stakhanova, A. J. Alzahrani, and A. A. Ghorbani, "Exploring reverse engineering symptoms in android apps," in *Proceedings of the Eighth European Workshop on System Security*. ACM, 2015, p. 7.

[49] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han, "Migdroid: Detecting app-repackaging android malware via method invocation graph," in *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*. IEEE, 2014, pp. 1–7.

[50] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 175–186.

[51] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *USENIX Security Symposium*, 2015, pp. 659–674.

[52] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang, "Detecting clones in android applications through analyzing user interfaces," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 163–173.

[53] S. Yue, W. Feng, J. Ma, Y. Jiang, X. Tao, C. Xu, and J. Lu, "Repdroid: an automated tool for android application repackaging detection," in *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 2017, pp. 132–142.

[54] P. Teufl, M. Ferk, A. Fitzek, D. Hein, S. Kraxberger, and C. Orthacker, "Malware detection by applying knowledge discovery processes to application metadata on the android market (google play)," *Security and Communication Networks*, vol. 9, no. 5, pp. 389–419, 2016.

[55] I. Gurulian, K. Markantonakis, L. Cavallaro, and K. Mayes, "You can't touch this: Consumer-centric android application repackaging detection," *Future Generation Computer Systems*, vol. 65, pp. 1–9, 2016.

[56] S. M. Kywe, Y. Li, R. H. Deng, and J. Hong, "Detecting camouflaged applications on mobile application markets," in *International Conference on Information Security and Cryptology*. Springer, 2014, pp. 241–254.

[57] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis, "Andradar: fast discovery of android applications in alternative markets," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2014, pp. 51–71.

[58] A. Kohli, "Decisiondroid: a supervised learning-based system to identify cloned android applications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 1059–1061.

[59] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 2016, pp. 468–471.

[60] C. Ren, K. Chen, and P. Liu, "Droidmarking: resilient software watermarking for impeding android application repackaging," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 635–646.

[61] W. Zhou, X. Zhang, and X. Jiang, "Appink: watermarking android apps for repackaging deterrence," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 1–12.

[62] F. Kanei, Y. Takata, M. Akiyama, T. Yagi, and T. Yada, "Poster: Protecting android apps from repackaging by self-protection code," 2017.

[63] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li, "Resilient decentralized android application repackaging detection using logic bombs," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, 2018, pp. 50–61. [Online]. Available: http://doi.acm.org/10.1145/3168820

[64] J. Song, M. Zhang, C. Han, K. Wang, and H. Zhang, "Towards fast repackaging and dynamic authority management on android," *Wuhan University Journal of Natural Sciences*, vol. 21, no. 1, pp. 1–9, 2016.

[65] P. Berthome, T. Fecherolle, N. Guilloteau, and J.-F. Lalande, "Repackaging android applications for auditing access to private data," in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*. IEEE, 2012, pp. 388–396.

[66] J.-H. Jung, J. Y. Kim, H.-C. Lee, and J. H. Yi, "Repackaging attack on android banking applications and its countermeasures," *Wireless Personal Communications*, vol. 73, no. 4, pp. 1421–1437, 2013.

[67] S.-W. Park and J. H. Yi, "Multiple device login attacks and countermeasures of mobile voip apps on android." *J. Internet Serv. Inf. Secur.*, vol. 4, no. 4, pp. 115–126, 2014.

[68] T. Cho, G. Na, D. Lee, and J. H. Yi, "Account forgery and privilege escalation attacks on android home cloud devices," *Advanced Science Letters*, vol. 21, no. 3, pp. 381–386, 2015.

[69] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, Jan 2014.

[70] J. Haffejee and B. Irwin, "Testing antivirus engines to determine their effectiveness as a security layer," in *Information Security for South Africa (ISSA), 2014*. IEEE, 2014, pp. 1–6.

[71] D. Maier, T. Müller, and M. Protsenko, "Divide-and-conquer: Why android malware cannot be stopped," in *2014 Ninth International Conference on Availability, Reliability and Security*, Sept 2014, pp. 30–39.

[72] H. Huang, K. Chen, P. Liu, S. Zhu, and D. Wu, "Uncovering the dilemmas on antivirus software design in modern mobile platforms," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2014, pp. 359–366.

[73] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders, "Secloud: A cloud-based comprehensive and lightweight security solution for smartphones," *Computers & Security*, vol. 37, pp. 215 – 227, 2013.

[74] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""andromaly": a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2012.

[75] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Security and Privacy in Communication Networks*. Springer, 2013, pp. 86–103.

[76] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day android malware detection," in *Proc. of the 10th Int. Conf. on Mobile Systems, Applications, and Services*, ser. MobiSys '12, 2012.

[77] G. Vasiliadis and S. Ioannidis, "Gravity: a massively parallel antivirus engine," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2010, pp. 79–96.

[78] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *Software Engineering (WCSE), 2012 Third World Congress on*. IEEE, 2012, pp. 101–104.

[79] K. Shaerpour, A. Dehghantanha, and R. Mahmod, "Trends in android malware detection," *Journal of Digital Forensics, Security and Law*, vol. 8, no. 3, pp. 21–40, 2013.

[80] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046619

[81] M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, July 2013, pp. 163–171.

[82] O. E. David and N. S. Netanyahu, "Deepsign: Deep learning for automatic malware signature generation and classification," in *Neural Networks (IJCNN), 2015 International Joint Conference on*. IEEE, 2015, pp. 1–8.

[83] H. Huang, K. Chen, C. Ren, P. Liu, S. Zhu, and D. Wu, "Towards discovering and understanding unexpected hazards in tailoring antivirus software for android," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: ACM, 2015, pp. 7–18.

[84] M. I. Al-Saleh, A. M. Espinoza, and J. R. Crandall, "Antivirus performance characterisation: system-wide view," *IET Information Security*, vol. 7, no. 2, pp. 126–133, 2013.

[85] D. Uluski, M. Moffie, and D. Kaeli, "Characterizing antivirus workload execution," *SIGARCH Comput. Archit. News*, vol. 33, no. 1, pp. 90–98, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1055626.1055639

[86] W. Yan and E. Wu, *Complex Sciences: First International Conference, Complex 2009, Shanghai, China, February 23-25, 2009. Revised Papers, Part 1.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ch. Toward Automatic Discovery of Malware Signature for Anti-Virus Cloud Computing, pp. 724–728. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02466-5_70

[87] C. A. Martínez, G. I. Echeverri, and A. G. C. Sanz, "Malware detection based on cloud computing integrating intrusion ontology representation," in *Communications (LATINCOM), 2010 IEEE Latin-American Conference on*, Sept 2010, pp. 1–6.

[88] J. Oberheide, E. Cooke, and F. Jahanian, "Cloudav: N-version antivirus in the network cloud," in *USENIX Security Symposium*, 2008, pp. 91–106.

[89] M. Schweiger and S. C. B. Endicott-Popovsky, "Malware analysis on the cloud: Increased performance, reliability, and flexibilty," in *Proceedings of the International Conference on Cloud Security Management: ICCSM 2013*. Academic Conferences Limited, 2013, p. 127.

[90] "Virus total," https://www.virustotal.com, accessed: April 2019.

[91] "Metascan online," https://metadefender.opswat.com, accessed: April 2019.

[92] "Jotti malware scanner," https://virusscan.jotti.org, accessed: April 2019.

[93] M. Cukier, I. Gashi, B. Sobesto, and V. Stankovic, "Does malware detection improve with diverse antivirus products? an empirical study," in *32nd International Conference on Computer Safety, Reliability and Security. IEEE*, 2013.

[94] P. Bishop, R. Bloomfield, I. Gashi, and V. Stankovic, "Diverse protection systems for improving security: a study with antivirus engines," 2012.

[95] ——, "Diversity for security: A study with off-the-shelf antivirus engines," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, Nov 2011, pp. 11–19.

[96] I. Gashi, B. Sobesto, S. Mason, V. Stankovic, and M. Cukier, "A study of the relationship between antivirus regressions and label changes," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2013, pp. 441–450.

[97] B. Miller, A. Kantchelian, M. C. Tschantz, S. Afroz, R. Bachwani, R. Faizullabhoy, L. Huang, V. Shankar, T. Wu, G. Yiu *et al.*, "Back to the future: Malware detection with temporally consistent labels," 2015.

[98] A. Mohaisen and O. Alrawi, *AV-Meter: An Evaluation of Antivirus Scans and Labels*. Cham: Springer International Publishing, 2014, pp. 112–131. [Online]. Available: https://doi.org/10.1007/978-3-319-08509-8_7

[99] D. Quarta, F. Salvioni, A. Continella, and S. Zanero, "Toward systematically exploring antivirus engines," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Cham: Springer International Publishing, 2018, pp. 393–403.

[100] H. Huang, C. Zheng, J. Zeng, W. Zhou, S. Zhu, P. Liu, S. Chari, and C. Zhang, "Android malware development on public malware scanning platforms: A large-scale data-driven study," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 1090–1099.

[101] E. Willems, "The good and the bad about av multi scanner services," 2017.

[102] M. Hurier, K. Allix, T. Bissyandé, J. Klein, and Y. L. Traon, "On the lack of consensus in anti-virus decisions: metrics and insights on building ground truths of android malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 142–162.

[103] F. Maggi, A. Bellini, G. Salvaneschi, and S. Zanero, "Finding non-trivial malware naming inconsistencies," in *International Conference on Information Systems Security*. Springer, 2011, pp. 144–159.

[104] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 4, Part 1, pp. 1104 – 1117, 2014.

[105] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. of Symp. Security and Privacy*, May 2012, pp. 95–109.

[106] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, pp. 252–276.

[107] M. Alazab, V. Moonsamy, L. Batten, P. Lantz, and R. Tian, "Analysis of malicious and benign android applications," in *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*. IEEE, 2012, pp. 608–616.

[108] J. Canto, M. Dacier, E. Kirda, and C. Leita, "Large scale malware collection: lessons learned," in *IEEE SRDS Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*. Citeseer, 2008.

[109] A. Kantchelian, M. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A. Joseph, and J. Tygar, "Better malware ground truth: Techniques for weighting anti-virus vendor labels," in *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '15. New York, NY, USA: ACM, 2015, pp. 45–56. [Online]. Available: http://doi.acm.org/10.1145/2808769.2808780

[110] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 230–253.

[111] M. Hurier, G. Suarez-Tangil, S. Dash, T. Bissyandé, Y. Traon, J. Klein, and L. Cavallaro, "Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17, 2017, pp. 425–435.

[112] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett *et al.*, "Knowledge-defined networking," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, pp. 2–10, 2017.

[113] S. Ayoubi, N. Limam, M. A. Salahuddin, N. Shahriar, R. Boutaba, F. Estrada-Solano, and O. M. Caicedo, "Machine learning for cognitive network management," *IEEE Communications Magazine*, vol. 56, no. 1, pp. 158–165, 2018.

[114] Z. M. Fadlullah, F. Tang, B. Mao, N. Kato, O. Akashi, T. Inoue, and K. Mizutani, "State-of-the-art deep learning: Evolving machine intelligence toward tomorrow's intelligent network traffic control systems," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2432–2455, 2017.

[115] J. Mata, I. de Miguel, R. J. Durán, N. Merayo, S. K. Singh, A. Jukan, and M. Chamania, "Artificial intelligence (ai) methods in optical networks: A comprehensive survey," *Optical Switching and Networking*, 2018.

[116] F. Musumeci, C. Rottondi, A. Nag, I. Macaluso, D. Zibar, M. Ruffini, and M. Tornatore, "A survey on application of machine learning techniques in optical networks," *arXiv preprint arXiv:1803.07976*, pp. 1–21, 2018.

[117] N. Kato, Z. M. Fadlullah, B. Mao, F. Tang, O. Akashi, T. Inoue, and K. Mizutani, "The deep learning vision for heterogeneous network traffic control: Proposal, challenges, and future perspective," *IEEE wireless communications*, vol. 24, no. 3, pp. 146–153, 2017.

[118] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, 2018.

[119] L. Cui, F. R. Yu, and Q. Yan, "When big data meets software-defined networking: Sdn for big data and big data for sdn," *IEEE network*, vol. 30, no. 1, pp. 58–65, 2016.

[120] P. Wang, S.-C. Lin, and M. Luo, "A framework for qos-aware traffic classification using semi-supervised machine learning in sdns," in *Services Computing (SCC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 760–765.

[121] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[122] T. Glennan, C. Leckie, and S. M. Erfani, "Improved classification of known and unknown network traffic flows using semi-supervised machine learning," in *Australasian Conference on Information Security and Privacy*. Springer, 2016, pp. 493–501.

[123] J. Zhang, Y. Xiang, Y. Wang, W. Zhou, Y. Xiang, and Y. Guan, "Network traffic classification using correlation information," *IEEE Transactions on Parallel and Distributed systems*, vol. 24, no. 1, pp. 104–117, 2013.

[124] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu, "Robust network traffic classification," *IEEE/ACM Transactions on Networking (TON)*, vol. 23, no. 4, pp. 1257–1270, 2015.

[125] M. Shafiq, X. Yu, A. A. Laghari, L. Yao, N. K. Karn, and F. Abdessamia, "Network traffic classification techniques and comparative analysis using machine learning algorithms," in *Computer and Communications (ICCC), 2016 2nd IEEE International Conference on*. IEEE, 2016, pp. 2451–2455.

[126] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[127] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Y. Geng, L. Chen, K. Chen, and H. Jin, "Online flow size prediction for improved network routing," in *Network Protocols (ICNP), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–6.

[128] Z. Chen, J. Wen, Y. Geng *et al.*, "Predicting future traffic using hidden markov models," in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. IEEE, 2016, pp. 1–6.

[129] N. Nikaein, M. Laner, K. Zhou, P. Svoboda, D. Drajic, M. Popovic, and S. Krco, "Simple traffic modeling framework for machine type communication," in *Wireless Communication Systems (ISWCS 2013), Proceedings of the Tenth International Symposium on*. VDE, 2013, pp. 1–5.

[130] B. Mao, Z. M. Fadlullah, F. Tang, N. Kato, O. Akashi, T. Inoue, and K. Mizutani, "Routing or computing? the paradigm shift towards intelligent computer network packet transmission based on deep learning," *IEEE Transactions on Computers*, vol. 66, no. 11, pp. 1946–1960, 2017.

[131] F. Tang, B. Mao, Z. M. Fadlullah, N. Kato, O. Akashi, T. Inoue, and K. Mizutani, "On removing routing protocol from future wireless networks: A real-time deep learning approach for intelligent traffic control," *IEEE Wireless Communications*, 2017.

[132] J. Shaikh, M. Fiedler, and D. Collange, "Quality of experience from user and network perspectives," *annals of telecommunications-annales des telecommunications*, vol. 65, no. 1-2, pp. 47–57, 2010.

[133] M. S. Mushtaq, B. Augustin, and A. Mellouk, “Empirical study based on machine learning approach to assess the qos/qoe correlation,” in *2012 17th European Conference on Networks and Optical Communications*, June 2012, pp. 1–7.

[134] L. Amour, M. I. Boulabiar, S. Souihi, and A. Mellouk, “An improved qoe estimation method based on qos and affective computing,” in *2018 International Symposium on Programming and Systems (ISPS)*, April 2018, pp. 1–6.

[135] V. Menkovski, G. Exarchakos, and A. Liotta, “Machine learning approach for quality of experience aware networks,” in *2010 International Conference on Intelligent Networking and Collaborative Systems*, Nov 2010, pp. 461–466.

[136] X. Luo, J. Liu, D. Zhang, and X. Chang, “A large-scale web qos prediction scheme for the industrial internet of things based on a kernel machine learning algorithm,” *Computer Networks*, vol. 101, pp. 81 – 89, 2016, industrial Technologies and Applications for the Internet of Things.

[137] S. Aroussi and A. Mellouk, “Survey on machine learning-based qoe-qos correlation models,” in *2014 International Conference on Computing, Management and Telecommunications (ComManTel)*, April 2014, pp. 200–204.

[138] T.-K. Hui and C.-K. Tham, “Adaptive provisioning of differentiated services networks based on reinforcement learning,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 33, no. 4, pp. 492–501, 2003.

[139] A. Yu, H. Yang, W. Bai, L. He, H. Xiao, and J. Zhang, “Leveraging deep learning to achieve efficient resource allocation with traffic evaluation in datacenter optical networks,” in *2018 Optical Fiber Communications Conference and Exposition (OFC)*. IEEE, 2018, pp. 1–3.

[140] M. Hayashi, “Machine learning-assisted management of a virtualized network,” in *2018 Optical Fiber Communications Conference and Exposition (OFC)*. IEEE, 2018, pp. 1–3.

[141] W. Mo, C. L. Gutterman, Y. Li, S. Zhu, G. Zussman, and D. C. Kilper, “Deep-neural-network-based wavelength selection and switching in roadm systems,” *Journal of Optical Communications and Networking*, vol. 10, no. 10, pp. D1–D11, 2018.

[142] C. L. Gutterman, W. Mo, S. Zhu, Y. Li, D. C. Kilper, and G. Zussman, “Neural network based wavelength assignment in optical switching,” in *Proceedings of the Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*. ACM, 2017, pp. 37–42.

[143] D. Barman and I. Matta, “Model-based loss inference by tcp over heterogeneous networks,” in *Proceedings of WiOpt*, vol. 4, 2004.

[144] I. El Khayat, P. Geurts, and G. Leduc, “Enhancement of tcp over wired/wireless networks with packet loss classifiers inferred by supervised learning,” *Wireless Networks*, vol. 16, no. 2, pp. 273–290, 2010.

[145] A. Jayaraj, T. Venkatesh, and C. S. R. Murthy, "Loss classification in optical burst switching networks using machine learning techniques: improving the performance of tcp," *IEEE Journal on Selected Areas in Communications*, vol. 26, no. 6, pp. 45–54, 2008.

[146] B. Hariri and N. Sadati, "Nn-red: an aqm mechanism based on neural networks," *Electronics Letters*, vol. 43, no. 19, pp. 1053–1055, 2007.

[147] D. H. Hagos, P. E. Engelstad, A. Yazidi, and O. Kure, "A machine learning approach to tcp state monitoring from passive measurements," in *2018 Wireless Days (WD)*, April 2018, pp. 164–171.

[148] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A machine learning approach to tcp throughput prediction," *IEEE/ACM Transactions on Networking (TON)*, vol. 18, no. 4, pp. 1026–1039, 2010.

[149] Y. Edalat, J.-S. Ahn, and K. Obraczka, "Smart experts for network state estimation," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 622–635, 2016.

[150] J. Barron, M. Crotty, E. Elahi, R. Riggio, D. R. Lopez, and M. P. de Leon, "Towards self-adaptive network management for a recursive network architecture," in *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*. IEEE, 2016, pp. 1143–1148.

[151] M. Zorzi, A. Zanella, A. Testolin, M. D. F. De Grazia, and M. Zorzi, "Cognition-based networks: A new perspective on network optimization using learning and distributed intelligence," *IEEE Access*, vol. 3, pp. 1512–1530, 2015.

[152] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, "Machine learning for networking: Workflow, advances and opportunities," *IEEE Network*, vol. 32, no. 2, pp. 92–99, 2018.

[153] K. Zheng, Z. Yang, K. Zhang, P. Chatzimisios, K. Yang, and W. Xiang, "Big data-driven optimization for mobile networks toward 5g," *IEEE network*, vol. 30, no. 1, pp. 44–51, 2016.

[154] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "Research challenges for traffic engineering in software defined networks," *IEEE Network*, vol. 30, no. 3, pp. 52–58, 2016.

[155] "iPerf- the ultimate speed test tool for tcp, udp and sctp," https://iperf.fr/, accessed: 2019-01-02.

[156] A. Botta, A. Dainotti, and A. Pescapé, "A tool for the generation of realistic network workload for emerging networking scenarios," *Computer Networks*, vol. 56, no. 15, pp. 3531–3547, 2012.

[157] P. Pavon-Marino and J.-L. Izquierdo-Zaragoza, "Net2plan: an open source network planning tool for bridging the gap between academia and industry," *IEEE Network*, vol. 29, no. 5, pp. 90–96, 2015.

[158] S. Uhlig, B. Quoitin, J. Lepropre, and S. Balon, "Providing public intradomain traffic matrices to the research community," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 1, pp. 83–86, 2006.

[159] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly, "SNDlib 1.0–Survivable Network Design Library," in *Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium*, April 2007, http://sndlib.zib.de, extended version accepted in Networks, 2009.

[160] C. Rottondi, L. Barletta, A. Giusti, and M. Tornatore, "Machine-learning method for quality of transmission prediction of unestablished lightpaths," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 10, no. 2, pp. A286–A297, 2018.

[161] S. Shahkarami, F. Musumeci, F. Cugini, and M. Tornatore, "Machine-learning-based soft-failure detection and identification in optical networks," in *2018 Optical Fiber Communications Conference and Exposition (OFC)*. IEEE, 2018, pp. 1–3.

[162] E. Seve, J. Pesic, C. Delezoide, S. Bigo, and Y. Pointurier, "Learning process for reducing uncertainties on network parameters and design margins," *Journal of Optical Communications and Networking*, vol. 10, no. 2, pp. A298–A306, 2018.

[163] F. Morales, M. Ruiz, L. Gifre, L. M. Contreras, V. López, and L. Velasco, "Virtual network topology adaptability based on data analytics for traffic prediction," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 9, no. 1, pp. A35–A45, 2017.

[164] T. A. Eriksson, H. Bülow, and A. Leven, "Applying neural networks in optical communication systems: possible pitfalls," *IEEE Photonics Technology Letters*, vol. 29, no. 23, pp. 2091–2094, 2017.

[165] H. Zang, J. P. Jue, B. Mukherjee *et al.*, "A review of routing and wavelength assignment approaches for wavelength-routed optical wdm networks," *Optical networks magazine*, vol. 1, no. 1, pp. 47–60, 2000.

[166] F. Martinelli, N. Andriolli, P. Castoldi, and I. Cerutti, "Genetic approach for optimizing the placement of all-optical regenerators in wson," *IEEE/OSA Journal of Optical Communications and Networking s*, vol. 6, no. 11, pp. 1028–1037, 2014.

[167] Y. Pointurier and F. Heidari, "Reinforcement learning based routing in all-optical networks," in *2007 Fourth International Conference on Broadband Communications, Networks and Systems (BROADNETS'07)*. IEEE, 2007, pp. 919–921.

[168] X. Chen, J. Guo, Z. Zhu, R. Proietti, A. Castro, and S. Yoo, "Deep-rmsa: A deep-reinforcement-learning routing, modulation and spectrum assignment agent for elastic optical networks," in *2018 Optical Fiber Communications Conference and Exposition (OFC)*. IEEE, 2018, pp. 1–3.

[169] F. S. Abkenar and A. G. Rahbar, "Study and analysis of routing and spectrum allocation (rsa) and routing, modulation and spectrum allocation (rmsa) algorithms in elastic optical networks (eons)," *Optical Switching and Networking*, vol. 23, pp. 5 – 39, 2017.

[170] C. Pennachin and B. Goertzel, "Contemporary approaches to artificial general intelligence," in *Artificial general intelligence*. Springer, 2007, pp. 1–30.

[171] D. J. Hand, "Principles of data mining," *Drug safety*, vol. 30, no. 7, pp. 621–622, 2007.

[172] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. New York, NY, USA: ACM, 2009, pp. 1113–1120. [Online]. Available: http://doi.acm.org/10.1145/1553374.1553516

[173] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[174] J. Leskovec *et al.*, *Mining of massive datasets*. Cambridge university press, 2014.

[175] G. James, D. Witten, T. Hastie, and R. Tibshiran, *An introduction to statistical learning with applications in R*. Springer Texts in Statistics, 2015.

[176] D. R. Cox, "The regression analysis of binary sequences," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 215–242, 1958.

[177] G. E. Box and G. C. Tiao, *Bayesian inference in statistical analysis*. John Wiley & Sons, 2011, vol. 40.

[178] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[179] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep 1995. [Online]. Available: https://doi.org/10.1007/BF00994018

[180] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 3, pp. 660–674, May 1991.

[181] T. K. Ho, "Random decision forests," in *Document analysis and recognition, 1995., proceedings of the third international conference on*, vol. 1. IEEE, 1995, pp. 278–282.

[182] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[183] R. C. Dubes and A. K. Jain., *Algorithms for Clustering Data*. Prentice Hall, 1988.

[184] I. Jolliffe, *Principal component analysis*. Springer, 2011.

[185] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[186] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11 – 26, 2017.

[187] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[188] K. Sato, C. Young, and D. Patterson, "An in-depth look at google's first tensor processing unit (tpu)," *Google Cloud Big Data and Machine Learning Blog*, vol. 12, 2017.

[189] B. Thompson, *Exploratory and confirmatory factor analysis: Understanding concepts and applications*. American Psychological Association, 2004.

[190] J. J. Hox and T. M. Bechger, "An introduction to structural equation modeling," 1998.

[191] M. Girvan and M. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[192] M. S. Granovetter, "The strength of weak ties," in *Social Networks*, S. Leinhardt, Ed. Academic Press, 1977, pp. 347 – 367.

[193] F. Chollet, *Deep learning with python*. Manning Publications Co., 2017.

[194] T. Raykov and G. A. Marcoulides, *A first course in structural equation modeling*. Routledge, 2012.

[195] D. B. West *et al.*, *Introduction to graph theory*. Prentice hall Upper Saddle River, 2001, vol. 2.

[196] "Mobile os market share in 2018," https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, accessed: April 2019.

[197] "White paper: Kaspersky security network," Tech. Rep. [Online]. Available: https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2017/06/20080315/KESB_Whitepaper_KSN_ENG_final.pdf

[198] "Google play web site," https://play.google.com, accessed: April 2019.

[199] "Tacyt site at elevenpaths," https://www.elevenpaths.com/es/tecnologia/tacyt/index.html, accessed: April 2019.

[200] W. Tesfay, T. Booth, and K. Andersson, "Reputation based security model for android applications," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, June 2012, pp. 896–901.

[201] M. Hahsler, C. Buchta, B. Gruen, and K. Hornik, *arules: Mining Association Rules and Frequent Itemsets*, 2016, r package version 1.4-1. [Online]. Available: http://CRAN.R-project.org/package=arules

[202] Y. Rosseel, "lavaan: An R package for structural equation modeling," *Journal of Statistical Software*, vol. 48, no. 2, pp. 1–36, 2012. [Online]. Available: http://www.jstatsoft.org/v48/i02/

[203] "Signatureminer: An av intelligence tool," https://github.com/ignmarti/SignatureMiner, accessed: April 2019.

[204] T. Vidas and N. Christin, "Sweetening android lemon markets: measuring and combating malware in application marketplaces," in *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 2013, pp. 197–208.

[205] "A look into repackaged apps and its role in the mobile threat landscape," https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-into-repackaged-apps-and-its-role-in-the-mobile-threat-landscape/, accessed: April 2019.

[206] "Clonespot proof of concept," http://163.117.192.31:8080/CloneSpot/commands/info, accesed: April 2019.

[207] "Netgen: A network data generation tool," https://github.com/ignmarti/Netgen, accessed: April 2019.

[208] H. Zang, J. P. Jue, and B. Mukherjee, "A review of routing and wavelength assignment approaches for wavelength-routed optical WDM networks," *Optical Networks Magazine*, vol. 1, pp. 47–60, 2000.

[209] M. D. Vaughn and R. Wagner, "Metropolitan network traffic demand study," in *Lasers and Electro-Optics Society 2000 Annual Meeting. LEOS 2000. 13th Annual Meeting. IEEE*, vol. 1. IEEE, 2000, pp. 102–103.

[210] I. I. CPLEX, "V12. 1: User's manual for cplex," *International Business Machines Corporation*, vol. 46, no. 53, p. 157, 2009.