ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

Department of Computer Science and Engineering
Second Cycle Degree in Computer Science and Engineering

Clustering Analysis of Windows Malware using Static Features and Concept Drift Detection

Supervisor
Dr. Melis Andrea

Co-Supervisors
Prof. Aonzo Simone
Dr. Dambra Savino
Dr. Han Yufei

Candidate
Fabri Luca

Graduation Session IV
Academic Year 2023-2024

Abstract

Machine learning for malware classification shows promising results in terms of performance, but models are prone to degradation due to malware evolution. New malware families emerge every year, and attackers continually adapt techniques to evade detection by these systems. Given its critical application, it is essential to keep these models up to date and maintain their performance, despite the fast evolution and broad nature of malware. This challenge can be linked in the machine learning literature to concept drift, a phenomenon that leads to the degradation of classifiers' performances over time.

Parallely, while classification is widely used for malware detection, clustering methods provide a complementary approach by uncovering hidden structures in the data and identifying emerging malware families. More importantly, clustering can serve as a tool to track changes in malware feature distributions over time, offering a way to detect and analyze data distribution shifts.

This thesis explores different ML techniques for data mining malware samples based on static features. First, after an initial phase of dataset creation, clustering approaches are assessed for identifying groups of Windows malware. The features that characterize each cluster are computed leveraging hierarchical clustering algorithms and XAI techniques. Empirical experiments are conducted to study the relationship between these clusters and current family labeling systems and packing algorithms.

Additionally, concept drift detection is applied with respect to malware family labels leveraging a state-of-the-art technique, proposing some modifications to the existing project. This analysis enables the study of temporal changes in family assignments and highlights how malware families evolve, reveals potential inconsistencies in existing labeling systems, and provides a deeper understanding of the dynamics of malware ecosystems over time.

To my family.

Acknowledgements

First of all, I would like to thank Prof. Simone (Eurecom), Dr. Yufei (Inria), and Dr. Savino (NortonLifeLock) for the great opportunity they gave me to carry out an internship at Eurecom and to deeply study these research topics. It has been a wonderful experience filled with lots of challenges and very gratifying. Their advice has been essential in helping me conduct this thesis, and I am deeply grateful.

Secondly, I would like to thank my supervisor, Dr. Andrea Melis for agreeing to supervise this work.

I would like to thank all the people I met during my internship abroad in France, in particular: Readh, Medhi, and Andrea. I had a very good time with them, enjoying and exploring Côte d'Azur. I hope to see you soon. Also thanks to all the other people who made this internship possible: both UniBo, UniCA, and Eurecom offices.

This thesis is just the last piece of the puzzle of these university years. They have been years filled with both joy and sorrow, and I will never forget them. Most importantly, I would do it all over again.

A big thanks to my family, who supported me every day on my journey to achieving this goal, and to Elisa, who helped me through difficult times and stood by me in every choice I made.

Finally, thanks to all my university friends: Giovanni, Mattia, Alberto, Luca, Davide, Simone, Riccardo (and many others), and all my lifelong friends from my hometown, Senigallia.

Contents

Abstract **iii**

1 Introduction **1**

2 Motivation, Background, and Related Work **3**

2.1 Malware Analysis 3

2.1.1 Static analysis 3

2.1.2 Dynamic analysis 8

2.2 Machine Learning for Malware Analysis 9

2.2.1 Malware detection and family classification 9

2.2.2 Clustering malware 13

2.2.3 Issues of ML-based malware detection 14

2.3 Concept drift 15

2.3.1 Concept drift in malware domain 15

2.3.2 Mathematical formulation 16

2.3.3 Concept drift types 16

2.3.4 Concept drift detection 18

2.4 Hypothesis testing 22

2.4.1 p-value 23

2.4.2 Statistical hypothesis tests 25

2.5 Conformal Evaluation 32

2.5.1 Conformal Prediction theory 32

2.5.2 Non-Conformity Measure 32

2.5.3 Statistical decision assessment 34

2.5.4 Transcend Framework 34

2.5.5 Practical Conformal Evaluation 35

2.6 Related Work - Workplan, Tasks and Milestones 38

2.6.1 Overview 38

2.6.2 Tasks 39

2.6.3 Milestones 41

3 Static Features Extraction **43**
3.1 Introduction . . . 43
3.2 Training/Test split . . . 44
3.2.1 Time-based split . . . 44
3.3 Data collection and preparation . . . 48
3.3.1 Static features . . . 49
3.4 Design and implementation . . . 52

4 Clustering Windows Malware **55**
4.1 Introduction . . . 55
4.2 Pipeline . . . 56
4.2.1 Preprocessing . . . 56
4.2.2 Dimensionality Reduction . . . 57
4.2.3 Hierarchical clustering . . . 58
4.3 Results . . . 59
4.3.1 Clustering quality assessment . . . 59
4.3.2 Optimal number of clusters . . . 63
4.3.3 Other ground "truth" analysis . . . 64

5 XAI for Cluster Analysis **69**
5.1 Introduction . . . 69
5.2 Two *two-step* approaches . . . 70
5.2.1 Random Forest method based on Gini importance . . . 70
5.2.2 Local Explanations method . . . 71
5.3 Experiments . . . 72
5.4 Results and Conclusions . . . 79

6 Concept drift detection **81**
6.1 Introduction . . . 81
6.2 Multiclass Non-Conformity Measure (NCM) . . . 82
6.2.1 Random Forest Proximities . . . 82
6.3 Pipeline Overview . . . 83
6.4 Results and Analysis . . . 84
6.4.1 Alpha assessment . . . 84
6.4.2 Global threshold . . . 85
6.4.3 Concept drift . . . 86

7 Conclusions **91**
7.1 Future work . . . 92

92

Bibliography **92**

List of Figures

2.1 Create reverse shell rule from capa-rules 6
2.2 PE 32-bit format 11
2.3 Concept drift detection methods [BAK22] 18
2.4 Performance-based detector [BAK22] 19
2.5 [Lam22] 21
2.7 Figure of one-sample K-S test inspired by [Mas51]. The continuous curve represents the theoretical distribution, and the dashed curves are at distance $\pm d_\alpha(N)$. Reject unless the step-function lies entirely between the broken curves. 31
2.8 Two sample K-S test 31
2.9 CE approaches 37

3.1 Report $51.1\% - 48.9\%$ Train-Test split. 45
3.2 Report $70\% - 30\%$ Train-Test split. 46
3.3 Report 62.33%-37.67% (optimal) Train-Test split. 47
3.4 Malware submission date distribution 48
3.5 Objective function visualizations 48
3.6 Top Feature Extractor UML 53
3.7 Static Feature Extractor UML 53
3.8 Static Feature Extraction Process UML 54

4.2 `AgglomerativeClustering` over `MiniBatchSparsePCA` dimensionality reduction 60
4.3 `AgglomerativeClustering` over `UMAP` dimensionality reduction . . 61
4.4 `AgglUMAP` quality assessment - Unbalanced clusters in terms of number of samples 62
4.5 `AgglMbsPCA` quality assessment - Supervised scores tendency over the number of clusters 63
4.6 Packing analysis: Top 20 clusters by Jaccard index for increasing dendrogram levels, Jaccard index distribution along packing algorithms. 66

4.7 Malware family analysis: Top 30 clusters by Jaccard index for increasing dendrogram levels, Jaccard index distribution along malware families. 68

5.1 Random Forest-based two-step method based on *Gini importance*. . 71
5.2 Two-step method based on local explanations. 72
5.3 Feature importance distribution grouped per feature type and cluster 75
5.4 Cumulative distribution function of feature contribution per cluster 76
5.5 Feature contribution distribution per feature type (absolute and normalized). 77
5.6 Feature importances grouped for the whole dendrogram cut (n. 1500 clusters). 78

6.1 Alpha assessment: credibility score (p-value) distribution of the calibration set 84
6.2 Confidence score distribution of calibration set 86
6.3 Thresholding information: average credibility and confidence comparison, CDF of p-values for incorrect predictions 86
6.4 Credibility score distribution of testing samples from seen families . 87
6.5 Families with all drifting samples in the testing set 88
6.6 Credibility score distribution of testing samples from unseen families 89
6.7 Credibility score distribution for testing samples from emerging families with average and 100% credibility above the threshold, respectively, grouped by predicted family. 89

Chapter 1

Introduction

Modern Windows malware analysis has to deal with the enormous amount of data samples that every year are collected. In 2023, *Kaspersky* registered over 411,000 malicious files daily [Kas23], while *AV Test* more than 104 million new malware over the year [Ins23]. Due to the limitations of pure signature-based systems, and the general human effort required to maintain up-to-date databases of known malicious patterns, machine learning started to grow also in this area.

With its ability to learn from data and make predictions, emerged as a promising solution that can support automatic systems such as anti-viruses both for malware detection and family classification.

Supervised methods have been proven effective for both tasks, achieving almost perfect accuracy. On the opposite, due to the complex nature of malware, very few works were presented about clustering, and it's still considered an open field to investigate by the research community.

While previous signature-based systems had to cope with polymorphic malware, attackers in the ML domain also tried to bypass systems. By altering the structure of the code, attackers can avoid detection by ML classifiers, successfully infecting machines. Evolving malware of a known family or new malware families introduced by attackers links to a known problem in machine learning referred to as concept drift.

The scope of this thesis is to shed light on the problem of concept drift in malware classification. Experiments using real-world data have been carried out

and, leveraging state-of-the-art solutions, techniques are explored for detecting concept drift across malware families.

Parallely, the thesis explores clustering algorithms applied to malware samples using static features. Using XAI techniques, distinguishing features that characterize the clusters are also extracted to provide interpretable insights into the underlying patterns of malware.

Structure of the Thesis Chapter 2 provides a state-of-the-art overview of malware analysis, with a deeper focus on applied ML and the reasons behind its adoption. Issues on ML malware detection and classification are then explored and linked to the concept drift phenomenon. Starting from its mathematical formulation, the chapter elucidates the different solutions available in the literature for its detection.

Chapter 3 describes in depth the data extraction process that has been executed, listing the static features that were considered important by linking them to previous works done in the field. Chapter 4 experiments with clustering algorithms for grouping malware samples, and 5 applies XAI techniques for making the results interpretable.

Chapter, 6 applies an existing solution for concept drift detection with respect to malware families.

Finally, 7 draws the conclusions about the overall work, highlighting the key findings and discussing potential directions for future research and improvements.

Chapter 2

Motivation, Background, and Related Work

2.1 Malware Analysis

Malware analysis is a critical process in cybersecurity that involves dissecting and understanding malicious software to mitigate its impact and prevent future infections. This process can be broadly categorized into two methodologies: static and dynamic analysis. Both the two have strengths and weaknesses, making them complementary activities in the comprehensive analysis of malware.

2.1.1 Static analysis

Static analysis involves examining the malware without executing its code. This approach focuses on understanding the structure and composition of the malware by inspecting its code, control flow, as well as file format descriptions, such as binary sections, imports, strings, and APIs.

Static malware analysis uses a signature-based approach, which involves extracting code digital footprints and comparing them to a known list of malicious patterns.

Malware researchers can also leverage decompilation and disassembly. Tools like IDA Pro, Ghidra or Radare2 allow to convert the binary code into higher-level programming language, allowing analysts to study the malware's logic and

functionality.

As static analysis doesn't consist of running the program, it has the advantage of being a relatively fast way to detect malicious activity.

On the other hand, to foil the analysis of anti-malware systems and evade detection, malware uses packing and other forms of obfuscation, which makes static analysis even more challenging.

Signature-based detection

A 'signature' in signature-based detection is commonly known as a pattern associated with a malware sample. This pattern can be a sequence of bytes inside a file or inside network traffic that is mainly known as an attempt to evade security mechanisms, perform privilege escalation, and do some unauthorized action.

Current anti-virus technologies, among others, rely on this approach to detect malicious programs. They scan for malware by searching for patterns that match some signature contained in a database.

Signatures are usually described as YARA rules (as well as other formats like OpenIOC or Snort rules - to identify malicious network activity), that consist of a set of strings and a boolean expression through which one can determine the matching criteria for a specific malware family.

AV engines that include signature-based detection techniques include ClamAV [Cla24] and *Kaspersky* [Kas24].

Also, two other types of software useful for malware analysis that rely on signatures to detect anomalies are Capa and Suricata.

Capa

Capa is a tool developed by the FLARE's team to automatically identify malware capabilities.

It mainly consists of two components: first, a feature analysis engine that extracts features from executable files, such as strings, disassembly, and control flow, and second, a logic engine, which searches for predefined matching rules, expressed as a combination of features.

The feature analysis engine, in particular, extracts features from multiple scopes, starting from the most specific such instructions, and towards the most general (file/global).

Scope	Features
File	(sub)string, function-name, section, export, namespace, class, import, forwarded export, embedded pe, mixed mode
Function	loop, recursivecall, calls from, calls to
Basic Block	tight loop, stack string
Instruction	namespace, class, api, property, nzxor, peb access, (sub)string, bytes, offset, cross section flow, indirect call, call $+5, operand, number, mnemonic, fs access, gs access, unmanagedcall
(Global)	os, format, arch

Table 2.1: Extracted Static features from Capa

Once the features are extracted, they are passed as input to the logic engine. The latter is based on *Capa rules*, each one representing a specific executable capability considered important to recognize a malware sample. A *Capa rule* is a logical expression whose terms are one or more extracted features. They are encoded in a YAML document that contains its metadata and a tree of statements to express its logic.

Thanks to the use of FLIRT signatures, Capa can automatically differentiate between programmer code and library functions code, allowing it to focus only on the logic written by the programmer.

In figure 2.1, the rule "*Create a reverse shell*" is shown, comprising the name and namespace, which allows associating the capability to a technique or analysis category, and the logical tree with the necessary features and parameter values that have to match.

Capa is a useful tool for malware analysts, it supports 570 capability detection rules that can easily be integrated into common disassemblers like Ghidra and IDA pro as a plugin. By adhering to the rule YAML format, capabilities can be added from the database, allowing the community to contribute as new ones are discovered. Furthermore, in the recent versions, Capa supports dynamic analysis as well by processing CAPE sandbox reports.

Here there's a high-level overview of capabilities Capa 2.0 currently captures:

```yaml
rule:
  meta:
    name: create reverse shell
    namespace: communication/c2/shell
    authors:
      - moritz.raabe@mandiant.com
    scopes:
      static: function
      dynamic: thread
    att\&ck:
      - Execution::Command and Scripting Interpreter::Windows Command
      ↪ Shell [T1059.003]
    mbc:
      - Impact::Remote Access::Reverse Shell [B0022.001]
    examples:
      - C91887D861D9BD4A5872249B641BC9F9:0x401A77
  features:
    - or:
      - and:
        - match: create pipe
        - api: kernel32.PeekNamedPipe
        - api: kernel32.CreateProcess
        - api: kernel32.ReadFile
        - api: kernel32.WriteFile
      - and:
        - match: host-interaction/process/create
        - match: read pipe
        - match: write pipe
      - and:
        - match: create pipe
        - match: host-interaction/process/create
        - or:
          - basic block:
            - and:
              - count(api(SetHandleInformation)): 2 or more
              - number: 1 = HANDLE_FLAG_INHERIT
          - call:
            - and:
              - count(api(SetHandleInformation)): 2 or more
              - number: 1 = HANDLE_FLAG_INHERIT

```

Figure 2.1: Create reverse shell rule from capa-rules

1. Host Interaction describes program functionality to interact with the file system, processes, and the registry;

2. Anti-Analysis describes packers, Anti-VM, Anti-Debugging, and other related techniques;

3. Collection describes functionality used to steal data such as credentials or credit card information;

4. Data Manipulation describes capabilities to encrypt, decrypt, and hash data Communication describes data transfer techniques such as HTTP, DNS, and TCP.

Limitations of signature-based detection

While signature-based methods detect similar versions of known malware families with a small error rate, they become insufficient as an ever-increasing number of new malware samples are being identified [VB18]:

1. They are relatively easy to bypass, as attackers can alter some parts of the program. To do so, malware writers invented techniques like server-side polymorphism [Kas21]. Polymorphic malware utilizes dead-code insertion, subroutine reordering, encryption, and additional mechanisms for automatically transforming a file's content (while retaining its functionality) in order not to match any AV signature [CH18];

2. They are generally specific and cannot detect malware even if it uses the same functionality;

3. They don't protect against new threats whose signatures are not listed in the database;

4. The signature database has to be maintained up to date, requiring human effort.

These are some of the motivations that pushed researchers to investigate machine-learning techniques for malware detection and malware family classification.

2.1.2 Dynamic analysis

Dynamic analysis, as opposed to static, requires the program to be executed. Typically, the execution is done inside a controller environment such as sandboxes, and their activity is monitored to study their runtime behaviour.

Dynamic analysis overcomes the limitations of static analysis on packing, as if the malware is packed the unpacking procedure is carried out before it gets executed. On the other hand, dynamic analysis is an expensive process as it requires building a controlled environment and monitoring the program activity for a significant amount of time.

Furthermore, recent works have measured that 40%–80% of modern malware use at least one evasive technique to detect and thwart instrumented environments (e.g., debuggers and virtual machines) [MNK+21, BKB17]. In these cases evasion techniques must be handled as well to increase the likelihood of the malware detonating, increasing the difficulty of dynamic analysis.

Since dynamic analysis involves identifying malware behaviours, it is particularly helpful for uncovering threats not previously documented, such as *zero-day* threats. These threats are not usually found using static malware analysis, which is why it is so crucial to keep organizations secure [Jon23].

Dynamic analysis involves:

- Monitoring system calls, such as operations to create or modify a file, open network connections or write specific registries. Also, they include analysis of process/thread creation, or mutexes used;
- Network analysis. Malware often contacts remote servers to send and receive data, such as for exfiltrating data;
- Dynamic code analysis, which involves tracing the execution flow of the program;
- Memory analysis. Researchers trace memory during the program execution to identify hidden activities.

2.2 Machine Learning for Malware Analysis

Modern Windows malware analysis has to deal with the enormous amount of data samples that every year are collected. In 2023, *Kaspersky* registered over 411,000 malicious files daily [Kas23], while *AV Test* more than 104 million new malware over the year [Ins23].

Due to the limitations of pure signature-based systems listed in 2.1.1, and the general human effort required to maintain up-to-date databases of known malicious patterns, it is clear that these solutions do not scale well.

Given these motivations machine learning started to grow also in this area. With its ability to learn from data and make predictions, emerged as a promising solution that can support automatic systems like anti-viruses both for malware detection and family classification.

Both supervised and unsupervised methods are currently being investigated by the research community.

2.2.1 Malware detection and family classification

In the Windows OS, supervised methods have been proven effective for both binary and multi-class classification problems, achieving almost perfect accuracy.

In [SS15], they applied binary classification using a set of 997 virus files and 490 benign executables, by comparing models using only static features, only dynamic ones, and both types of features. Using as static features some printable strings and as dynamic sequences of API calls, using the integrated static and dynamic method they could reach an accuracy of 98.7%.

In [SS15], they trained a Random Forest model using the Malimg Dataset, consisting of 9,342 malware samples of 25 different malware families. The dataset has been split into 80% training and 20% testing, and 10-fold cross-validation is performed, achieving an overall accuracy on the testing set equal to 95.26%.

In [MLJ+21] they categorized learning-based PE malware family classification techniques into three classes based on the executable format. The first category groups methods that involve converting the executable into an image format and then applying image classification. The second one makes use of sequential models

from NLP applied to a sequence of binary code. The third one consists of the decompilation of the malware and adopting graph structure analysis on the control flow graph (CFG) of the assembly code.

Also in [GSCK23], a taxonomy of the diverse machine learning algorithms is given. Ten machine learning algorithms for malware detection were considered, based on the analysis of 77 selected studies. They found that SVM is the most widespread malware detection algorithm, with 24%, followed by DT, with a percentage of 15%. Other algorithms include Naïve Bayes, KNN, Linear and Logistic regression.

Static features

Static features in the WindowsOS consist of parsed and format-agnostic features. The former includes information gathered from PE executables, while the latter consists of general features that are not specific to PE files, and can be extracted from any file.

PE format is a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code. This includes dynamic library references for linking, API export and import tables, resource management data, and thread-local storage (TLS) data.

Analogous formats to PE are ELF (used in Linux and most other versions of Unix) and Mach-O (used in macOS and iOS).

The PE data structures include DOS Header, DOS Stub, PE File Header, Image Optional Header, Section Table, Data Dictionaries, and Sections.

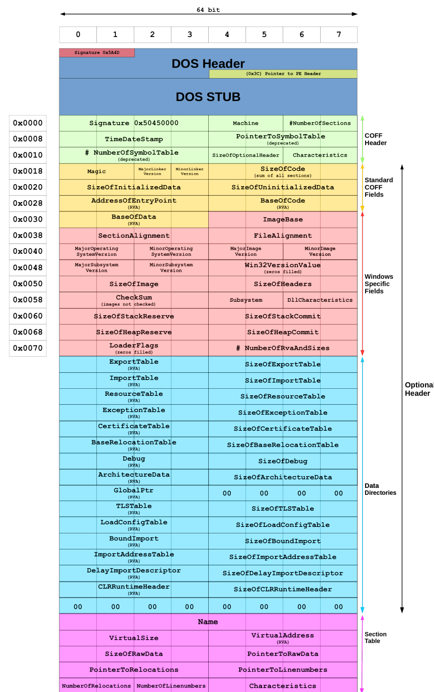


Figure 2.2: PE 32-bit format

Public datasets

The EMBER dataset [AR18] is an example of a dataset that includes PE file information. It is an extensively used dataset used by the research community in many works, composed of static features only. In this work, PE file information includes:

- General file information. They are contained in the PE File Header and consist for example of the virtual size of the file and the number of imported/exported functions;
- Header information from the COFF header. Some features contained in this part of the PE standard are timestamp in the header, the target machine (string), and a list of image characteristics (list of strings);
- Imported and exported functions, the former ones extracted from the import

address table;

- Section information, which contains properties of each single section.

EMBER dataset also includes agnostic features, such as raw byte histogram, byte entropy histogram, and strings. All these features have been proven effective for binary and/or family classification. For example, strings have been used in [SMdlR+14] to perform family classification. They first created two groups of printable strings, one extracted using the term frequency-inverse document frequency (*tf-idf*) function and the other using the prominent strings extracted from the vocabulary. They achieved an accuracy of 91.02% by considering the entire vocabulary and an accuracy of 80.52% by considering the top 20 strings from *tf-idf* for each malware family.

In terms of agnostic features, also binary n-grams have been extensively used for both binary and family classification. They were originally proposed in [JM11], but have been subject to some criticism [RZC+16] later. In the latter paper, they show that previous studies on n-gram features have flows that lead to an overestimation of classification accuracy, that most of the n-grams stem from string features and lastly that n-gram features promote overfitting.

In [DHA+23] they proposed another dataset based on PE malware samples. It is the most diverse labeled malware dataset in terms of families to date. It comprises an equal amount of samples for each malware family so that no one is under-represented or over-represented. The balanced dataset is composed of 67000 samples, each of which belongs to one of 670 malware families.

The malware were collected from **VirusTotal** [Vir]. Above the others, its API allows to retrieve a real-time stream of all files submitted. The dataset samples were recorded from 83-non consecutive days between August 2021 and March 2022.

The work focuses on the extraction of both static and dynamic features according to those provided by the literature and experiments are done in order to compare the classification performance of different between these different groups of features: only static, only dynamic and combined.

The study shows that for binary and family classification the F1 score is higher for only using static features compared to only dynamic ones. The combination of static and dynamic features brings marginal improvement in the scores, but the

number of perfectly classified families nearly duplicates.

Experiments are done to study the impact of packers and protectors. Indeed, they did not clear the dataset from packed malware but instead used this information to analyze the results. These latter show that there isn't a relevant decrease in accuracy on the classification of those samples. This means that although these technologies function well to deter static analysis (in particular reverse engineering), they do not significantly affect ML classifiers, which are still able to successfully identify byte-level signatures.

They also tested the model using data which includes new families and has a different distribution from the training data. This scenario is known as **OOD** (Out-Of-Distribution) test.

- The accuracy of binary classification using only static or dynamic features deteriorates significantly;
- The accuracy deterioration is more significant in the non-uniform training setting than that in the uniform setting: classifiers built on very unbalanced datasets perform equally well when tested on data with the same unbalanced distribution, but generalize poorly on other datasets, likely because many families are underrepresented in the training and thus the model fail to proper capture them.
- Static features generalize poorly to unseen families, while dynamic performs better: "This is due to the nature of the features themselves: static information can precisely pinpoint only known samples, while dynamic behaviour can better generalize also to unknown ones."

2.2.2 Clustering malware

Clustering is an unsupervised ML method that makes use of distance metrics to identify clusters of data.

In the malware analysis literature, few works exist in this area, focusing on identifying clusters of malware based on either their behaviours (dynamic features) or static features.

Such an example is [BOA+07]. In this study, the researchers developed a dynamic method that runs malware within a virtual environment to record its behavioural fingerprints. These fingerprints document all malicious activities carried out by the malware, such as file modifications and process creation. The recorded behaviours were then compared using single-linkage hierarchical clustering with normalized compress distance (NCD) as the distance metric. This innovative approach at that time offered a novel perspective on the relationships between different malware samples. Additionally, it proved effective in enhancing the understanding and categorization of existing malware.

In [BCH+09], they developed a novel clustering technique based on dynamic analysis.

The system monitors the execution of a malware sample in a controlled environment and builds a malware behaviour profile. This profile contains information about the OS objects that the program operates on, as well as the type of operations and dependencies. Then, leveraging LSH clustering, they were able to compute an approximate, single-linkage hierarchical clustering from a set of behavioural profiles such that samples that exhibit similar behaviour are combined in the same cluster.

The proposed technique overcomes the one of the previous paper ([BOA+07]). As they state, a clustering approach that uses those profiles either produce results that are significantly less precise (by using the Jaccard index and LSH), or it does not scale to real-world datasets (when using NCD).

2.2.3 Issues of ML-based malware detection

While malware detection and classification methods using machine learning have been proven to have great accuracy, problems started to rise in this area too, where attackers try to modify programs to bypass detection.

This phenomenon is known in the literature as concept drift. This drift arises due to the continuous evolution of malware techniques, such as obfuscation, polymorphism, and the development of new attack vectors. Additionally, legitimate software updates and changes in user behaviour can alter the patterns these models rely on for accurate detection. Consequently, as the characteristics of malware

and benign software evolve, the effectiveness of static models degrades, leading to increased false positives and false negatives. Addressing concept drift requires ongoing model updates and adaptive learning strategies to maintain robust malware detection capabilities.

2.3 Concept drift

Concept drift refers to the fact that the underlying posterior probability distribution of a target variable of a classification problem changes over time. Initially proposed by [SG04] in 1986, concept drift can occur due to various reasons such as evolving user preferences, seasonal effects, changes in the environment, or new patterns emerging in the data. When concept drift occurs, a model trained on historical data may become less accurate or even obsolete, as the patterns it learned no longer reflect the current state of the data.

Managing concept drift requires techniques such as online learning, ensemble methods, drift detection, and windowing techniques to ensure models remain accurate and relevant in dynamic environments.

This section starts by describing the role of concept drift in the malware domain, then its definition is given as well as its categorization. Then, state-of-the-art techniques for concept drift detection are introduced.

2.3.1 Concept drift in malware domain

In the malware domain, concept drift is well known and it's an effective problem that anti-virus systems have to assess. As stated in the previous sections, malware can indeed evolve to bypass systems, even machine learning-based solutions. One example is adversarial attacks.

Existing solutions aim to periodically retrain the model whenever drift is detected. However, if the model is retrained too frequently, there will be little novelty in the information obtained to enrich the classifier. On the other hand, a loose retraining frequency leads to periods of time where the model performance cannot be trusted [JSD+17].

2.3.2 Mathematical formulation

Concept drift 1. *Assuming that P_t represents the joint probability distribution between input variable x and target variable y at time t_0 and P_{t+w} represents the joint probability distribution between x and y at time $t+w$, then concept drift will occur if the following equation holds:*

$$\exists x : P_t(x,y) \neq P_{t+w}(x,y) \tag{2.1}$$

Also, authors presented an additional constraint to the definition in 2.1 which states that the new concept should be valid in at least two time periods:

$$w = \tau_d(i+1) - \tau_d(i) > 1, \forall i \tag{2.2}$$

where $\tau \in Z+$ is the time point, and $d(i)$ denotes the time point order of the i-th concept drift appeared in the system. This rule ensures that the identified concept is rather a real pattern and not detected by chance. That's to say: constraint in 2.2 allows to differentiate between a new pattern and outliers that last momentarily [BAK22].

According to the Bayesian decision theory, the joint distribution $P(x,y)$ in equation 2.1 can be rewritten as:

$$P_t(X,y) = P_t(y|X) * P_t(X) = P_t(X|y) * P_t(y) \tag{2.3}$$

2.3.3 Concept drift types

Concept drift can be categorized by the probabilistic source of change and the pattern of arrival:

Probabilistic source of change

According to the characteristics of joint distribution described in equation 2.3, concept drift can have multiple causes to occur, based on the probabilistic source:

- $P_t(y|X)$ denotes the posterior probability distribution of the target labels. Here the statistical properties of the target variable change over time. In

other terms, the relationship between the input and target variable that a model learnt no longer holds and therefore it cannot make valid predictions anymore;

- $P_t(X)$ is the input data probability distribution. Also referred to refers to *covariate shift*: the distribution of input data that an ML model was trained on differs from the distribution of the data the model is deployed to;

- $P_t(y)$ denotes the prior probability distribution of the target labels. Also known as *prior-probability shift* is the change of the distribution of the classes over time.

Note that using the Bayes rule it's possible to state that if one probability source changes, it also leads at least another one to differ. For example, if $P_t(y|X) \neq P_{t+w}(y|X)$:

$$\frac{P_t(X|y)P_t(y)}{P_t(X)} = \frac{P_{t+w}(X|y)P_{t+w}(y)}{P_{t+w}(X)} \tag{2.4}$$

Transition of change

Concept drift can be categorized based on the pattern of how the drift evolves in the system [BAK22]:

- Sudden. Occurs when the target distribution changes from one concept to another abruptly at a point in time; An example of this type of drift is the start of the COVID-19 lockdown in March 2020, which drastically changed population behaviors all over the world;

- Gradual. Occurs when the target distribution changes progressively from one concept to another;

- Recurring. Occurs when a precedently-seen concept reappears again after a time interval. A typical example of this kind of fluctuation is the yearly change in seasons, which pushes consumers to purchase warm coats during the colder months, reduces demand as temperatures increase in the spring, and then resumes in the fall. The difference with the gradual drift is that

in this case, the old concept reoccurs periodically, while in gradual drift the old concept fades out;

- Incremental. Occurs when a new concept replaces the old one slowly in a continuous manner. It is considered a subtype of gradual drift, but here there is no obvious boundary that separates the occurrence of the different concepts;

2.3.4 Concept drift detection

Concept drift detection refers to the methodology that helps to determine the time instant when a change arises in the properties of the target object. It is a component of the concept drift handling framework that triggers the adaptation pipeline, whose purpose is to maintain up to date the current machine learning model in order to avoid its degradation.

In [BAK22] they categorized concept drift detectors into different approaches: Data distribution-based, performance-based, error rate-based, hybrid and contextual-based approaches.

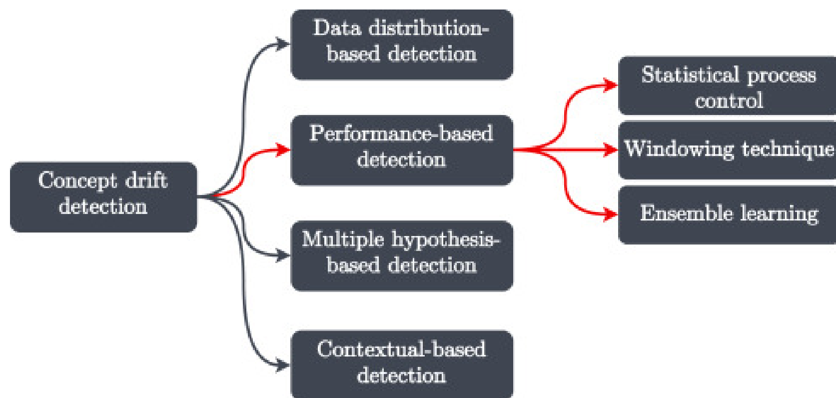


Figure 2.3: Concept drift detection methods [BAK22]

Performance-aware drift detectors

Performance-aware drift detectors are the most adopted ones and typically trace deviations in the online learner's output error to detect changes.

In particular, performance-based detectors are based on the **PAC** (Probability Approximately Correct) learning model. In a stationary distribution context, the error rate is expected to decrease as the learner encounters more examples. Therefore, if the error rate stops decreasing, it indicates that the learned relationship has become obsolete, signalling concept drift.

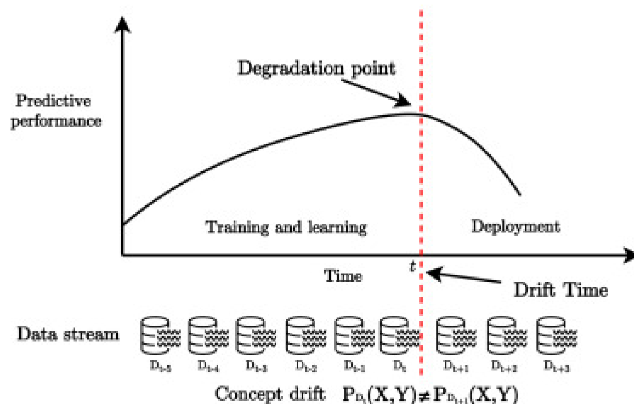


Figure 2.4: Performance-based detector [BAK22]

These methods can be grouped according to the strategy used to detect performance drops: statistical process control, windowing techniques, and ensemble learning.

- Statistical process control (SPC). They monitor the quality of the model based on the evolution of its online error rate. Typically a threshold is set and drift is detected if the model exceeds this significant test level. An example of such detectors is the **DDM (Drift Detection Method)** [GMCR04].

  In DDM, whenever a change in the error rate is detected, either drift is signalled to the user or the algorithm is warning him that a change may occur in the near future. DDM approach is based on the idea that the probability of misclassification (error rate) at any time can be monitored to detect significant changes.

  The error rate in particular is modeled as a Bernoulli random variable with Binomial distribution. It monitors p_t , its probability at time t, and the standard deviation s_t as: $s_t = \sqrt{p_t(1-p_t)/i}$.

The detection threshold is computed in the function of two statistics, obtained when $p_i + s_i$ is minimum:

- $p_{\min}$: minimum recorded error rate;
- $s_{\min}$: minimum recorded standard deviation

At instant t, the algorithm computes:

- p_i: error rate at instant i;
- s_i: standard deviation at instant i

Then:

- If $p_i + s_i \geq p_{\min} + 2 * s_{\min}$ warning zone is notified;
- If $p_i + s_i \geq p_{\min} + 3 * s_{\min}$ drift is detected.

There are multiple extensions of DDM that enhance its performance to solve more general tasks. In the following figure, other types of drift detectors based on SPC are shown.

Method	Idea
Drift Detection Method (DDM)	Compares the probability of misclassification p + the standard deviation s at time t with their min values
Adaptative Online Incremental Learning (AOIL)	Monitors the mean and variance of the loss error
Spectral Entropy Drift Detector (SEDD)	Computes spectral entropy to check the fluctuation's magnitude during the learning process
EMWA for Concept Drift Detection (ECDD)	Adjusts EMWA (Exponentially Weighted Moving Average) to monitor changes in the error rate
Convolutional Neural Network with Change Detection Tests (CNN-CDT)	Monitors the classification error using CUmulative SUM (CUSUM) test
AUC-based family	Uses the AUC and variants to detect drift in imbalanced multi-class settings. Variants: - Prequential Multi-class AUC (PMAUC) - Weighted AUC (WAUC) - Equal Weighted AUC (EWAUC)
Online Sequential Extreme Learning Machine (OS-ELM)	Detects drift by analyzing the dissimilarities between the output weights of the model for every chunk of new data

Figure 2.5: [Lam22]

- Windowing techniques divide the data stream into sliding windows based on the size of the data or the time interval, monitoring the performance of the latest window introduced in the system and comparing it with a reference one.

  An example of a windowing technique is ADaptive WINdowing (**ADWIN**) [ASLP07]. ADWIN manages a dynamic window of recent data assuming stability in distribution. It splits this window into two sub-ones (W_0, W_1) to detect changes, comparing their averages. When distribution change is detected, ADWIN replaces W_0 with W_1 and initializes a new W_1.

  In particular, when processing a stream data point, ADWIN first adds the tuple to the adaptive window. Then, it iterates over all possible combinations of two sliding subwindows W_0 and W_1. If drift is detected from two subwindows, the oldest data point from the adaptive window is deleted.

ADWIN is based on Hoeffding bound to measure the change between two means μ_0, μ_1 of two 'sufficiently' large windows W_0, W_1:

$$|\mu_0 - \mu_1| > 2\epsilon_{\text{cut}}$$

where ϵ_{cut} is the optimal cut:

$$\epsilon_{\text{cut}} = \sqrt{\frac{1}{2m} \ln \frac{4|W|}{\delta}}$$

m is the harmonic mean of the two windows, δ pre-defined confidence parameter.

Then, the algorithm analyzes the con- tent of the adaptive window to identify concept drifts. To that end, Adwin iterates over all possible combinations of two large enough sliding subwindows, as shown in Figure 1. If the value distributions of the two subwindows are different enough, Adwin detects a concept drift and removes the oldest tuple from the

Drift detection using autoencoders

In [JRA20], they proposed a drift detector based on autoencoder architecture. Autoencoders are neural networks that are trained to reconstruct the input data from a latent representation with lower dimensionality. If properly trained i.e. they are able to capture the hidden patterns of the data, changes in their cost function might occur if a sample is drifted from the training ones. Two cost functions are applied in this paper, the cross-entropy and the reconstruction error. Preliminary experiments show that autoencoders can both detect gradual and sudden concept drift.

2.4 Hypothesis testing

Hypothesis testing is a fundamental statistical method that allows to make inferences about the population with a sample. It provides a structured process for

determining whether there is enough evidence in a sample to support a particular belief or hypothesis about the population.

More generally, hypothesis tests help to determine the effect of an independent variable on a dependent variable.

For example, suppose one wants to see if there is a difference in age between people who use different programming languages: Python, Java, C++. The independent variables are the programming languages, while the dependent the age.

Statistical hypotheses then allow us to assess the following, i.e. either accept:

- Null Hypothesis H_0: there's no relationship between the independent and dependent variables. In other words, the differences in the populations occur by chance rather than an actual effect, implying that there is no significant underlying cause for the differences observed between the groups.

  In the example it reflects that the difference in ages between the programming languages is not significant;

- Alternative Hypothesis H_a or H_1. Conversely, there is a statistically significant relationship between independent and dependent variables.

In the literature, there exists a bunch of statistical hypothesis tests, each of which has a different measure to assess the difference in the distributions and might be suitable for different kinds of data. Each measure is then combined with the p-value.

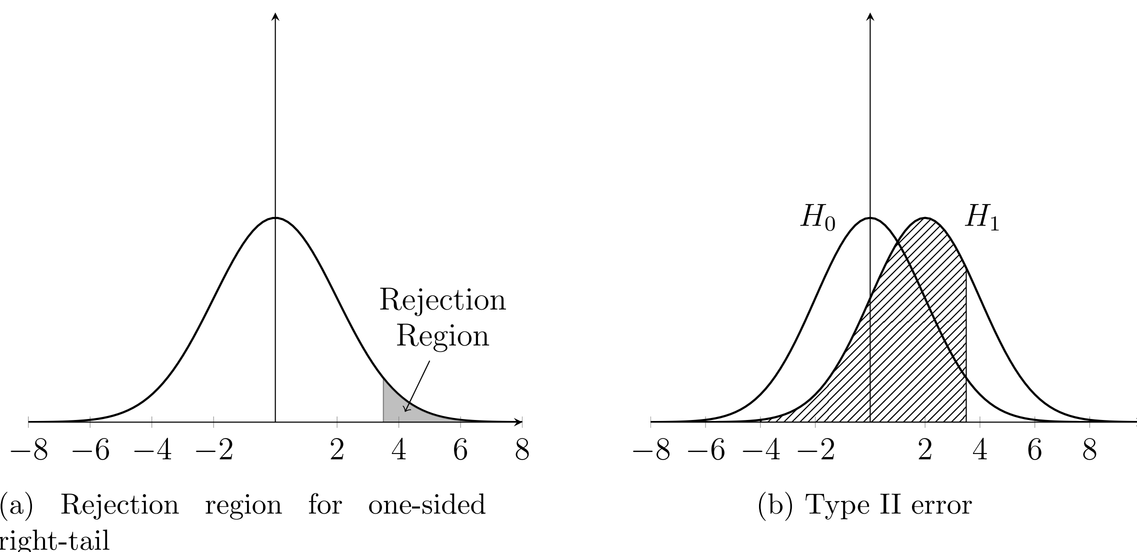
2.4.1 p-value

Concept drift 2. *In statistical significance testing, the p-value is defined as the probability under the null hypothesis to obtain a real-valued statistic equal to or more extreme than what was actually obtained. That is:*

- $p = Pr(T \geq t|H_0)$ *for a one-sided right-tail test-statistic distribution;*

- $p = Pr(T \leq t|H_0)$ *for a one-sided left-tail test-statistic distribution;*

- $p = 2\min\{Pr(T \geq t|H_0), Pr(T \leq t|H_0)\}$ *for double tail event.*



(a) Rejection region for one-sided right-tail

(b) Type II error

The smaller the p-value the more the statistic is dissimilar to the already observed ones and most likely the data comes from a different distribution. The null hypothesis H_0 is then rejected if any of these probabilities is less than or equal to a small, fixed, but arbitrarily pre-defined, threshold value α, which is referred to as the level of significance.

Supposing that the data follows a normal distribution, an example of rejection area is shown in 2.6a. As the threshold is decreasing as we most likely capture rare observations.

The threshold value α is determined on the consensus of the research community that the investigator is working in, and it's usually fixed to 0.05(5%). Some explanations about the choice of this value are given in [MAS14, GP94].

Other two important things to mention are *Type I* and *Type II* errors. If we have a decision rule that says to reject H_0 whenever an observation falls in the highest 5% of the distribution, and we do accordingly for a testing sample, we'll have 5% chance of erroneously rejecting the null hypothesis. This kind of error (rejecting when in fact it is true) is called a Type I error (false positive) [How07].

One might argue that a 5% chance of error is too risky and recommend tightening our criteria significantly, like rejecting only the lowest 1% of the distribution. While this is a valid approach, it's crucial to recognize that the more stringent the criteria, the greater the chance of committing the opposite i.e. failing to reject

a false claim that is actually true. This error is known as a Type II error (false negative), symbolized by β.

Type I and II errors are shown in figure 2.6a and 2.6b respectively. In the first one the rejection region of 5% threshold is shown, stating that there's 5% probability of Type I error to occur. In the second one, it's possible to see that the too stringent threshold is actually causing the null hypothesis to be accepted, even if the data that eventually will come up is sampled from a different distribution.

2.4.2 Statistical hypothesis tests

There exist multiple statistical tests in the literature, each of which produces in output a measure of how two or more distributions differ from each other. Their value is then combined with the p-value, whose definition was given in the previous section. This section is instead focused on the different types of statistical hypotheses: the definitions of some of them are introduced, their pros and cons are discussed as well as their application in the machine learning area.

ANOVA test

ANOVA (Analysis of Variance) is a parametric statistical test used to determine if there are significant differences between the means of two or more groups.

The test compares the means across the groups and calculates an F-statistic and p-value to determine if the differences are statistically significant.

Analysis of variance is the extension of the T-test for independent samples where there are more than 2 groups. There exist three types of ANOVA tests: One-Way Analysis of Variance, Two-Way Analysis of Variance and N-Way Analysis of Variance (MANOVA).

The F-statistic is defined as the ratio of the variance between group means to the variance within the same group.

If the group means are drawn from populations with the same mean values, the variance between the group means should be lower than the variance of the samples, following the central limit theorem. A higher ratio therefore implies that the samples were drawn from populations with different mean values [SSD].

$$F = \frac{\textit{between-group variability}}{\textit{within-group variability}}$$

In particular: F = MBS/MSE where MBS = Mean sum of squares between the groups, MSE = Mean squares of errors. MSB = SSB/(k–1), MSE = SSE/(n–k) where SSB is the sum of squares between groups and SSE sum of square errors within groups.

$k - 1$ and $n - k$ are the normalization terms of SSB and SSE respectively. They account for the fact that the row dispersion values depend on the number of samples n and number of groups k, so the overall measure would be misleading for increasing number of groups. As k increases:

- SSE tends to decrease because more groups generally lead to smaller within-group variability;
- Conversely, SSB might increase because more group means to consider results in a higher between-group sum of squares.

In summary, the term $\frac{k-1}{n-k}$ adjusts these dispersion measures, allowing a fair comparison.

There are two methods of concluding the ANOVA hypothesis test, both of which produce the same result:

- The textbook method is to compare the observed value of F with the critical value of F determined from tables. The critical value of F is a function of the degrees of freedom of the numerator and the denominator and the significance level (α). If $F \geq F_{\text{Critical}}$, the null hypothesis is rejected;
- The computer method using p-value.

The ANOVA test, as it's a parametric test, makes some assumptions about the underlying data. Consequently, if the following criteria are not satisfied, the test is deemed unsuitable for application:

- Normality: the data distribution of each group should be normally distributed;

- Homogeneity of variance: The variance of the data within each group should be equal;
- Independence: the observations within each group should be independent;
- Random sampling: the observations should be sampled randomly and are independent of each other;

ANOVA finds multiple applications in machine learning. Some examples are:

- Feature selection: ANOVA helps identify whether there are significant differences between the means of the target variable for different values of a categorical feature. If the null hypothesis is rejected, it means the variable is helpful in discriminating between different values of the target variable;
- Cross-validation: CV-ANOVA combines the principles of cross-validation and ANOVA to assess the statistical significance of differences in predictive performance among models or parameter settings evaluated across multiple folds of the data;
- Data drift: ANOVA can be used to assess if there's a data drift between the training and testing set. If present, it may be the cause of bad performances of machine learning models. However, ANOVA cannot be used in such cases where one wants to determine if an incoming data point comes from the same distribution the model was trained on, as it only compares the difference between groups.

Chi-squared test (χ^2-test)

The Chi-squared test (also known as the Pearson Chi-square test) is a non-parametric test for categorical variables. As such, it is distribution-free, meaning that it doesn't make assumptions about the underlying data distribution.

In particular, non-parametric tests should be used when any one of the following conditions pertains to the data [McH13]:

- The level of measurement of all the variables is nominal or ordinal;

- The sample sizes of the study groups are unequal (some parametric tests require groups of equal or approximately equal size);

- The original data were measured at an interval or ratio level, but violate one of the following assumptions of a parametric test:

  - The distribution of the data was seriously skewed or kurtotic (parametric tests assume an approximately normal distribution of the dependent variable);
  - The data violate the assumptions of equal variance, also known as homoscedasticity;
  - For any of a number of reasons, the continuous data were collapsed into a small number of categories, and thus the data are no longer interval or ratio.

Concept drift 3. *Chi-square is a measure of how far the observed counts are from the expected counts. In particular, it's defined as:*

$$\chi^2 = \sum \frac{(O - E)^2}{E}$$

Where O is the actual count of cases in each cell of the table, E is the expected value and the sum is over all the possible values of the categorical variable.

The test is used to determine if there is a statistically significant difference between the counts of the observed and expected values.

There are two types of Pearson Chi-square test:

- Chi-square goodness of fit: it's used whether one wants to test whether the observed frequency distribution of a categorical variable differs from the expected frequency distribution;

- Chi-square test of independence: mainly used when one wants to test if two categorical variables are related to each other.

Frequency distribution is usually built through a frequency distribution table, commonly known as contingency table.

As with all tests, Chi-square makes assumptions about the underlying data, some important ones are[McH13]:

- The data in the cells should be frequencies, or counts of cases rather than percentages or some other transformation of the data;
- The categories of the variables are mutually exclusive;
- Each subject may contribute data to one and only one cell in the χ^2. If for example the subject is tested multiple times over time, χ^2 cannot be used;
- The study groups must be independent. This means that a different test must be used if the two groups are related;
- A very large number of cells (over 20) can make it difficult to meet the assumption below and to interpret the meaning of the results;
- The value of the cell expected should be 5 or more in at least 80% of the cells, and no cell should have an expected of less than one.

Kolmogorov-Smirnov (KS) test

The Kolmogorov-Smirnov (K-S) test is a non-parametric statistical test used to determine whether a sample comes from a specified distribution or to compare two samples to assess if they come from the same distribution.

The one-sample K-S test, also known as the **Kolmogorov-Smirnov Test for Goodness of Fit** is based on the maximum distance between the empirical distribution function (EDF) of the sample and the cumulative distribution function (CDF) of the reference distribution.

The main idea is the following: suppose that a population is thought to have some cumulative frequency distribution function, F. The cumulative step function of a random sample (that is the empirical distribution function associated with the empirical measure of the random sample) of N observations is expected to be fairly close to F. If it's not close enough this is evidence that the hypothetical distribution is not the correct one [Mas51].

The inner workings of **two-sample K-S test** are somewhat similar, with the difference that the distance being measured is between both the empirical distribution functions of the two samples.

The K-S test is particularly useful because it makes no assumptions about the underlying distribution of the data, making it a versatile tool in statistical analysis. The test statistic, known as the D-statistic, measures the largest absolute difference between the EDF and the CDF, or between the EDFs, and its significance is assessed using critical values or p-values.

Concept drift 4 (K-S Test for Goodness of Fit (One-sample K-S test))**.** *Given the population cumulative distribution function F, and the cumulative step-function of n i.i.d. observations $F_n(x) = k/n$ where k is the number of observations less than or equal to x, the Kolmogorov–Smirnov statistic is defined as:*

$$D_n = \sup_x |F_n(x) - F(x)|$$

To determine whether the observed D-statistic is significant, it is compared to a critical value: for the one-sample K–S test, the null hypothesis is rejected at level α if $D_n > \sqrt{-\frac{1}{2n} \ln\left(\frac{\alpha}{2}\right)}$ where n is the size of the sample.

Concept drift 5 (Two-sample K-S test)**.** *Given the cumulative step-functions of n i.i.d. observations of two variables X_1 and X_2, the Kolmogorov–Smirnov statistic is defined as:*

$$D_{n,m} = \sup_x |F_{1,n}(x) - F_{2,n}(x)|$$

For the two-sample K–S test, the null hypothesis is rejected at level α if $D_{n,m} > c(\alpha)\sqrt{\frac{n+m}{nm}}$ where n and m are the sizes of first and second sample respectively and $c(\alpha)$ is a constant that depends on the significance level α.

Due to its sensitivity to differences in both the location and shape of the empirical cumulative distributions, the K-S test is widely applied in goodness-of-fit testing, model validation, and comparison of sample distributions across various scientific disciplines.

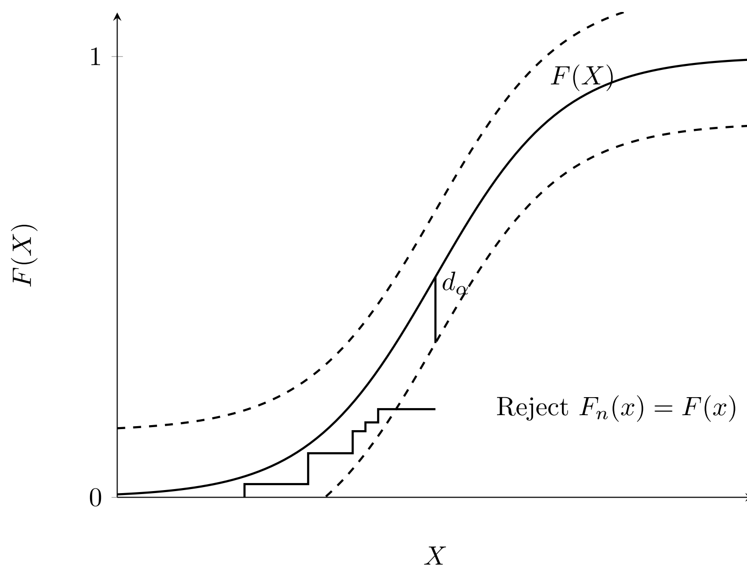


Figure 2.7: Figure of one-sample K-S test inspired by [Mas51]. The continuous curve represents the theoretical distribution, and the dashed curves are at distance $\pm d_\alpha(N)$. Reject unless the step-function lies entirely between the broken curves.

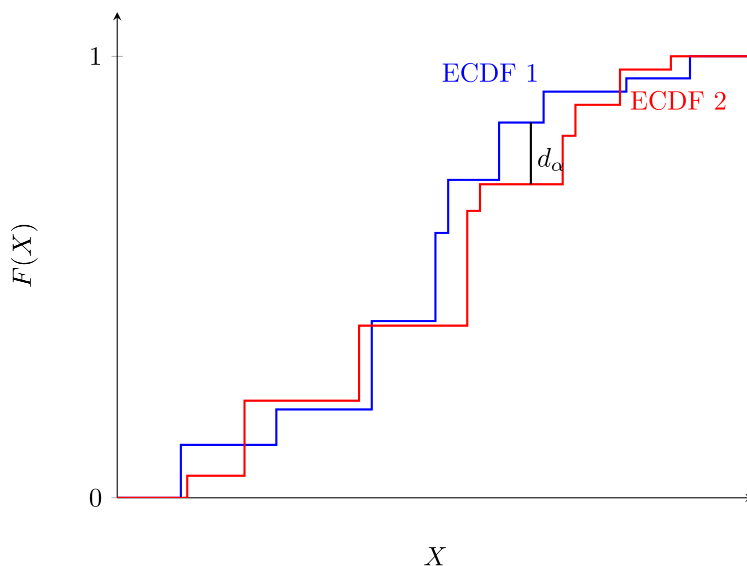


Figure 2.8: Two sample K-S test

2.5 Conformal Evaluation

In Transcend [JSD+17], they proposed a concept drift detection technique based on Conformal Prediction theory, known as Conformal Evaluation.

Well-known approaches for qualitative assessment of decisions of a learning model are based on the probability of a data sample fitting in a candidate class. However, since probabilities need to sum up to 1, it is likely that for previously unseen samples not belonging to any known class, the probability may be artificially skewed and the sample will be classified as a known class.

The approach proposed in this paper is instead based on statistical techniques to measure the model performance, showing that it outperforms the probabilistic ones.

2.5.1 Conformal Prediction theory

Conformal evaluator design is grounded in the theory of Conformal Prediction, a method for providing predictions that are correct with some guaranteed confidence.

Concept drift 6 (Conformal Prediction)**.** *Given a classifier g, a new example $z = (x, y)$, and a significance level ϵ, a conformal predictor produces a prediction region: a set of labels in the label space Y that is guaranteed to contain the correct label y with probability no more than $1 - \epsilon$.*

To calculate this label set, the conformal predictor relies on a nonconformity measure (NCM) derived from g, used to generate scores representing how dissimilar each example is from previous examples of each class.

To quantify this relative dissimilarity, p-values are computed by comparing the nonconformity scores between examples.

2.5.2 Non-Conformity Measure

Rather than making predictions, conformal evaluators borrow the same statistical tools (i.e., nonconformity measures and p-values) but use them to evaluate the quality of the prediction made by the underlying classifier g.

To be able to detect a drifting sample a measure to assess the dissimilarity of a new point to the previously encountered point is needed.

The non-conformity measure is derived from classifiers, meaning that they measure how "strange" a prediction is according to the different possibilities available.

The more strange the prediction is, the more the data point is different to the previously encountered points of that class.

Concept drift 7 (Non-Conformity Measure)**.** *Non-conformity measure (NCM) is a real-valued function that expresses how different a point z is to a bag of points $B = \wr z_1, z_2, \ldots, z_n \int$:*

$$\alpha_z = A(B, z)$$

In the literature, for each type of classifier, there exist multiple Non-Conformity Measures. Some examples are:

- In case of **Nearest Cluster Centroid classifier**, $A(B, z)$ is the euclidian distance from z to the data centroid of the points in B:

$$A(B, z) = d(z'(B), z)$$

- In case of **SVM**, $A(B, z)$ is the negated absolute distance from z to the learnt hyperplane. The more the point z is near the decision boundary, the more is dissimilar to the other points of that class and most likely be misclassified;
- Using **Random forest classifiers**, the NCM is defined as the percentage of decision trees giving a discordant prediction.

Once the NCM for each training point is computed, the p-value of an incoming point z aims to assess the proportion of training points that are more non-conformal compared to z:

$$S = \{A(B \backslash \{z\}, z) : z \in B\}$$
$$p_z = \frac{|\alpha \in S : \alpha \geq \alpha_z|}{|S|}$$

If p_z falls above a given significance level the null hypothesis is disproved and $\hat{y}_z$ is accepted as a valid prediction. As stated before, p-values offer significant

advantages compared to probabilities. Assume that test object has p-values of $p_z^1, p_z^2, \ldots, p_z^k$ and probabilities $r_z^1, r_z^2, \ldots, r_z^k$ of belonging to a class $l_1, l_2, \ldots, l_k$. In the case of probability, $\sum_i r_z^i$ must be sum up to 1.

For example, supposing $k = 2$: if z does not belong to either one of these classes and $r_z^1 \sim 0$ then r_z^2 will artificially tend to 1. This means that probability-based assessments might reach an incorrect solution for previously unseen samples.

2.5.3 Statistical decision assessment

The decision assessment of a new incoming point is carried out through the use of two evaluation metrics: Algorithm Confidence and Algorithm Credibility.

- Algorithm Credibility $A_{cred}(z^*)$ is defined as the p-value of the test object z^*. A high credibility means that the test object is very similar to the object in the class chosen by the classifier. High credibility values however only tell a partial story: between all classes there may be multiple ones with high credibility, meaning that multiple labels are matching. This further aspect is captured by Algorithm Confidence;

- Algorithm Confidence tells how certain the evaluated algorithm is to the choice. The highest value of confidence is reached when the algorithm credibility is the highest p-value. Low algorithm confidence indicates that the given object is similar to other classes as well.

  Concept drift 8 (Algorithm Confidence)**.**

$$A_{conf}(z^*) = 1 - max(P(z^*) \backslash A_{cred}(z^*))$$

$$P(z^*) = \{p_{z^*}^{l_i} : l_i \in L\}$$

2.5.4 Transcend Framework

Given the notions of NCM and evaluation metrics (Algorithm Credibility and Confidence), it's now possible to describe Transcend's core. Overall, Transcend uses two types of assessment to be able to evaluate the quality of an algorithm employed on a given dataset:

- Decision Assessment: Algorithm Credibility and Confidence allows to understand if the choice made by the underlying classifier is supported by statistical evidence. The best case would be to have high Credibility and Confidence, while low Credibility and high Confidence in the worst one;
- Alpha Assessment: It evaluates the quality of the NCM in terms of its ability to discriminate between correct and incorrect predictions. p-value distribution for each class is plotted, further splitting the chart between correct and incorrect predictions. If a threshold can be identified between the two, it can be used at test time to discard drifted points.

2.5.5 Practical Conformal Evaluation

Transcend uses Confidence Estimators (CE) based on NCM to spot and reject new samples. Although effective, this approach faces issues with experimental bias and is highly resource-intensive. The paper Transcending Transcend [BPPC24] aims to fill this gap by proposing some Conformal Evaluation implementations that can be used in real-world scenarios.

Transductive Conformal Evaluator (TCE)

In assessing the quality of the prediction of a new test point, there is the question of which previously encountered points the new point should be compared to. Typically, the new test point is compared against a set of calibration points. In Transcend they proposed TCE.

1. With a TCE, every training point is also used as a calibration point;
2. To generate the p-value of a calibration point, it is first removed from the set of training points and the underlying classifier trained on the remaining points;
3. Using a given NCM, its p-value is computed with respect to the points whose ground truth label matches its predicted label;

4. This procedure is repeated for every training point. Following this, Transcend's thresholding mechanism operates on the calculated p-values to determine per-class rejection thresholds;

While this solution produces very accurate results, it's extremely inefficient, as a new classifier should be trained for each training point.

Approx-TCE

In approx-TCE, calibration points are left out in batches, rather than individually. The training set is randomly partitioned into k folds of equal size. From the k folds, one is used as the target of the calibration and the remaining $k-1$ folds are used as the bag to which those points are compared to.

The approximation grows more accurate as k increases until k equals the cardinality of the training set at which point the approx-TCE and the TCE are equivalent. In this sense, the approx-TCE can be viewed as a generalization of the TCE.

Inductive CE (ICE)

The ICE splits the dataset in two non-empty partitions:

- the proper training set
- the calibration set

Unlike the TCE, p-values are not calculated for every training point, but only for examples in the calibration set, with the proper training set having no role in the calibration at all.

ICE is informationally inefficient. Only a small proportion of the training data is used to calibrate the conformal evaluator when ideally we would use all of it. The performance of the evaluator depends on the quality of the split, so the ability of the calibration set to generalize well.

Cross-CE (CCE)

The training set is partitioned into k folds of equal size. So that a p-value is obtained for every training example, each fold is treated as the calibration set in turn, with p-values calculated as with an ICE, using the union of the $k-1$ remaining folds as the proper training set to fit the underlying classifier.

When a new point arrives, the prediction from each classifier is evaluated against the corresponding calibration set. The final result is the majority vote over the k folds, i.e., the prediction of a particular class is accepted if the number of accepted classifications is greater than $k/2$, and rejected otherwise.

Note that this problem is embarrassingly parallel: parallelization and can be exploited to reduce computational time.

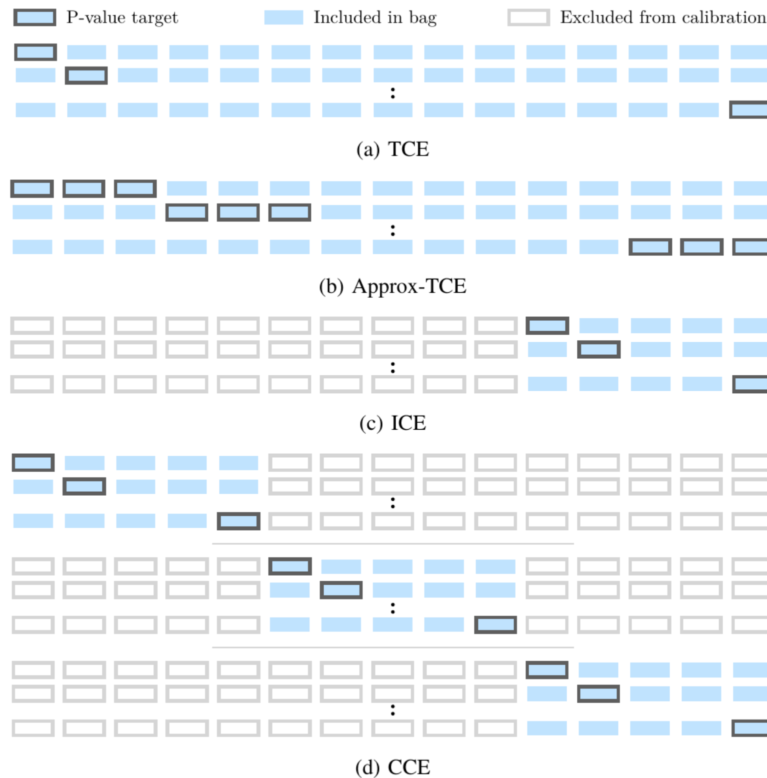


Figure 2.9: CE approaches

2.6 Related Work - Workplan, Tasks and Milestones

2.6.1 Overview

During the internship abroad for the final examination (at EURECOM), the work was performed using an incremental approach. In each cycle, corresponding to one week of work:

1. The focus is on one task of the project/research, with the goal of producing an increment;
2. After the week of work, the increment is reported and discussed with the supervisor and the research team during the scheduled week review. Then back to 1.

The project is broken down into four main phases:

- *Literature review* on malware static analysis, concept drift definition and taxonomy, concept drift detection techniques, as well as XAI tools. The primary goal is to select candidate algorithms to assess concept drift;
- *Dataset train/test split.* Preliminary phase whose goal is to build an adequate environment that supports the study of concept drift.
- *Static features extraction from malware.* Working on an already existing project, the goal is to produce a dataset that will be used in for next phases;
- *Clustering.* Using the built dataset, experiment with state-of-the-art hierarchical clustering algorithms to eventually find clusters of malware;
- *XAI for Cluster Analysis.* Use XAI techniques to study what are the main features that characterize each cluster;
- *Malware classification and concept drift detection.* Evaluate existing projects for concept drift detection and implement the missing parts. Detect an eventual concept drift with respect malware family in the testing set. Finally, collect and interpret the results.

2.6.2 Tasks

Each phase listed above is further split into one or more tasks:

- Phase 1: Literature review
  - Task 1.1: Previous research

    The goal of this task is to acquire a deep understanding of the paper "Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance" [DHA+23], which I needed to work on;
  - Task 1.2: Review on Concept drift as well as XAI

    The goal is the understanding of Concept drift definition and taxonomy, as well as the initial identification of candidate detection algorithms useful for the study.
- Phase 2: Train/Test split

  Using the previously collected executables from VirusTotal reports, define a time-based train/test split suitable for the study of malware concept drift.
  - Task 2.1: Day/Month split

    Analyze train/test splits using time windows of k days or n months.
  - Task 2.2: Timestamp split

    Compute a timestamp t, designating data before t as the training set and data at and after t as the testing set.
- Phase 3: Static features extraction from malware
  - Task 3.1: Project understanding

    Coordinate with the project members to understand the inner workings of the existing project;
  - Task 3.2: Project cleaning and re-design

    Clean and re-design the project to support code readability and maintainability;

  - Task 3.3: Optimize n-gram extraction and pipeline run

    Optimize n-grams extraction for getting the results in practical times, as well as to reduce memory footprint. Perform the full pipeline run to ultimately get the final dataset;

- Phase 4: Clustering
  - Task 4.1: Preprocessing

    Perform data preprocessing
  - Task 4.2: Dimensionality reduction

    Use state-of-the-art techniques for dimensionality reduction: both linear and non-linear methods;
  - Task 4.3: Hierarchical clustering

    Perform hierarchical clustering and evaluate cluster quality using unsupervised and supervised scores.
  - Task 4.4: Use ground truth labels for assessing the relationship of packing algorithms and malware families tags (provided by AVClass2) with clusters;
  - Task 4.5: Draw conclusions;
- Phase 5: XAI for Cluster Analysis
  - Task 5.1: Study and evaluate different XAI techniques for cluster analysis, choosing the best one for the case study;
  - Task 5.2: Implement and experiment with the selected solution, leveraging the clustering labels found from the clustering phase;
  - Task 5.3: Draw conclusions;
- Phase 6: Concept drift detection with respect of AVClass2 labels
  - Task 6.1: Choose the best method for addressing concept drift;
  - Task 6.2: Assess what are the missing parts to implement;
  - Task 6.3: Implement the Non-Conformity measure

  - Task 6.4: Concept drift detection

    Use concept drift detector grounded on Conformal Evaluation to statistically assess Concept drift

  - Task 6.5: Conclusions

    Interpret the results and conclude concept drift

- Phase 7: Draw the conclusions of the thesis and propose future works.

2.6.3 Milestones

Project milestones include the successful creation of the dataset based on the time train/test split, the identification of clusters of malware with reasonable quality results, the experimentation of XAI techniques based on clustering results and the identification of concept drift with respect to malware families.

Chapter 3

Static Features Extraction

This chapter details the project for the static features extraction process. The first section provides an introduction and explains the motivation behind the creation of a new dataset. Then, the chosen dataset train/test split is exposed and the listing and description of the extracted features are outlined. Finally, the chapter ends with a section on design and implementation. The project code is available on GitHub in the Malware Static Features Engine repository.

3.1 Introduction

This project builds directly on the work presented in the paper [DHA$^+$23], which was discussed in the previous chapter.

The primary motivation to recreate the entire dataset of malware static features relies on how the data extraction process works itself. Indeed, the pipeline passes through an initial stage where the most important features are identified and selected, using the training dataset only. Since the training/testing split will be a time-based split to further experiment concept drift, it will be different from the chosen one in the previous work.

The full pipeline is then re-run, and code optimization is achieved for practical purposes. For these reasons, part of the personal work involved gaining a deep understanding of the existing project's inner workings.

The process began with an initial code cleanup, followed by a partial redesign.

The implementation of the feature extraction process from the individual files, however, remained unchanged to ensure consistency with the previous work.

3.2 Training/Test split

The malware samples used for the static feature extraction process are the same ones selected for the original work. The set comprises 67000 malware, obtained from VirusTotal reports, each one belonging to one of 670 malware families. The malware, in their turn, was submitted to VirusTotal between 2006 and 2022. The submission date distribution can be visualized in the figure 3.4.

As malware family labels, those provided by AVClass2 [SC20] were used. AVClass2 is an automatic malware tagging tool that given the AV labels for a number of samples (VirusTotal JSON reports), extracts for each one a clean set of tags that capture properties, including the malware family.

The dataset has been previously chosen to be balanced in terms of families so that no one was underrepresented. For each family in particular there are 100 samples. More details of the chosen malware can be found in the dataset section of the background chapter.

Given the malware samples and their respective family, the training/test split is computed in order to extract the features subsequently.

3.2.1 Time-based split

On one side, the train/test split will determine indirectly what features will be extracted, but at the same time concept drift is also studied in a later phase.

This implies that the split has to be time-based. Ultimately, at this stage, a timestamp t is calculated, designating data before t as the training set and data at and after t as the testing set.

While on one side the split has to favour the majority of the samples to belong to the training dataset, a good environment to study concept drift would be to have as many new appearing families as possible, i.e. malware families never seen in the training dataset.

Given these considerations, three settings were identified, to best grasp the

study: $50\% - 50\%$, $70\% - 30\%$, and $62.33\% - 37.67\%$ Train-Test, built using an objective function.

1. **50%-50% Train-Test**

   $50\% - 50\%$ Training-Test split was considered the one with the minimum amount of training data.

   The number of appearing families in the testing set is the maximum compared to the other following splits.

   Given this minimum threshold, the time t can be incremented as long as the number of appearing families in the testing set remains the same, but does not exceed a ratio of $70\% - 30\%$.

   From the actual experiments, The training set in this setting is ultimately composed of 51.1% of data points.

   To compute the time-based split the bisection method was implemented to find the timestamp t that satisfies the train-test proportion. Moreover, we want to analyze the number of appearing malware families in the testing set.

   ```
   Split at 2021-08-26 12:40:17
   Training set length: 34240, (51.1%)
   Testing set length: 32760, (48.9%)
   Num families in training: 650
   Num families in testing: 650
   Common families: 630
   Families in training but not in testing: 20 (2.99%)
   Families in testing but not in training: 20 (2.99%)
   ```

   Figure 3.1: Report $51.1\% - 48.9\%$ Train-Test split.

2. **70%-30% Train-Test**

   In this split, a significant percentage of the training set is favoured, with less concern about the number of new appearing families in the testing set.

```
Split at 2021-12-09 08:48:58
Training set length: 46900, (70.0%)
Testing set length: 20100, (30.0%)
Num families in training: 663
Num families in testing: 606
Common families: 599
Families in training but not in testing: 64 (9.55%)
Families in testing but not in training: 7 (1.04%)
```

Figure 3.2: Report $70\% - 30\%$ Train-Test split.

3. **62.33%-37.67% Train-Test**

   This time-based Train-Test split was computed by implementing an objective function.

   The function to maximize linearly favours splits based on two criteria, that weigh the same to the function value:

   (a) The training set length in proportion approaches 70%;

   (b) The percentage of appearing families in the testing set increases;

   The identified split represents a trade-off between these two scores.

   This split has been considered the optimal one, as the size ratio is still appropriate, as well as the number of appearing families, 16(2.39%) gives some margin to study their effect in the classification context.

```
Split at 2021-09-03 13:47:49
Training set length: 41763, (62.33%)
Testing set length: 25237, (37.67%)
Num families in training: 654
Num families in testing: 632
Common families: 616
Families in training but not in testing: 38 (5.67%)
Families in testing but not in training: 16 (2.39%)
```

Figure 3.3: Report 62.33%-37.67% (optimal) Train-Test split.

In figure 3.5, two objective function visualizations are displayed, along with the distribution of the submission date of malware.

The two plots show the same information. The first one highlights the function's values at varying date split indexes: the unique submission dates are treated as equidistant values. The second one, instead, puts the submission date timestamp in the x-axis: since the submission date distribution is not uniform, the visualization gets stretched. The plots evaluate the functions in the time window defined between 50%-50% split and 70%-30% only, as they were considered the lower bound and upper bound, respectively.

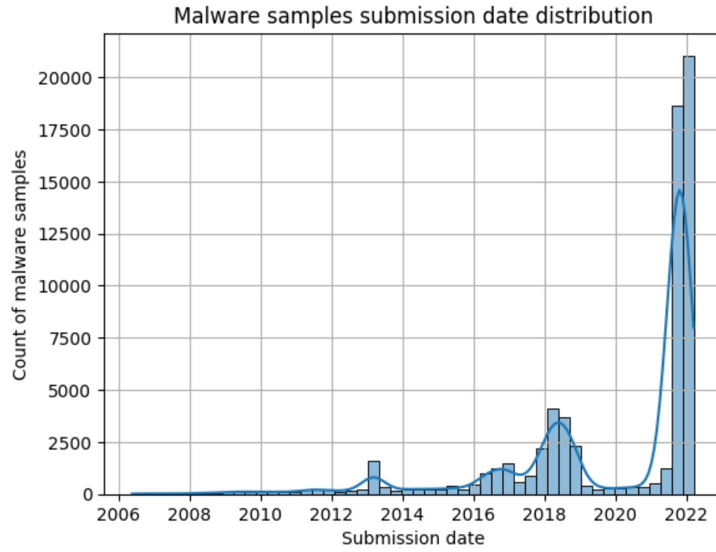


Figure 3.4: Malware submission date distribution

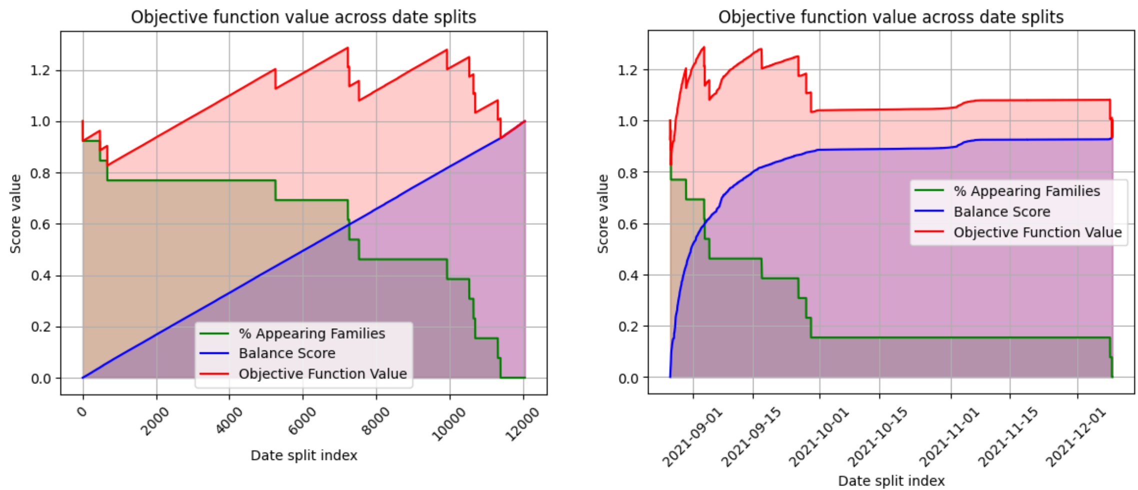


Figure 3.5: Objective function visualizations

3.3 Data collection and preparation

In this study, the data extraction process is exclusively focused on Portable Executable (PE) files. The PE format is a data structure that encapsulates the

information necessary for the Windows OS loader to manage the wrapped executable code. This includes dynamic library references for linking, API export and import tables, resource management data and thread-local storage (TLS) data (more details in the background chapter).

Due to the high number of features that can be extracted from PE files, the dataset creation has to pass between two stages:

1. First, the top static features are extracted from the malware. The top static features are computed using the malware in the training dataset and refer to those ones that are considered more significant for a subsequent classification phase.

   Based on the feature type, Information Gain or *tf-idf* functions are computed in order to estimate their importance in the classification. In the end, the best ones are selected to form the dataset features.

2. Using the One-Hot encoding approach, all the malware are then reprocessed to check the presence of some top features. With respect to the present feature value, the malware feature is set to true, false otherwise.

   The following sections describe what static features are taken into consideration. Motivation for their importance is also given by linking them to already existing evidence of their effectiveness in the classification context. Finally, explanation is given on how the best ones are filtered out.

3.3.1 Static features

PE file header [RMH21, VB18]

The main PE Header is a structure of type `IMAGE_NT_HEADERS` and mainly contains `SIGNATURE`, `IMAGE_FILE_HEADER`, and `IMAGE_OPTIONAL_HEADER`.

From `IMAGE_OPTIONAL_HEADER` some features have been extracted:

- `SizeOfHeaders`

- `AddressOfEntryPoint`

- ImageBase

- SizeOfImage

- SizeOfCode

- SizeOfInitializedData

- SizeOfUninitializedData

- BaseOfCode

- BaseOfData

- SectionAlignment

- FileAlignment

From IMAGE_FILE_HEADER:

- NumberOfSections

- SizeOfOptionalHeader

- Characteristics

Imports [VB18]

- To list the imported DLLs by each malware, it's necessary to iterate the data directory DIRECTORY_ENTRY_IMPORT. For each DLL is then possible to list its imported APIs;
- Once extracted, APIs and DLLs are filtered by excluding the ones that appear less than 0.1% or more than 99.9% of the overall set, respectively;
- For the top imports, DLLs aren't filtered while top 4500 APIs are selected based on the Information Gain.

Generics

Generic features comprise:

- The file size in bytes;
- The Shannon Entropy of byte value frequencies of the file.

n-grams [RZC+16]

n-grams of byte sequences consist of sequences of n bytes from a given sample of binary files.

- 4-grams and 6-grams are extracted, representing 4 and 6 consecutive bytes respectively. To get them, a sliding window is set up that iteratively shifts of one byte and in the end, the set is built to get the unique values;
- For practical issues, they are computed from 1000 of randomly chosen malware samples;
- The most relevant n-grams are computed by filtering the ones that appears less than 1% or more than 99% of the overall 1000 samples;
- Top 13000 n-grams are chosen, based on their Information Gain.

Opcodes sequences [KYMS16]

Operation code sequences are a set of consecutive assembly-level operations.

- Opcodes of 1, 2, 3 consecutive operations are extracted;
- To compute the relevance of each opcode n-gram *tf-idf* (term frequency-inverse document frequency) function is used:

  tf-idf is a type of information retrieval function that proportionally increases as the term appears in the dataset, but inversely proportionally increases by its frequency.

  The main idea of this function is to prioritize terms that appear in the dataset, but that are less frequent;
- As done for the imports (APIs and DLLs), opcodes that appear less than 0.1% or more than 99.9% of the overall training set length are filtered;
- The top 2500 opcodes are selected, based on the Information Gain metric.

Strings [SMdlR+14]

Printable strings refer to sequences of ASCII characters of an executable that are human-readable. These strings can often include text such as error messages, log entries, function names, or other meaningful content embedded within the binary code.

In the extraction process:

- Only strings of more than 3 characters are considered;
- Once getting all the strings, the top ones are selected from the top 0.01% in terms of frequency.

3.4 Design and implementation

The application's design is rather minimal. It consists of an interface and relative implementations for each extraction phase listed in the previous section.

- `TopFeatureExtractor` interface exposes a method to compute the top features and whose side effect is in the file system: `top()` implementation should save a file with the correspondent top features;
- `StaticFeatureExtractor`, whose implementation should contain the code for re-scanning all the malware samples for the top features, and for applying One hot encoding to the categorical variables.

The project has been implemented in Python3, mostly because of the versatility of the `pefile` library, which allows to extract PE file information easily. Multiprocessing has been also exploited to parallelize the tasks, using the `Pool` object from the `multiprocessing` module. `Pool`'s `map` is the parallel equivalent of the builtin `map` function: tasks are scheduled to the specified number of processes and their return values are saved. The function blocks until all tasks succeed.

After applying One Hot Encoding to the categorical variables, the dataset comprises 46351 features.

In the following figure 3.8 is shown the activity UML diagram about the static feature extraction process.

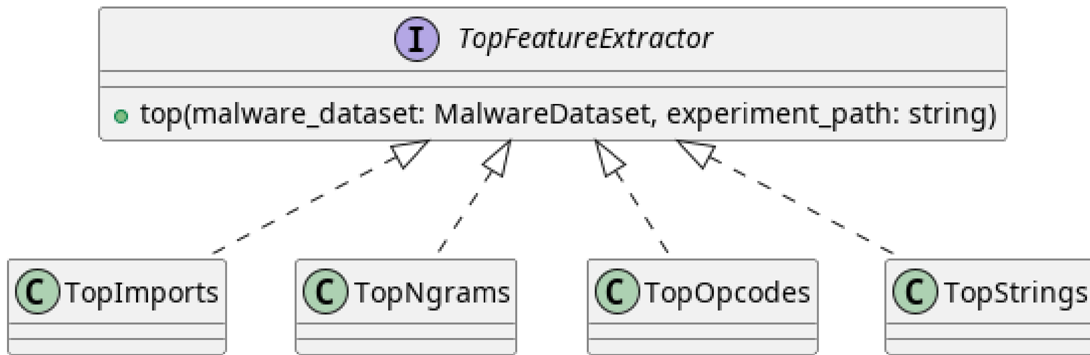


Figure 3.6: Top Feature Extractor UML

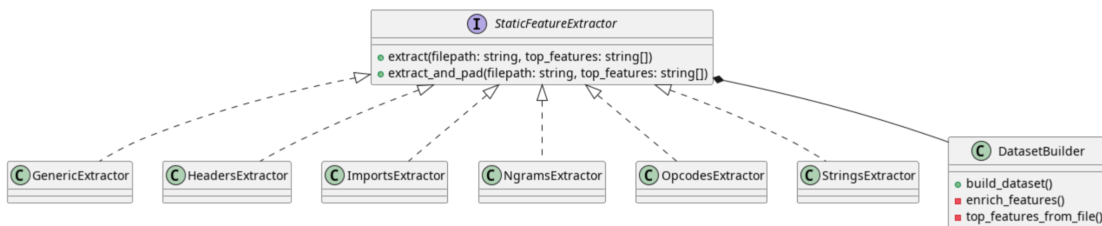


Figure 3.7: Static Feature Extractor UML

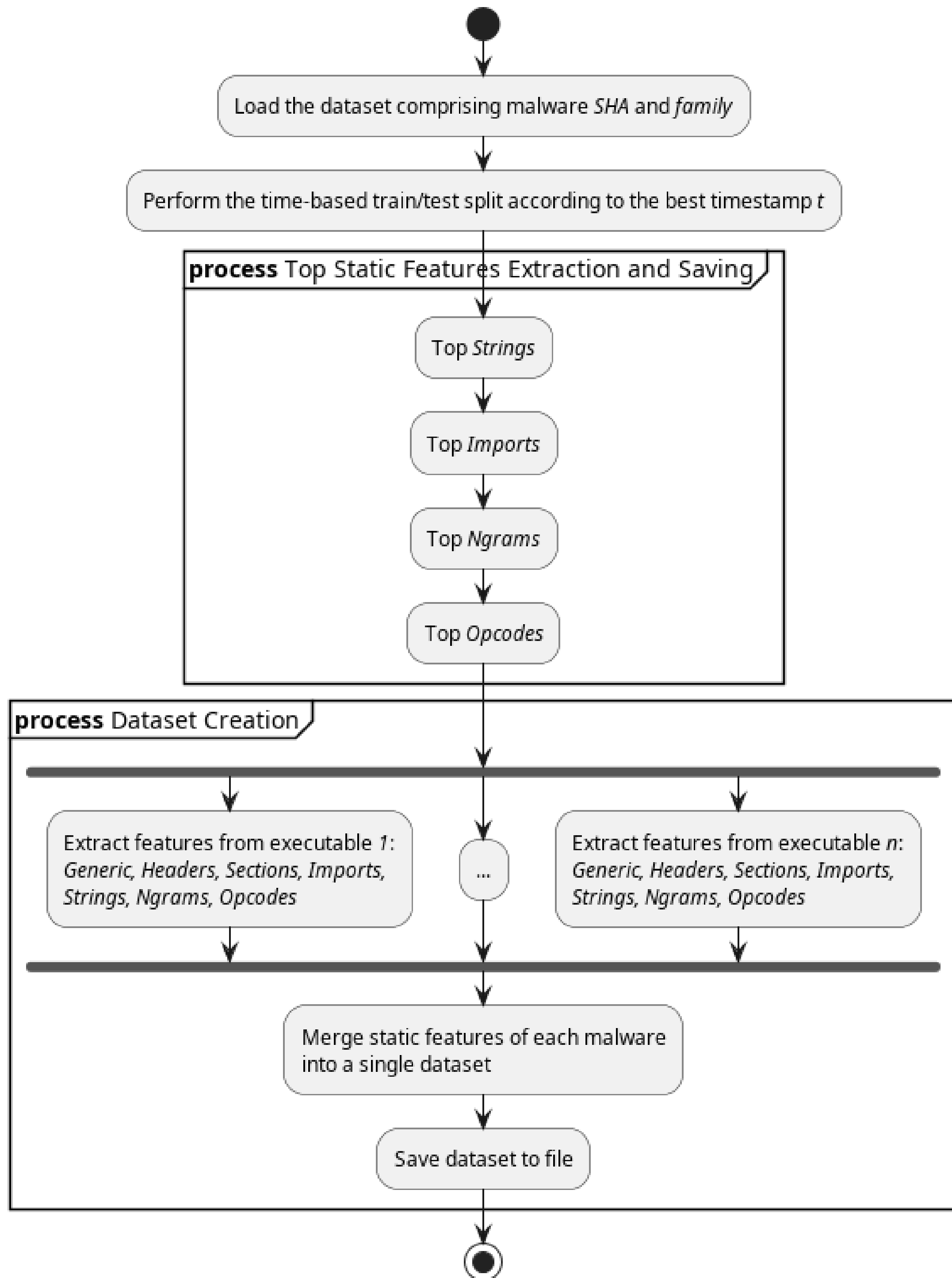


Figure 3.8: Static Feature Extraction Process UML

Chapter 4

Clustering Windows Malware

4.1 Introduction

This chapter builds upon the feature extraction project from malware highlighted in the previous chapter and the successful execution of the corresponding pipeline. It focuses on the clustering phase, where experiments are conducted to explore and evaluate different hierarchical clustering algorithms. The primary goal is to determine whether underlying relationships exist among various malware samples based on shared features.

By creating a hierarchical structure, the algorithms allow for the identification of clusters of malware at different levels of granularity, allowing a deeper understanding of how these samples relate to one another across different scales.

After assessing clustering quality at different dendrogram levels, the optimal cut is selected for further analysis.

The chapter also focuses on the relationship between the defined clusters and current labeling systems such as AVClass2 for family labels and off-the-shelf packers and protectors by using the signature-based Detect It Easy (DIE) tool, and the Yara rules of Avast RetDec. Both labels were previously extracted for the paper "Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance" [DHA+23].

All the clustering-related code is developed using GitHub and can be found in the Clustering Windows Malware repository.

4.2 Pipeline

This section elucidates the overall pipeline involved in the clustering phase. From an initial preprocessing stage, different dimensionality reduction techniques are applied and evaluated, as well as different hierarchical clustering algorithms. Finally, results are outlined leveraging common unsupervised and supervised scores.

4.2.1 Preprocessing

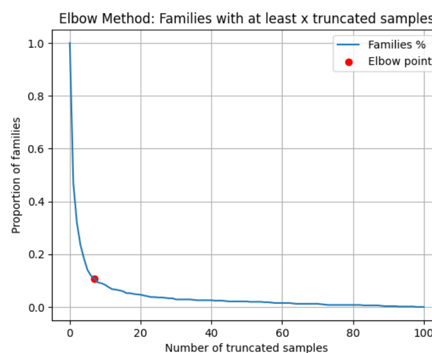
Data cleaning

The first preprocessing stage involves data cleaning. Given the initial dataset of shape $(67000, 44731)$, instances belonging to **truncated files** are deleted: these samples are corrupted and may have invalid values on a subset of features. More precisely, all samples belonging to a family are deleted if at least 7% of them are truncated. This number is chosen by applying the elbow method, in such a way that not an excessive amount of families are discarded, as well as still providing a valid train/test split in terms of the ratio of samples. Decreasing it drastically would result in too many deleted families. On the other hand, a 'high' percentage would result in data with too much noise.

After the data cleaning stage, the dataset is cleaned from 72 families, which results in a final shape of $(59800, 44731)$.

```
Split at 2021-09-03 13:47:49
Training set length: 37428, (62.59%)
Testing set length: 22372, (37.41%)
Num families in training: 572
Num families in testing: 560
Common families: 534
Families in training but not in testing: 38 (6.35%)
Families in testing but not in training: 26 (4.35%)
```

(a) Dataset report after data cleaning



(b) Truncated sample filtering

One Hot Encoding

One Hot Encoding is carried out in the static feature extraction pipeline, but only on the features belonging to the top ones. Another round of One Hot Encoding is applied to the remaining categorical features.

Only the categorical variable `pesectionProcessed_entrypointSection_name` is left, comprising 1622 different values. The final dataset shape after this stage and by setting SHA256 as the index is $(59800, 46351)$.

Feature selection

Feature selection is applied as well in order to remove zero variance features. Indeed, they don't provide any information for ML models to predict a target variable, as well as for clustering.

A total of 255 features are deleted.

Data normalization

Finally, the last preprocessing phase involves data normalization. Min-Max normalization is applied in order to scale all the features to the range $[0, 1]$.

4.2.2 Dimensionality Reduction

Due to the high dimensional nature of the feature space built in the previous stages, there are practical issues concerning the performance of both clustering and supervised ML algorithms. Dimensionality reduction techniques are needed for faster computations. Indeed, they allow for the reduction of computational load, improving their efficiency and speed. Furthermore, they help to reduce the amount of noise present in the data by extracting relevant information.

Different dimensionality reduction techniques are applied during the experiments, and their effect on clustering is subsequently assessed by computing quality scores. All these implementations are provided by the scikit-learn [PVG+11] library. The reduced feature space has been chosen to comprise 1000 columns.

- `MiniBatchSparsePCA`. It is a faster, but less accurate variant of Sparse PCA, a dimensionality reduction algorithm based on Principal Components Anal-

ysis. It is a linear reduction algorithm characterized by sparse components i.e. components that can have zero coefficients, and it is beneficial for explainability purposes;

- `SpectralEmbedding`. It's a non-linear dimensionality reduction technique based on spectral graph theory. It builds the transformation by finding the eigenvectors and corresponding eigenvalues of the Laplacian matrix;
- **Autoencoder**. The autoencoder is a type of neural network architecture whose goal is to learn a latent representation of the input data. It is composed of an encoder and decoder part: while the first one constructs a compressed representation of the input (capturing the main patterns in the data), the latter in turn serves to reconstruct it back to the original dimension. The learning procedure guarantees a minimal reconstruction loss, thus forcing the latent space to contain the core information;
- `UMAP` (Uniform Manifold Approximation and Projection). It is a non-linear dimensionality reduction algorithm that can directly run on sparse matrices. Due to the sparsity of the input dataset, caused by the one hot encoding procedure (from the others), this method has been also considered a good candidate algorithm.

4.2.3 Hierarchical clustering

Two hierarchical clustering algorithms are experimented over the dimensionality-reduced feature space: agglomerative clustering and HDBSCAN. Scores are analyzed to finally choose the best method.

- `AgglomerativeClustering`. Using a bottom-up approach, it starts by considering single points as different clusters. The clusters are then merged based on the linkage criteria: ward, average, complete, and single are examples of linkage that are commonly used.

  Ward distance has been considered more appropriate for our problem, as it generates more balanced clusters in terms of the number of samples. scikit-

learn and scipy [VGO+20] libraries also help with the creation of the dendrogram tree and extraction of the cluster labels at different heights.

- `HDBSCAN`. It is an extended version of DBSCAN and is particularly useful when clusters have different densities. It automatically finds the best number of clusters, without the need to specify it beforehand to the user.

  In this case, HDBSCAN python implementation provided by the hdbscan is used. The library allows us to get the cluster labels at the different levels of the generated dendrogram, the Single linkage tree. Experiments showed that the generated clusters are very unbalanced: data is, most of the time, merged into the same cluster. The result is the presence of a big cluster and very small ones with few samples (≈ 1 sample per cluster).

4.3 Results

4.3.1 Clustering quality assessment

Unsupervised scores

The following two figures present clustering quality scores obtained using the agglomerative clustering algorithm. In the first figure, `MiniBatchSparsePCA` implementation is used for the dimensionality reduction, while in the second `UMAP`. Only those two techniques are assessed in this section, as spectral embedding and autoencoder show very similar tendencies as the PCA variant. Both figures show plots on three unsupervised scores, computed by cutting the dendrogram at different levels by extracting the cluster labels:

- **Davies-Bouldin score**. The minimum score is zero, with lower values indicating better clustering;

- **Silhouette Score**. Highest average Silhouette score means better-defined clusters;

- **Calinski-Harabasz index**, also known as variance ratio criterion. A higher score relates to a model with better-defined clusters.

These three unsupervised scores are commonly used in the literature for hierarchical clustering to assess cluster quality without the need for ground truth labels. Since the Silhouette Score complexity is $O(n^2)$, applying it to large datasets becomes impractical. Thus, the plots show the scores at 50 clusters offset between two samples, allowing to restrict the analysis to core points. More precisely, cluster labels range from 2 clusters to 3702, corresponding to an upper bound of 10 samples per cluster. Next to the Silhouette score, the `DB` score and `CH` score are more lightweight in terms of complexity, so they are considered practical for assessing clustering quality.

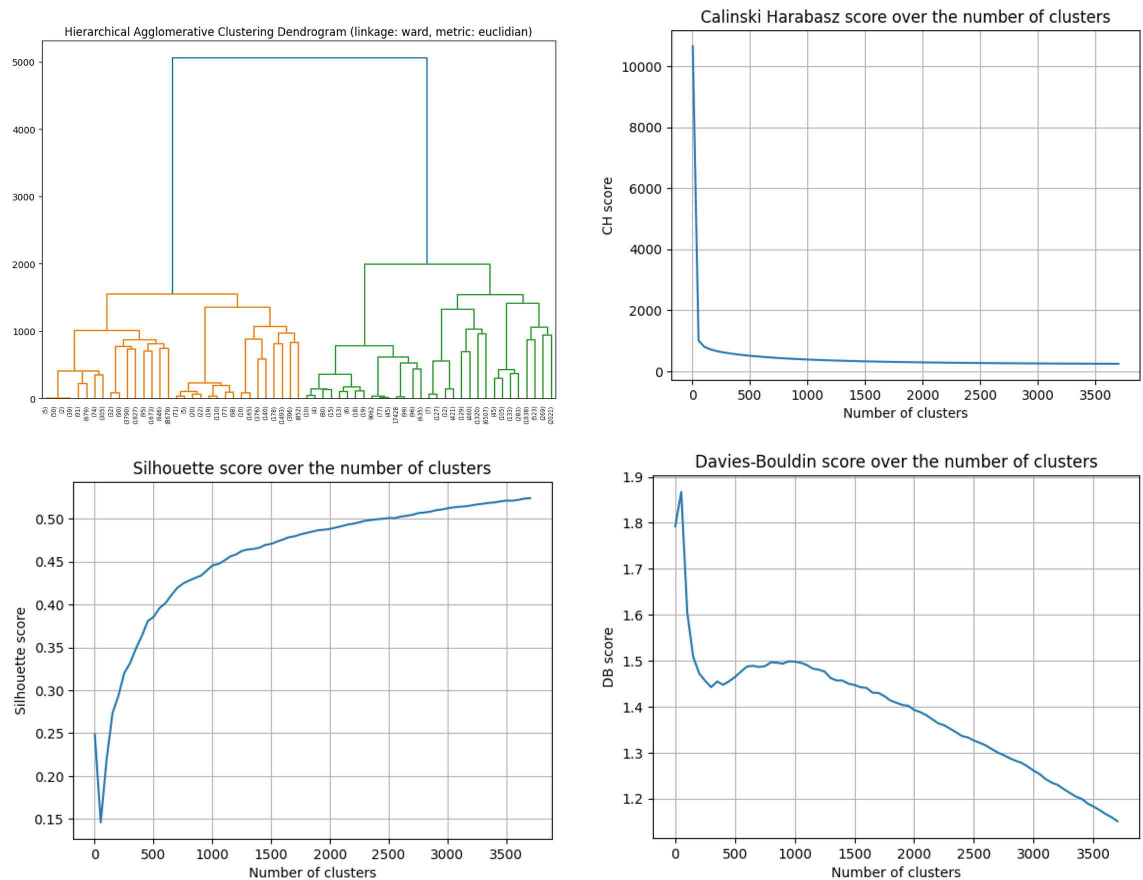


Figure 4.2: `AgglomerativeClustering` over `MiniBatchSparsePCA` dimensionality reduction

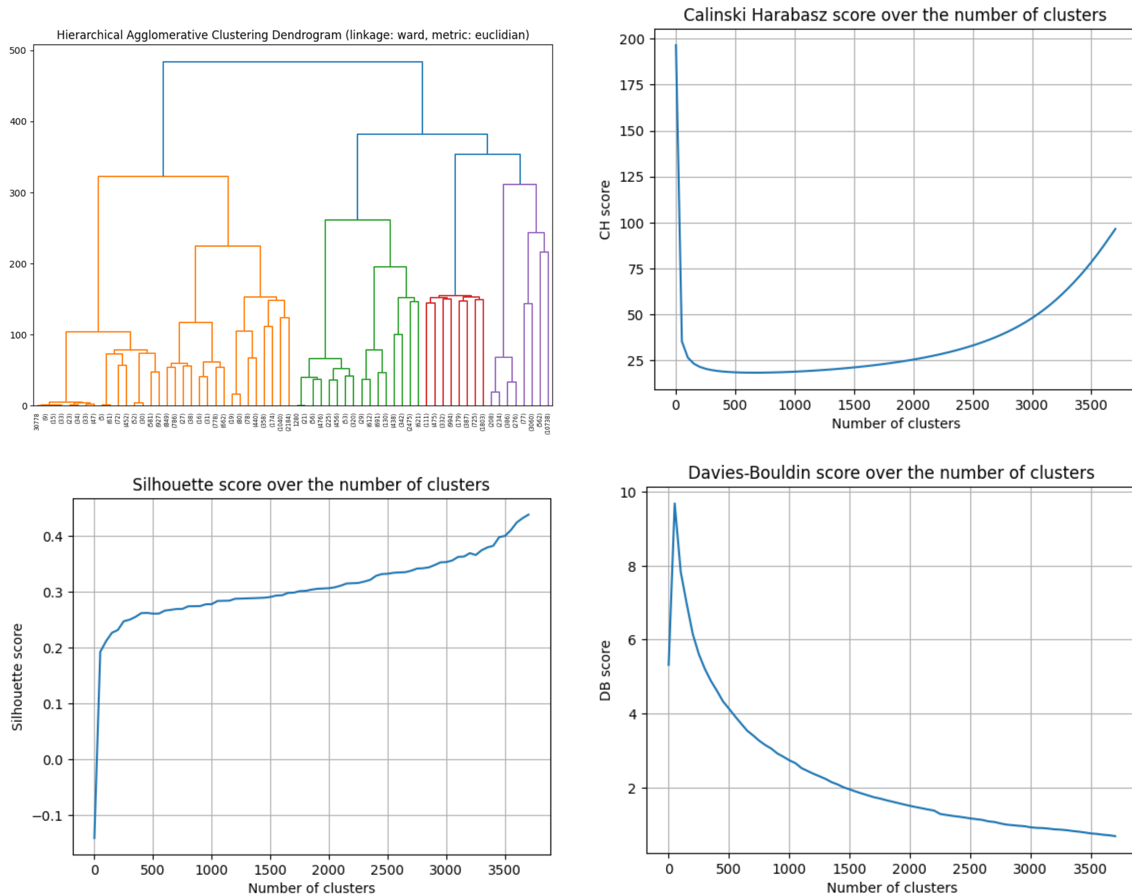


Figure 4.3: `AgglomerativeClustering` over `UMAP` dimensionality reduction

- **Agglomerative Clustering over MiniBatch Sparse PCA**. From now on, we will refer to this version as `AgglMbsPCA`. Unsupervised clustering scores (figure 4.2) show conflicting results. While Silhouette score and `DB` scores decrease by raising the number of clusters, `CH`, after the initial levels of the dendrogram tree, drastically decreases and remains stable afterwards. Silhouette and `DB` indicate increasing cluster quality over the number of clusters. Silhouette score, in particular, exceeds 0.5 after some point, which generally indicates reasonable cluster definition.

  The dendrogram tree plot gives a visual glimpse of how the hierarchical clusters are distanced from each other. Longer vertical lines mean more distanced clusters: two big clusters can be identified at the highest level,

while deeper levels of the dendrogram show less separated clusters. Moreover, at level 5 the resulting clusters are visually balanced;

- **Agglomerative Clustering over UMAP**. Going forward, we will refer to this version as `AgglUMAP`. Unsupervised scores (figure 4.3) show similar tendencies as the first approach, with the difference of `CH` score that increments in the latest levels of the dendrogram. Further analysis shows that this result is due to the unbalanced nature of clusters: the same big cluster is being split, generating clusters of single points. The resulting clusters are either relatively small (≈ 1 point) or relatively big. In figure 4.4, the Cumulative Distribution function of the number of samples per cluster is shown, as well as the box plot of the distribution. Both show the unbalanced nature of clusters, focusing on the dendrogram cut comprising 1500 clusters.

  For this reason, `AgglUMAP` has not been considered appropriate for further analysis.

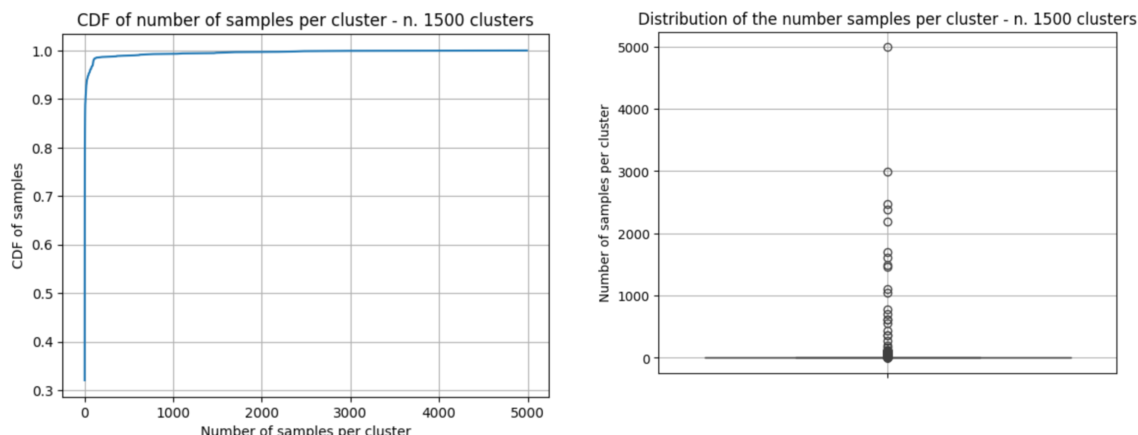


Figure 4.4: `AgglUMAP` quality assessment - Unbalanced clusters in terms of number of samples

Supervised scores - Cluster relationship with malware families

As ground truth labels, malware family labels provided by AVClass2 [SC20] were used.

Since its classification process is probabilistic and relies on heuristics, the resulting labels may contain uncertainties and inconsistencies. Treating these labels as absolute ground truth can introduce biases in evaluating clustering performance. Furthermore, family labels may be also aliases of one another: in a perfect situation a single cluster would appear with all these families grouped together. Such aliases however are not available in the literature, so this analysis has been excluded.

In this work, AVClass2 labels are used for comparison purposes while acknowledging their probabilistic nature and potential limitations in accurately reflecting true malware family assignments.

Three supervised scores are leveraged: Normalized Mutual Info (`NMI`), Adjusted Mutual Info (`AMI`), and Adjusted random score (`ARI`). The first two range between 0 and 1, while `ARI` is between -0.5 and 1. In all the cases, higher values represent increasing correlation. `ARI` has the lower bound of -0.5, and assumes negative values to indicate especially discordant clustering.

The following figures display the supervised scores focusing on the first clustering version (`AgglMbsPCA`).

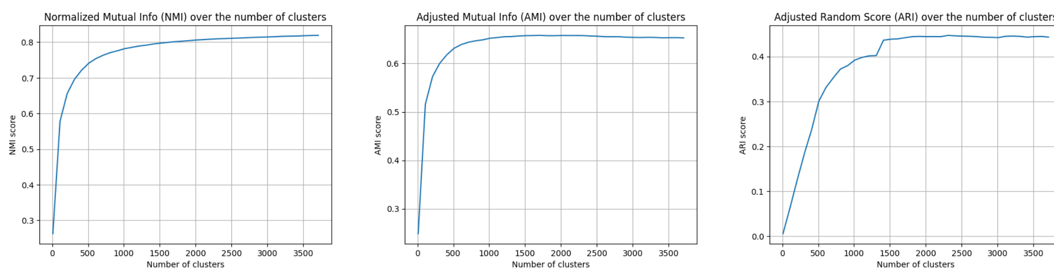


Figure 4.5: `AgglMbsPCA` quality assessment - Supervised scores tendency over the number of clusters

The supervised scores show that there is some relationship between the generated clusters and AVClass2 labels. All of them remain constant afterward a certain height of the dendrogram tree.

4.3.2 Optimal number of clusters

The optimal number of clusters in `AgglMbsPCA` is chosen to achieve good performance on both unsupervised and supervised scores. In unsupervised scores, good

Silhouette and DB are preferred over CH. Also, Silhouette score $\gtrapprox 0.5$ is suggested. An initial 2500 cluster may be appropriate in the first instance, but supervised scores show that there is no improvement starting from 1500 clusters.

This latter number is still appropriate as the Silhouette score and DB score don't decrease significantly, so it has been chosen to be the optimal number of clusters. Thus, future analysis that will be conducted on this thesis, regarding cluster relationship with ground truth labels and explainability, will focus on this dendrogram level.

The final scores at 1500 clusters have the following values:

Score	**Value**
Silhouette	0.48
DB	1.45
CH	325.73
NMI	0.78
AMI	0.66
ARI	0.44

Table 4.1: Optimal number of clusters - AgglMbsPCA scores

4.3.3 Other ground "truth" analysis

Given the AVClass2 labels and packing information about the malware, other experiments are undertaken to study their relationship with the generated clusters.

Following the same methodology, labels are compared to clusters along the dendrogram, and the best overlapping score is outputted. This analysis may reveal clusters of samples packed with the same algorithm, or clusters of samples belonging to the same malware family.

More specifically, referring to the packing analysis (equivalent to the family one):

1. The dendrogram is cut at multiple levels: from a lower bound of 10 clusters to an upper bound that equivalently corresponds to an average of 10 samples per cluster, using an offset of 100 clusters (levels);

2. For each level:

    (a) One cluster at a time is selected, considering the samples in it as `true` label, the others as `false`;

    (b) One packing algorithm is picked and all the samples packed with it are set as `true` label, `false` otherwise;

    (c) The Jaccard index is applied to investigate the relationship between a specific cluster and packing algorithm. The Jaccard index measures the similarity between two sets A and B: $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$;

    (d) For each packing algorithm, save the cluster with the highest Jaccard index;

3. For each packing algorithm, output the cluster along the dendrogram with the highest Jaccard index.

Packing algorithm - Cluster relationship

Packing executables refers to the process of compressing an executable file in order to reduce its size. While packing can be used for legitimate purposes, such as software protection and anti-tampering, it is also used for malicious purposes, as attackers use it to obfuscate the content of the malware, evading detection by security tools and hindering reverse engineering.

As stated in the introduction, information about packed malware collected by [DHA+23] is used. Packed malware detected in the training dataset comprises 23.78% of the samples, each one packed to one of 155 distinct algorithms.

Overall, 12 (8.0%) packing algorithms are associated with a cluster at some point along the dendrogram ($J \geq 0.8$).

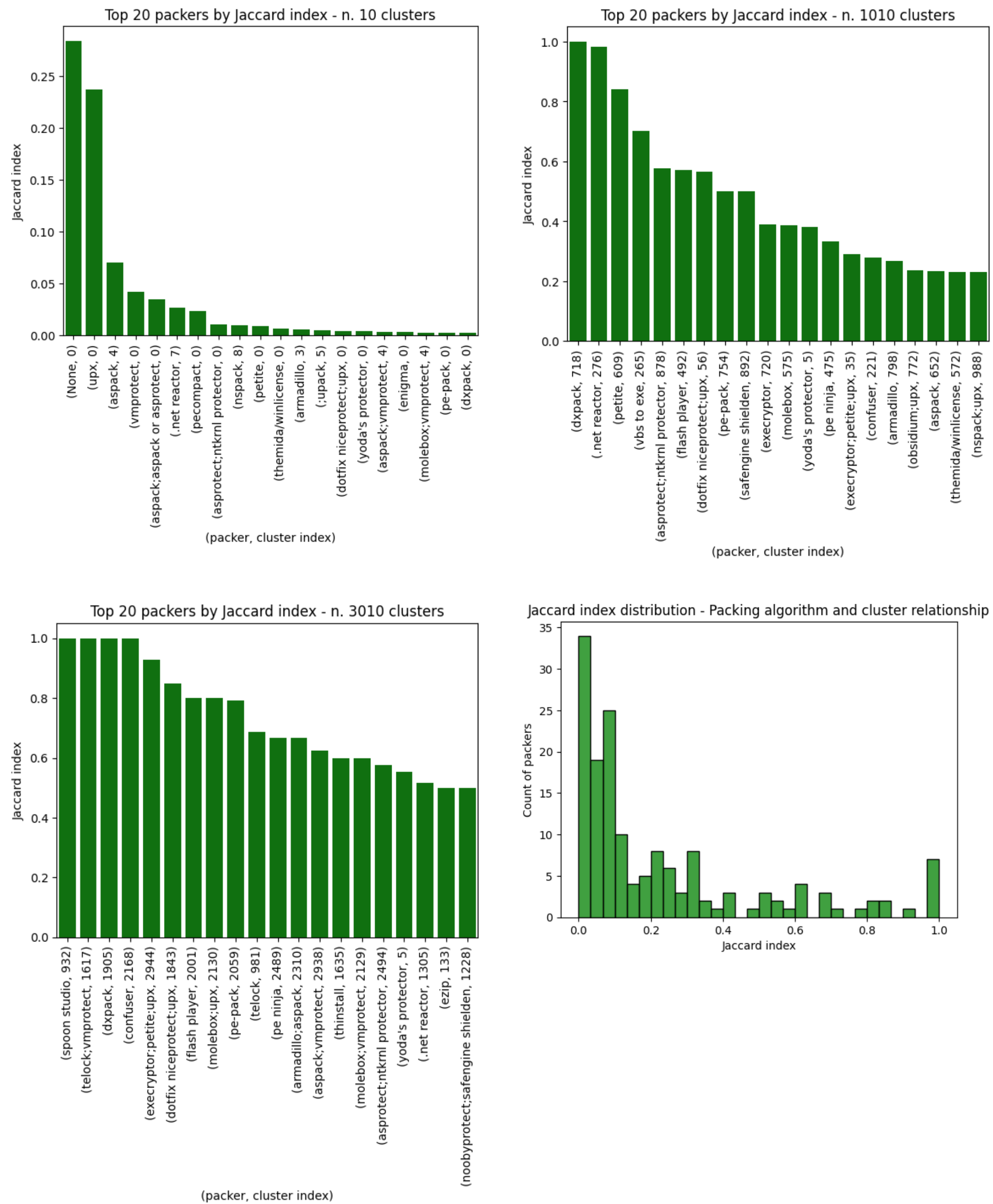


Figure 4.6: Packing analysis: Top 20 clusters by Jaccard index for increasing dendrogram levels, Jaccard index distribution along packing algorithms.

Packing Algorithm	Jaccard Index J	Cluster Index	Dendrogram Level	# Training Set Samples
`petite`	0.84	396	610	157
`ezip`	1.00	138	3310	2
`.net reactor`	0.98	101	210	62
`spoon studio`	1.00	871	2610	3
`telock;vmprotect`	1.00	1361	2410	8
`dotfix` `niceprotect;upx`	0.85	837	1210	73
`dxpack`	1.00	718	1010	43
`flash player`	0.80	1223	1710	4
`molebox;upx`	0.80	1141	1510	10
`confuser`	1.00	1153	1510	5
`eziriz .net reactor`	1.00	2609	3710	1
`execryptor;petite;upx`	0.93	2259	2310	13

Table 4.2: Packers with high relationship - $J \geq 0.8$.

Malware family - Cluster relationship

Malware family analysis follows the same pipeline as the packing one.

Compared to the packing algorithms, malware families overlap with clusters starting from the initial levels of the dendrogram, as empirically shown in the following images (figure 4.7).

Results show that 129 (23.0%) malware families (AVClass2 labels) belonging to the training dataset are associated with a cluster at some point along the dendrogram ($J \geq 0.8$).

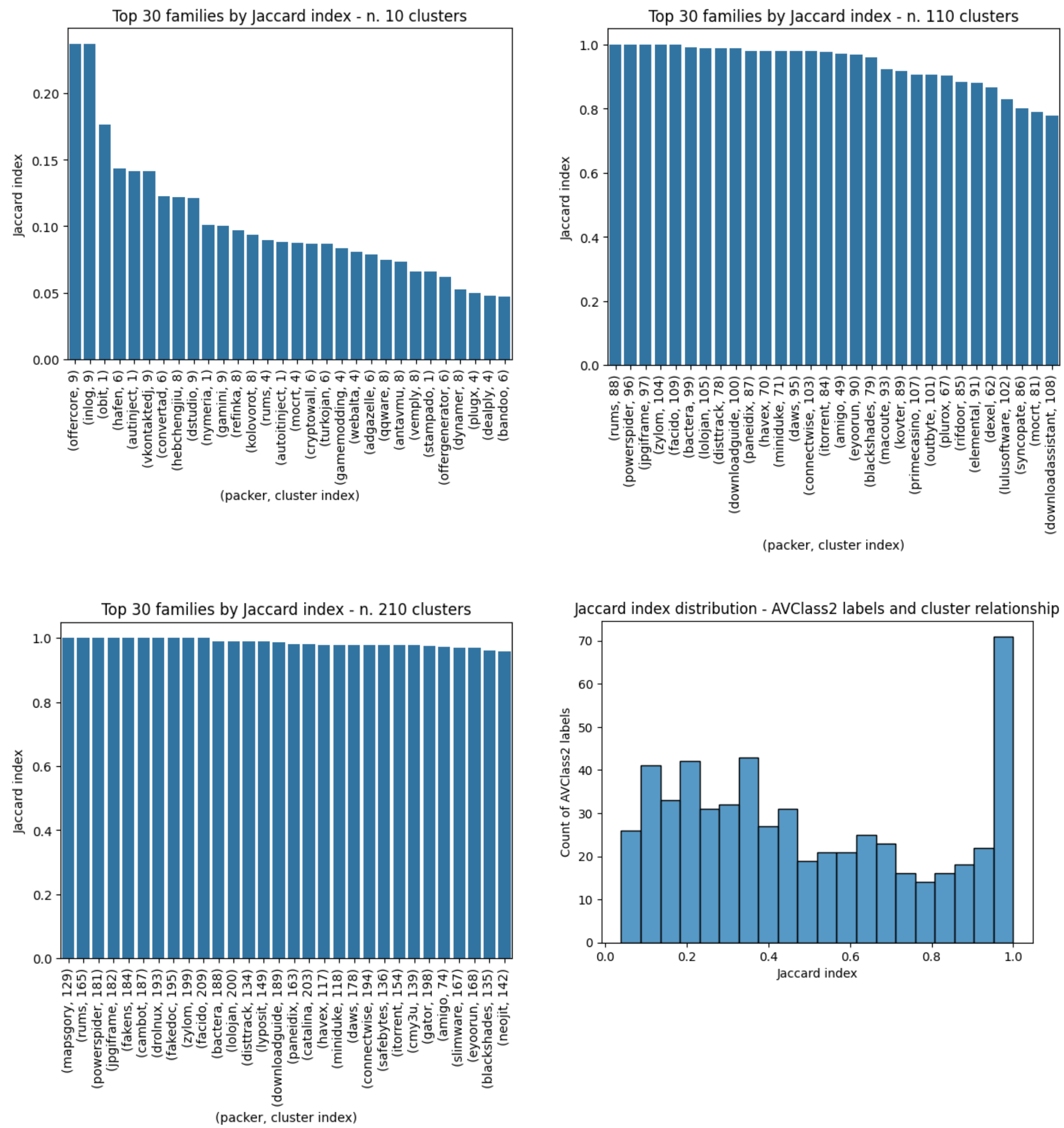


Figure 4.7: Malware family analysis: Top 30 clusters by Jaccard index for increasing dendrogram levels, Jaccard index distribution along malware families.

Chapter 5

XAI for Cluster Analysis

5.1 Introduction

Clustering is a valuable technique to study the relationship between data samples, due to its ability to group samples together based on some shared features. While intrinsically interpretable, clustering research before the advent of eXplainable AI was focused on improving performance metrics, such as scalability and accuracy, leaving interpretability out of scope.

The research in XAI has begun in recent years mainly in the classification field, while in cluster analysis just in the last few years. XAI in cluster analysis is currently divided into two philosophies, one focusing on the creation of **clustering algorithms interpretable by design**, while the other involves a **two-step approach** [Can20].

As the name suggests, the first approach focuses on creating new clustering algorithms specifically for explainability purposes. In contrast, the second approach consists of the usage of two models: one for clustering and another for classification. By training a classifier on the generated clustering labels, it is possible to apply supervised XAI techniques to generate explanations.

While a survey and a comparison of these approaches may be interesting for research purposes, it is not the scope of this thesis to investigate and experiment with all the state-of-the-art solutions.

In the context of Windows malware (under the adoption of static features),

understanding why executables are clustered together plays an important role though, as it may reveal a set of features that correspond to an attack technique or evasion strategy. Without explainability, clusters may contain arbitrary groupings that lack practical significance.

Given the optimal number of clusters computed in the previous chapter (n. 1500 clusters), this one elucidates the explainable clustering pipeline that has been implemented, giving the motivations behind its adoption concerning two different two-step solutions.

5.2 Two *two-step* approaches

Two different two-step approaches were taken into consideration for assessing the features that characterize each generated cluster.

The first approach is a *model-dependent* method that leverages *global explanations*, i.e., explanations that offer a comprehensive understanding of how the model makes decisions across all data points. In contrast, the second approach is *model-agnostic* and relies on *local explanations*, which provide insights into individual predictions.

5.2.1 Random Forest method based on Gini importance

The first solution is based on Random Forest classification models.

Random Forest is a type of ML classifier that is interpretable by default, as each Decision Tree composing the Random Forest follows a simple, rule-based structure. There, decisions are made by splitting features at various thresholds. This intrinsic structure allows for straightforward interpretation by extracting the feature-level contribution for the splitting process.

Feature importances provide global insights about their overall contribution. In scikit-learn Random Forest implementation, they can be easily extracted. As stated in the documentation, the approach is impurity-based, and it is also referred to as *Gini Importance* (Mean Decrease in Impurity (*MDI*)). The Gini importance measures each feature's importance as the sum over the number of splits (across all trees) that include the feature, proportionally to the number of samples it splits.

The overall pipeline consists of training a separate Random Forest model for each revealed cluster. More specifically, each Random Forest is trained to a new set of labels, where samples belonging to the cluster under consideration are set as `true` labels, `false` label otherwise. Finally, by computing the feature importances, one can extract the features that characterize that specific cluster.

While computing the feature importances is relatively fast in Random Forest models, this method doesn't scale well. Indeed, it has been considered impractical for this case study, as 1500 different classifiers should be trained. Thus, it is excluded from experiments.

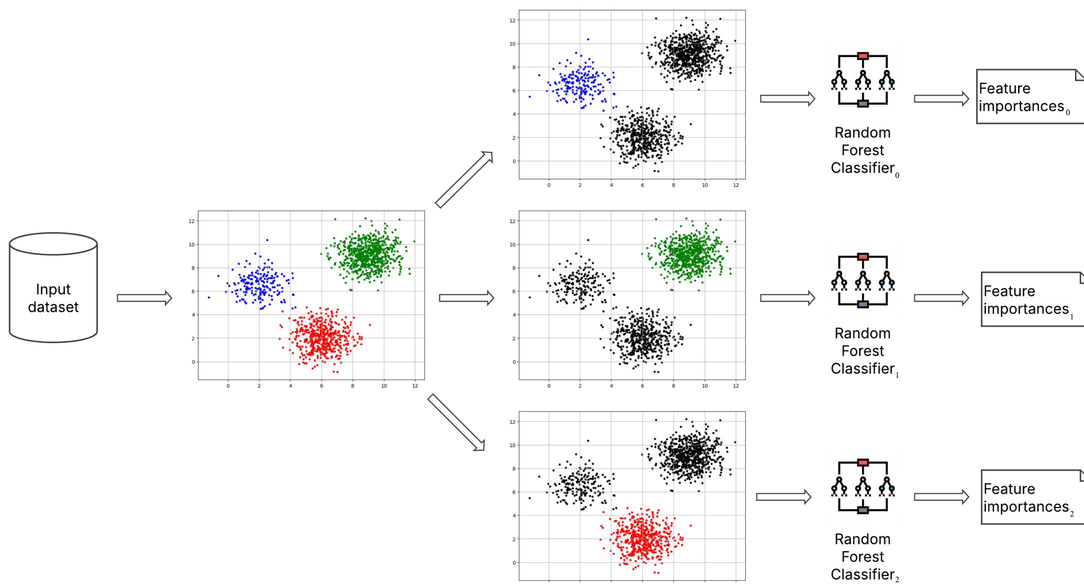


Figure 5.1: Random Forest-based two-step method based on *Gini importance*.

5.2.2 Local Explanations method

The second solution, unlike the first one, is model-agnostic and based on local explanations.

A single ML classifier is trained on the produced clustering labels, alongside a *local explainer*, such as **LIME** or **SHAP**.

For practical purposes, a percentage of instances is randomly sampled from each cluster, and their local explanations are produced. Providing local explanations

for a subset of the points is a more practical way of extracting cluster explanations, but it is informally inefficient: the computed explanations depend on the quality of the chosen points. While approximation is introduced, this method may be faster for large training sets or serve as a useful way to gain initial insights into feature contributions, with full computation reserved for later analysis.

Once local explanations are generated for every instance in the subset, we can average the feature contributions per cluster.

This method has been chosen for experimentation purposes, as it provides a more practical approach for extracting explanations with a large number of clusters (n. 1500).

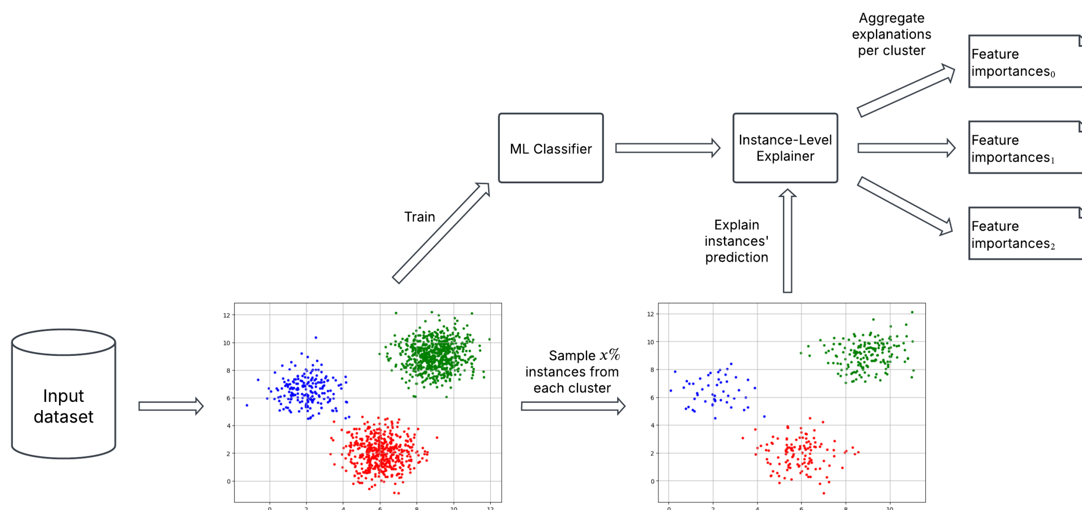


Figure 5.2: Two-step method based on local explanations.

5.3 Experiments

As stated in the previous section, experiments are conducted using the two-step method based on local explanations. The trained ML classifier is a Random Forest, as it can perfectly discriminate the clusters i.e. it has 100% accuracy over the training set. For the local explainer, *LIME* is exploited, as it offers a more lightweight method for extracting explanations compared to *SHAP*.

A `LimeTabularExplainer` object was created and used in the first instance

to compute the local explanations of 25% of points for each cluster. After an initial analysis, the explainer has been utilized to produce explanations of every point in the training set (comprising 37428 instances). We can get them by calling `explain_instance` method.

`explain_instance` takes as argument the sample whose prediction we want to explain, the classifier prediction probability function, the maximum number of features present in the explanation (that should be equal to the number of features), the size of the neighborhood to learn the linear model (`num_samples`) and the top labels, indicating the top k labels with highest prediction probabilities we want to produce explanations (in this case set to 1 as we want the one with the highest prediction probability).

Still, here `num_samples` is set to 1000, with a default value of 5000, so approximation increases.

In Listing 1 is briefly shown the implemented code to extract sample explanations. Also here, a `Pool` of processes are created to enable multiprocessing. Each sample explanation is saved to a file for saving RAM space, that would rather be occupied by the join operation.

The features that characterize each cluster are then computed by averaging the feature importances for the samples within it. In figure 5.5 the top 50 features by contribution are shown for clusters of labels 0, 250, 1000, 1250. Figures 5.3 and 5.4 display information for clusters of labels 0, 250, 500, 750, 1000 and 1250. The first one shows a box plot of the feature contribution grouped per cluster and type, and the second is the Cumulative distribution function (CDF) of the feature contribution averaged per cluster.

Finally, the last two plots in 5.6 summarize feature importance distribution grouped per type, but for all training instances. These plots indicate the features that mostly weigh for characterizing the clusters at the optimal dendrogram cut (n. 1500 clusters). The plot 5.6a displays the sum of the feature importances per type, while 5.6b is the mean.

```python
from multiprocessing import Pool
import pickle
import lime

def get_explanations(idx):
    expl_map = explainer.explain_instance(
        X_train[idx],
        predict_fn=clf.predict_proba,
        num_features=X_train.shape[1],
        num_samples=1000,
        top_labels=1,
        ).as_map()

    # Save explanations to file instead of returning it to save ram
    with open(f'models/expl_{idx}.pkl', 'wb') as f:
        pickle.dump(expl_map, f)

explainer = lime.lime_tabular.LimeTabularExplainer(
                X_train.values,
                feature_names=list(X_train.columns),
                class_names=clf.classes_,
                random_state=42,
                mode="classification"
            )

n_proc = 16
with Pool(n_proc) as p:
    p.map(get_explanations, list(range(X_train.shape[0])))
```

Listing 1: Explanations extraction using `LimeTabularExplainer` implementation.

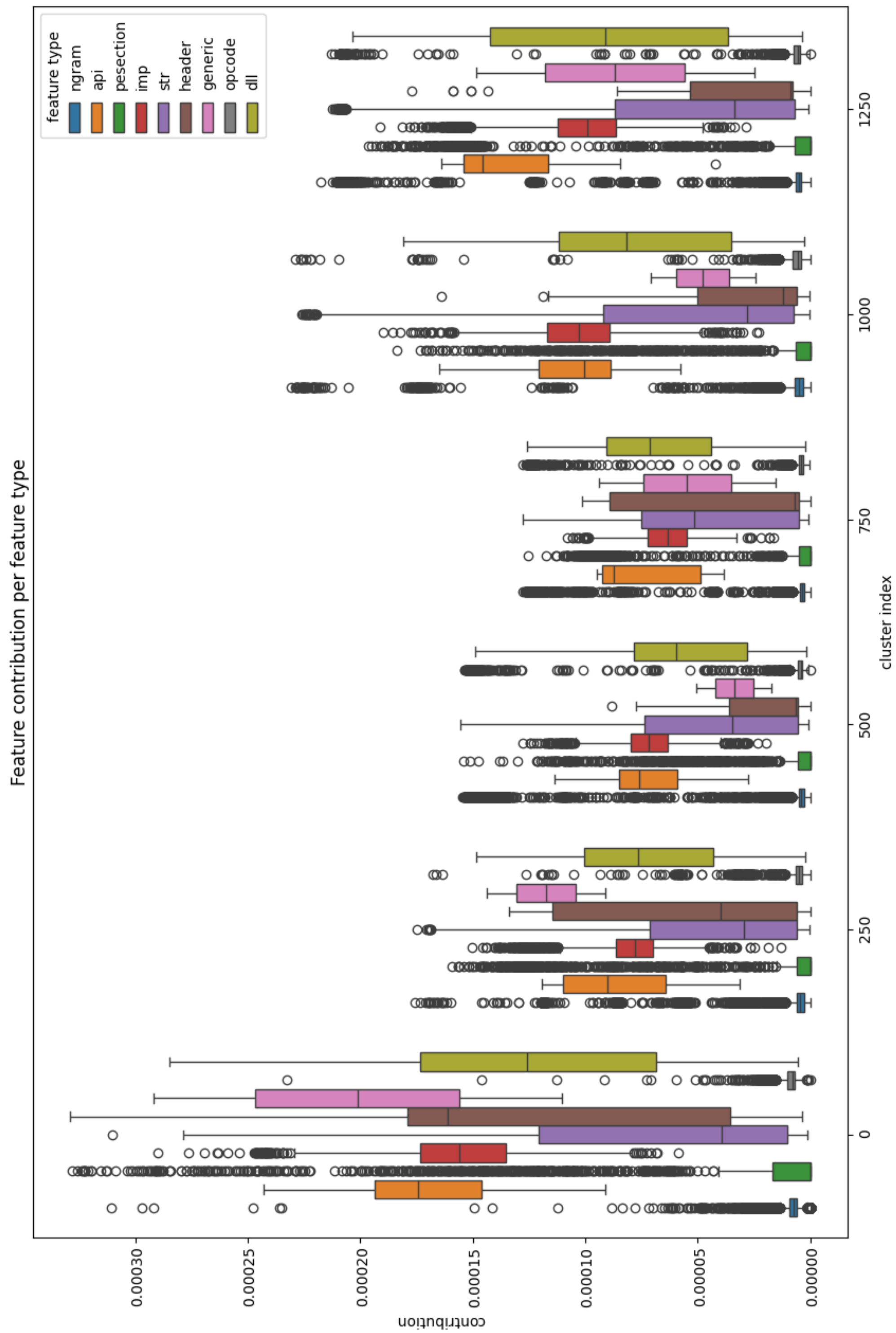


Figure 5.3: Feature importance distribution grouped per feature type and cluster

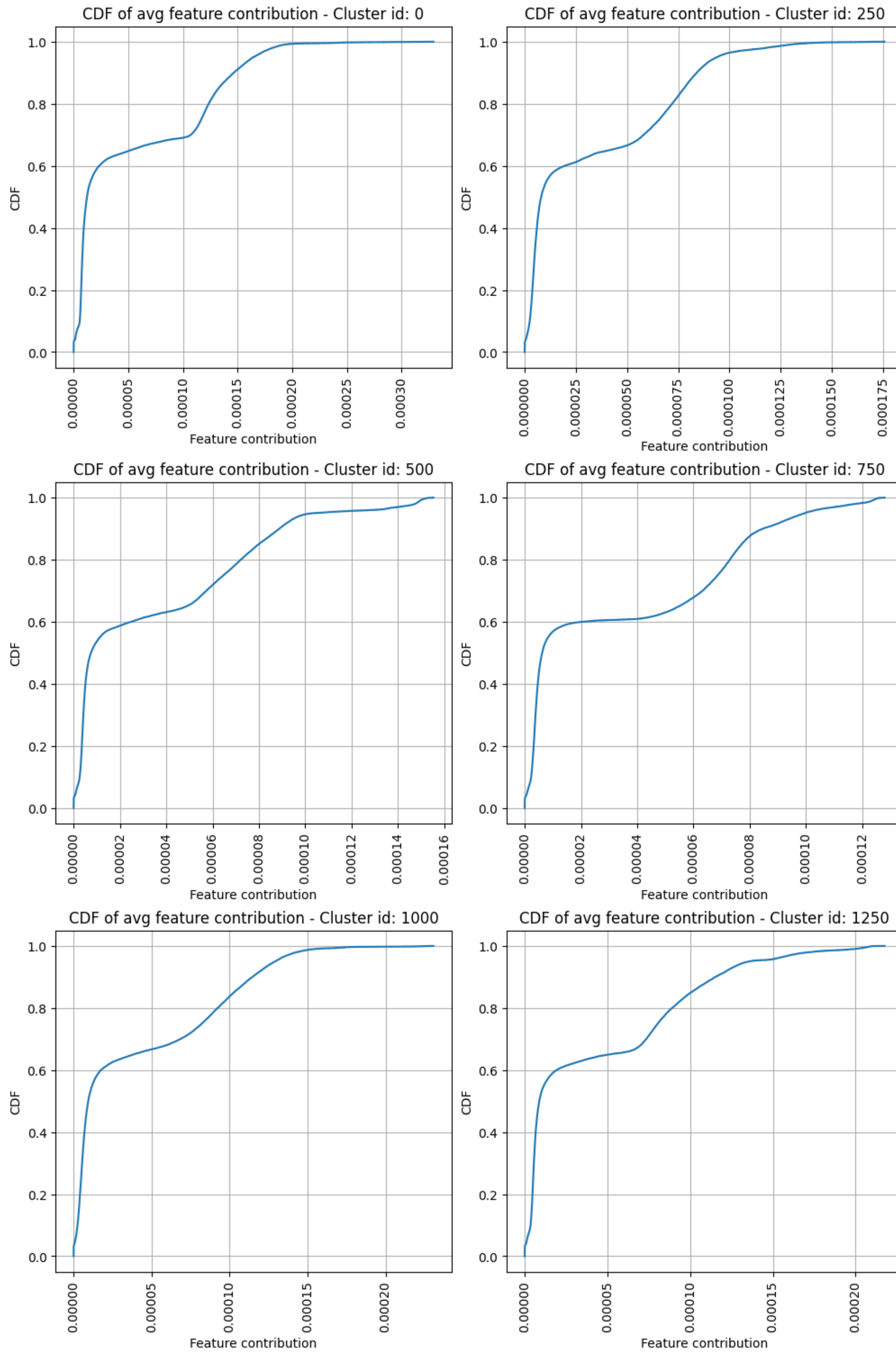
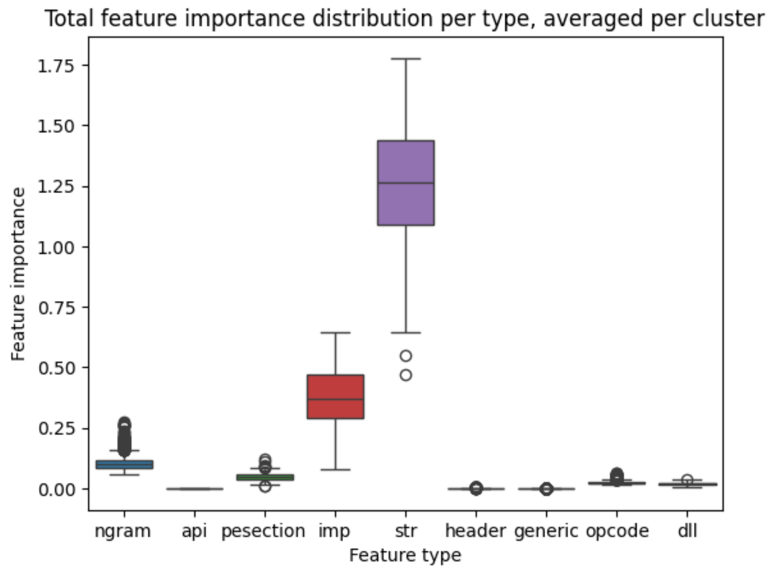
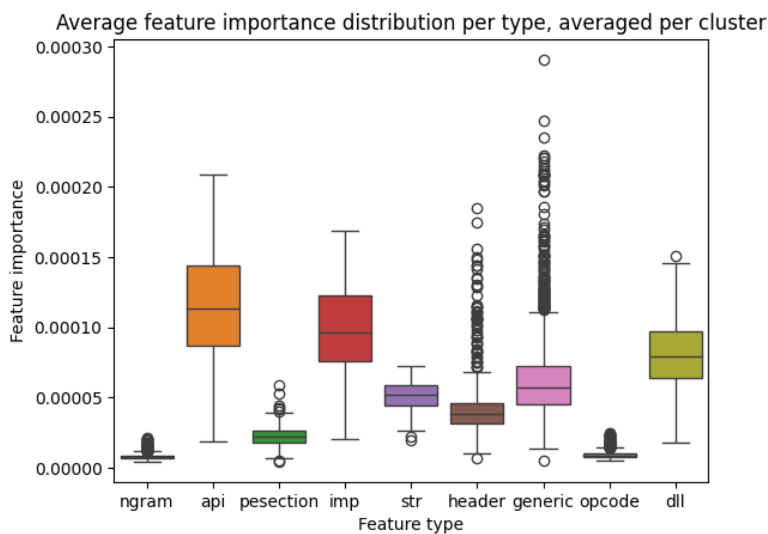


Figure 5.4: Cumulative distribution function of feature contribution per cluster

Figure 5.5: Feature contribution distribution per feature type (absolute and normalized).



(a) Total feature importance per type, averaged by cluster



(b) Average feature importance per type, averaged by cluster

Figure 5.6: Feature importances grouped for the whole dendrogram cut (n. 1500 clusters).

5.4 Results and Conclusions

From the generated plots we can draw several conclusions. The majority of features within each chosen cluster have near-zero contributions ($\approx$ 60% of the features) while only a few exhibit high contributions, varying from top 1% to 10%.

Overall, when considering all the clusters from the dendrogram cut, string features have the highest contribution, in absolute terms. However, when the contribution is normalized by the number of features per type (i.e. average contribution per type), this number drastically drops. This suggests that the relatively high importance of string features is primarily due to their large quantity rather than their individual importance.

On the other hand, imports have relatively high contributions both in absolute terms and on average.

Furthermore, n-grams and opcodes — 13000 and 2500 features, respectively — have low absolute and average contributions. This indicates that despite the high number of present features, low-level information of malware isn't useful for characterizing clusters at this level of the dendrogram.

One possible explanation is that fine-grained representations such as n-grams and opcodes capture micro-level variations that do not strongly influence the broader structural similarities captured by clustering. Instead, higher-level semantic features, such as API imports and DLLs, play a more dominant role in identifying meaningful relationships among malware samples.

Chapter 6

Concept drift detection

6.1 Introduction

This chapter assesses concept drift detection in the context of AVClass2 family labels. Given the initial time-based train/test split, a concept drift detection pipeline was executed to identify drifting families and examine the impact of emerging ones, unseen in the training set.

This work directly builds on the foundation established by the Transcendent project [BPPC24], which leverages **Conformal Evaluation**. The open-source code hosted on GitHub was modified to support multiclass problems.

The Transcendent project originally supports binary classification only. However, as the authors noted in their publication, the code can be extended to multiclass scenarios using a one-vs-all ensemble of Conformal Evaluators. Given the dataset's hundreds of labels, this approach was considered impractical.

To address this, a multiclass **Non-Conformity Measure (NCM)** was implemented, and the code was adapted to work with the *ICE (Inductive Conformal Evaluator)* solution. Although *ICE* is the least formally efficient, it is the most computationally viable option and is recommended by the authors (next to *CCE*).

While this work extends the Transcendent framework, the thresholding mechanism was temporarily disabled due to time constraints, with adjustments left for future work. Instead of using per-class thresholds, a global threshold was derived for all classes using a simplified criterion.

This chapter outlines the concept drift detection pipeline, emphasizing the design of the NCM and the experiments that have been carried out. The updated code is publicly available in a personal Transcendent fork.

6.2 Multiclass Non-Conformity Measure (NCM)

The first step in choosing the multiclass NCM is to define the ML model that will be used to classify malware samples into families, as the NCM should be derived accordingly.

While deep learning models may be an interesting way of classifying malware, they don't suit this dataset, as too few samples for each family are provided (100 per family, specifically). In the context of malware family classification, Random Forest model has proven in many works to be an effective solution though, as it achieves good performances. Thus, it has been chosen as a candidate algorithm for investigating concept drift.

From the literature, there exists a bunch of RF NCMs, mainly built for Conformal Predictors. The one chosen for this project is based on Random Forest proximities, presented in [DN10], and whose Conformal Evaluator solution is referred to *CP-RF-kNN*. The NCM, which from now on we will name ***RF-kNN***, has been proven to be valid and efficient, based on conducted experiments applied to Conformal Prediction.

> "*The nonconformity measure is the ratio of the average proximity of the example with examples of other classes to the average proximity of the example to examples of the same class. In both averages we consider only proximities of those k examples that have the greatest values of proximities among examples of the same class y and among all the other examples.*"

6.2.1 Random Forest Proximities

Random Forest Proximities is a method to measure the similarity between two samples using a pre-trained Random Forest model. Proposed by the author of the

RF algorithm, Leo Breiman, he considers them "one of the most useful tools in random forests" [1].

The measure comes from the idea that if two samples i and j activate the same leaves across the Decision Trees composing the Random Forest, they are similar as they share feature values.

That being said, proximities in Random Forest measure the similarity of two samples by the proportion of shared activated leaves. Formally, given samples i and j:

$$\text{prox}(i, j) = \text{prox}(j, i) = \frac{1}{|\text{Trees}|} \sum_{t \in \text{Trees}} [\text{leaf}(i, t) == \text{leaf}(j, t)] \tag{6.1}$$

where $\text{leaf}(i, t)$ is the index of the activated leaf for sample i and Decision Tree t.

RF proximities can be computed leveraging code available on Proximities and Prototypes with Random Forests guide. This implementation is adopted for this project.

6.3 Pipeline Overview

As mentioned in the introduction, experiments are carried out using the *ICE* solution (without thresholding). The pipeline consists of several sequential phases:

1. *Prelude*: the dataset is prepared, and any necessary preprocessing is performed. Additionally, the training dataset is further split into a proper training set and a calibration set, using random partitioning. The calibration set represents 34% of the full training set;

2. *Calibration*: The Random Forest model is trained on the proper training set and, leveraging *RF*-5*NN* NCM, the calibration points' p-values are computed to assess the algorithm's Credibility and Confidence. These two metrics are essential for the Alpha assessment phase, determining whether the NCM can effectively distinguish between correct and incorrect predictions using statistical evidence.

[1] `https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm#prox`

3. *Deployment*: The classifier is trained on the full training set, and Credibility and Confidence scores are calculated for the test set. Leveraging the global threshold identified in the previous step, concept drift is eventually detected.

6.4 Results and Analysis

6.4.1 Alpha assessment

The alpha assessment can be used to empirically investigate the quality of the underlying NCM. In Transcend [JSD+17], this is done by plotting the distribution of p-values for each class, further splitting it by correct vs incorrect predictions. If a viable threshold can be derived, with incorrect predictions having low p-values, it means the NCM can effectively capture predictions with low credibility and can be used to statistically support the decisions.

Rather than plotting the distribution for each class, in this project, the alpha assessment is carried out by simply splitting the distributions of correct vs incorrect predictions, as we have a high number of classes.

As shown in the following figure (6.1), the two distributions differ. This implies that the NCM *RF*-5*NN* is a good metric for discriminating whether the prediction is valid, providing statistical evidence.

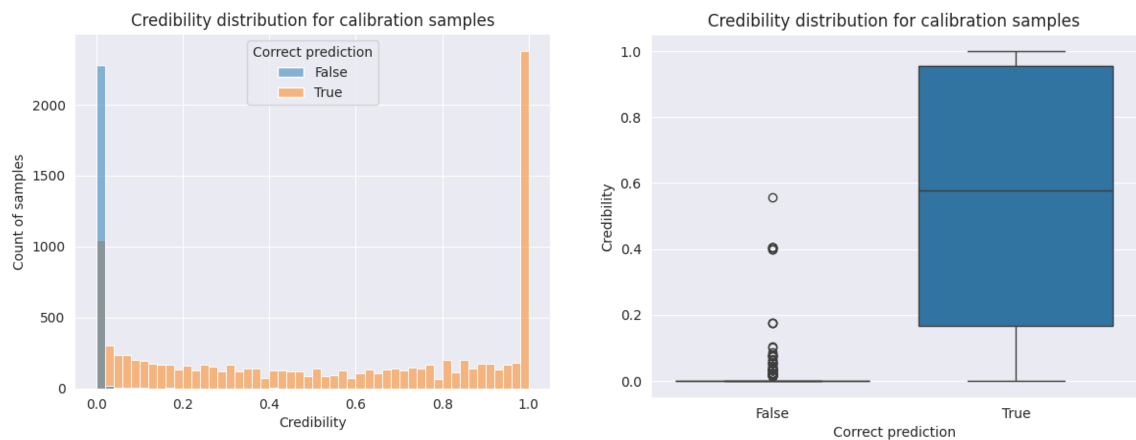


Figure 6.1: Alpha assessment: credibility score (p-value) distribution of the calibration set

6.4.2 Global threshold

The information provided by alpha assessment is also used for computing the global threshold t. Calibration points' p-values, or equivalently credibility values, are combined with the confidence score, which assesses whether the predicted point is also similar to other classes: it may happen that the choice made by the classifier is not linked to the highest p-value, suggesting that the confidence is sub-optimal.

Using the same criteria, the confidence score distribution is plotted, by splitting it by correct vs incorrect predictions.

The results show that both correct and incorrect predictions have high confidence scores, meaning that algorithm credibility is the highest p-value. The conclusion is that the algorithm can uniquely identify the classes and that calibration objects are not similar to two or more classes.

Since high algorithm confidence is reached in both cases, the global threshold t is derived using algorithm credibility only, with p-values $< t$ delineating drifted points, and credible prediction vice-versa.

By plotting the CDF of the credibility score from incorrect predictions, it's possible to see that there's a huge elbow after p-value 0. While this number may be a viable threshold, in concept drift detection applied to malware samples, prioritizing the detection of false positives over false negatives is generally preferred. Thus, we further analyze the conventional threshold of $0,05$. Using this threshold, we can see that it covers $98,88\%$ of incorrect choices. This threshold has been considered acceptable for the problem, as it covers even more incorrect predictions.

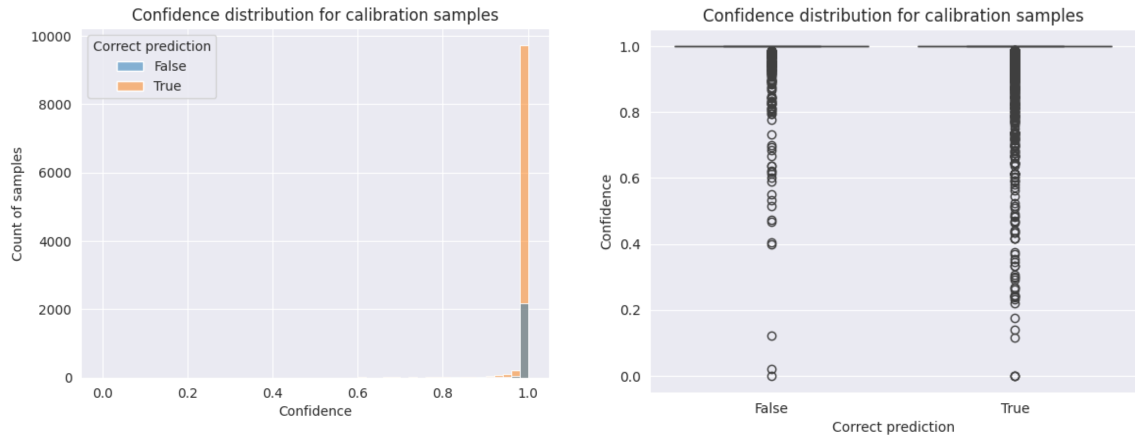


Figure 6.2: Confidence score distribution of calibration set

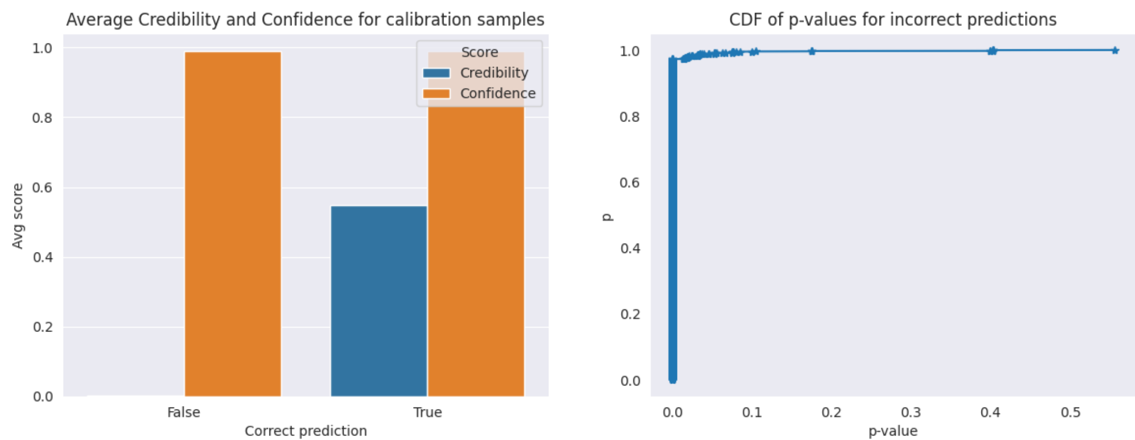


Figure 6.3: Thresholding information: average credibility and confidence comparison, CDF of p-values for incorrect predictions

6.4.3 Concept drift

Once the threshold is defined, the testing set is checked for concept drift. The Random Forest model is trained on the full training set and the same scores are computed: points whose credibility scores fall below the threshold are signalled as drifted. The analysis in this section is split into two cases: family evolution and unseen families analysis. The first one assesses whether already-seen families

in the training set have some form of evolution while the latter studies how new families are categorized between the known ones.

Malware family evolution

By considering testing set samples belonging to known malware families:

- The Random Forest model has $66,47\%$ accuracy;
- A total of 7648 points are drifting ($38,68\%$), belonging to 451 different families ($84,46\%$ of seen families);
- The number of known families that have all drifting testing points is 45. Most of them (31 - $68,89\%$), have less than 20% of the points in the testing dataset (see figure 6.5);
- 1573 ($7,96\%$) of them have incorrect predictions but high credibility scores. These samples belong to 191 distinct families;
- On average, each true family is predicted to $1,83$ different families.

Figure 6.4: Credibility score distribution of testing samples from seen families

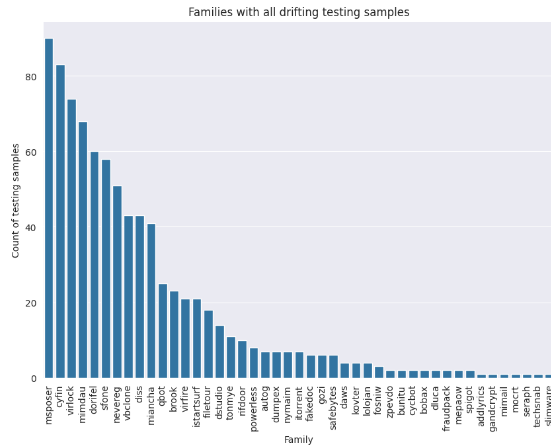


Figure 6.5: Families with all drifting samples in the testing set

Unseen malware families

For families not present in the training set, one possible experiment is to assess which ones are considered to be drifting and, conversely, check which known family they are classified to when most of their samples have high-credibility predictions. In the latter case, a high-credibility classification next to a single family prediction might suggest that the unseen family is an alias of a known one.

The number of families unseen in the training set is 26. By considering these families, some of them have a high credibility score. Specifically, 4 families have an average credibility score above the threshold. 3 of them 100% of its samples with high credibility. In figure 6.7 is shown the boxplot of the credibility score for these families, further splitting it by predicted families. As it's possible to see, these families are classified to one or a maximum of 2 families. These families share similar signatures and suggest they are aliases of predicted ones.

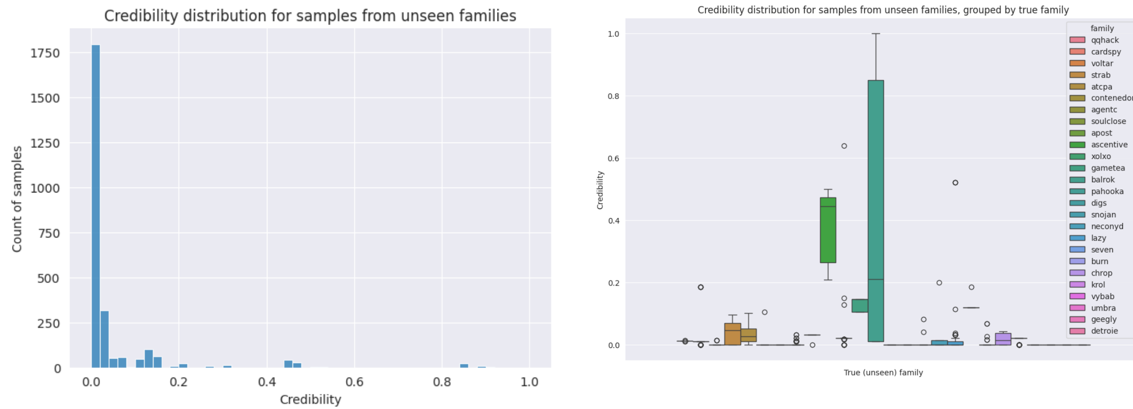


Figure 6.6: Credibility score distribution of testing samples from unseen families

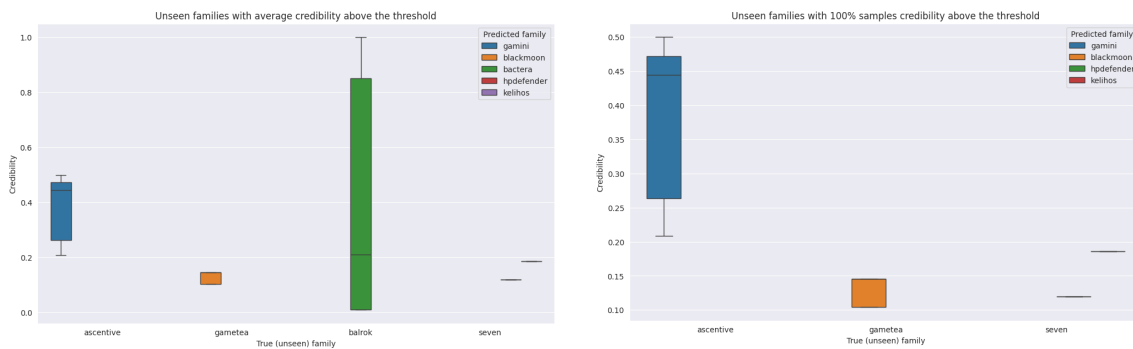


Figure 6.7: Credibility score distribution for testing samples from emerging families with average and 100% credibility above the threshold, respectively, grouped by predicted family.

Chapter 7

Conclusions

The work conducted in this thesis has resulted in the experimentation of multiple ML algorithms to address and shed light on the problem of malware clustering and concept drift.

Project milestones have been successfully completed. After redesigning and optimizing some phases, the dataset based on malware static features was built leveraging existing work. Different clustering solutions were evaluated, mostly regarding hierarchical algorithms, finally identifying the optimal number of clusters, which demonstrated good quality scores.

By combining clustering and XAI methods, insights into why samples are grouped together were revealed.

Even if not strictly interpretable, these static features may be useful to better understand the intrinsic relationship of malware, getting what are those features that are shared across samples and global information. For example, whenever clustering has optimal quality scores, we were able to see that higher-level information prevails over fine-grained features.

Furthermore, current labeling systems provide insights into malware families, but these algorithms are inherently probabilistic and cannot be used as perfect ground truth. Also, the fast evolution and heterogeneous nature of malware make the analysis more challenging. Clustering algorithms may be useful to address these problems, providing an automated data-driven tagging mechanism to verify the relationship between new and existing samples.

Parallel to malware clustering, concept drift definition and detection techniques are reviewed in the first instance, specifically taking into consideration state-of-the-art techniques for malware detection.

Through the injection of adversarial samples or the emergence of attack techniques, attackers can indeed alter the performance of ML classifiers. Leveraging concept drift detection, performance degradation can be identified in the early stages (either with respect to clustering results or another tagging mechanism) to trigger model retraining, preventing the machine from being infected by new types of malware.

By using AVClass2 labels, the last chapter of this thesis focused on addressing concept drift. Transcendent was applied with some modifications to the available project. Using the determined time-based split, we were able to conclude that the testing set appears to have drifted points.

7.1 Future work

This thesis experimented in two parallel lines with different missing points about malware analysis using static features.

First of all, the clustering results may serve as a preliminary analysis for further development. The hierarchical definition of clusters could be a method for addressing whether a new data point can be classified into one of the existing clusters, providing the user with the main features the sample shares with other malicious samples.

Moreover, clustering labels can be used to address concept drift concerning the found clusters, which could give deeper insights into changes from a data perspective.

This latter detection method, combined with data drift and outlier detection techniques, not only can be used to identify new emerging families but also new obfuscation techniques or polymorphic modifications designed to evade detection.

Lastly, future work should focus on integrating these labels (either clustering labels or other tags from known tools) with explainable drift detection techniques, to better interpret what are the main features that caused the drift and change over time.

Bibliography

[AR18] Hyrum S. Anderson and Phil Roth. Ember: An open dataset for training static pe malware machine learning models, 2018.

[ASLP07] Chid Apte, David Skillicorn, Bing Liu, and Srinivasan Parthasarathy. *Proceedings of the 2007 SIAM International Conference on Data Mining (SDM).* Society for Industrial and Applied Mathematics, Philadelphia, PA, 2007.

[BAK22] Firas Bayram, Bestoun S. Ahmed, and Andreas Kassler. From concept drift to model degradation: An overview on performance-aware drift detectors. *Knowledge-Based Systems*, 245:108632, 2022.

[BCH+09] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In ISOC, editor, *NDSS 2009, 16th Annual Network and Distributed System Security Symposium, February 8-11, 2009, San Diego, USA*, San Diego, 2009. © ISOC. Personal use of this material is permitted. The definitive version of this paper was published in NDSS 2009, 16th Annual Network and Distributed System Security Symposium, February 8-11, 2009, San Diego, USA.

[BKB17] Osbert Bastani, Carolyn Kim, and Hamsa Bastani. Interpreting blackbox models via model extraction. *CoRR*, abs/1705.08504, 2017.

[BOA+07] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In Christopher Kruegel, Richard Lippmann,

and Andrew Clark, editors, *Recent Advances in Intrusion Detection*, pages 178–197, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[BPPC24] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. Transcending transcend: Revisiting malware classification in the presence of concept drift, 2024.

[Can20] Marcello Cannone. Explainable ai for clustering algorithms. Master's thesis, Politecnico di Torino, Corso di laurea magistrale in Ingegneria Informatica (Computer Engineering), 2020.

[CH18] Yehonatan Cohen and Danny Hendler. Scalable detection of server-side polymorphic malware. *Knowledge-Based Systems*, 156:113–128, 2018.

[Cla24] ClamAV. Clamav documentation, 2024.

[DHA+23] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance, 2023.

[DN10] Dmitry Devetyarov and Ilia Nouretdinov. Prediction with confidence based on a random forest classifier. In Harris Papadopoulos, Andreas S. Andreou, and Max Bramer, editors, *Artificial Intelligence Applications and Innovations*, pages 37–44, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[GMCR04] João Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with drift detection. In Ana L. C. Bazzan and Sofiane Labidi, editors, *Advances in Artificial Intelligence – SBIA 2004*, pages 286–295, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[GP94] Kimberlee Gauvreau and Marcello Pagano. Why 5%? *Nutrition*, 10 1:93–4, 1994.

[GSCK23] Nor Zakiah Gorment, Ali Selamat, Lim Kok Cheng, and Ondrej Krejcar. Machine learning algorithm for malware detection: Taxonomy, current challenges, and future directions. *IEEE Access*, 11:141045–141089, 2023.

[How07] D.C. Howell. *Statistical Methods for Psychology.* Thomson Wadsworth, 2007.

[Ins23] AV-TEST Institute. New malware, 2023.

[JM11] Sachin Jain and Yogesh Kumar Meena. Byte level n–gram analysis for malware detection. In K. R. Venugopal and L. M. Patnaik, editors, *Computer Networks and Intelligent Computing*, pages 51–59, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Jon23] Shanice Jones. What is dynamic malware analysis? `https://www.bitdefender.com/blog/businessinsights/what-is-dynamic-malware-analysis/`, 2023.

[JRA20] Maciej Jaworski, Leszek Rutkowski, and Plamen Angelov. Concept drift detection using autoencoders in data streams processing. In Leszek Rutkowski, Rafał Scherer, Marcin Korytkowski, Witold Pedrycz, Ryszard Tadeusiewicz, and Jacek M. Zurada, editors, *Artificial Intelligence and Soft Computing*, pages 124–133, Cham, 2020. Springer International Publishing.

[JSD+17] Roberto Jordaney, Kumar Sharad, Santanu K. Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 625–642, Vancouver, BC, August 2017. USENIX Association.

[Kas21] Kaspersky. Machine learning for malware detection. Technical report, Kaspersky, 2021.

[Kas23] Kaspersky. Rising threats: cybercriminals unleash 411,000 malicious files daily in 2023, 2023.

[Kas24] Kaspersky. Kaspersky scan engine detection technologies, april 2024.

[KYMS16] Boojoong Kang, Suleiman Y. Yerima, Kieran Mclaughlin, and Sakir Sezer. N-opcode analysis for android malware classification and categorization. In *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, pages 1–7, 2016.

[Lam22] Thomas Lambart. Concept drift detection: An overview, 2022. Accessed: 2024-06-19.

[Mas51] Jr. Massey, Frank J. The kolmogorov-smirnov test for goodness of fit, March 1951.

[MAS14] Mangesh Musale, Thomas H. Austin, and Mark Stamp. Hunting for metamorphic javascript malware. *Journal of Computer Virology and Hacking Techniques*, 11(2):89–102, September 2014.

[McH13] Mary L. McHugh. The chi-square test of independence, 2013.

[MLJ+21] Yixuan Ma, Shuang Liu, Jiajun Jiang, Guanhong Chen, and Keqiu Li. A comprehensive study on learning-based PE malware family classification methods. *CoRR*, abs/2110.15552, 2021.

[MNK+21] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. Longitudinal study of the prevalence of malware evasive techniques. *CoRR*, abs/2112.11289, 2021.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[RMH21] Tina Rezaei, Farnoush Manavi, and Ali Hamzeh. A pe header-based method for malware detection using clustering and deep embedding techniques. *Journal of Information Security and Applications*, 60:102876, 2021.

[RZC+16] Edward Raff, Richard Zak, Russell Cox, Jared Sylvester, Paul Yacci, Rebecca Ward, Anna Tracy, Mark McLean, and Charles Nicholas. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, 14(1):1–20, September 2016.

[SC20] Silvia Sebastián and Juan Caballero. Avclass2: Massive malware tag extraction from av labels, 2020.

[SG04] Jeffrey C. Schlimmer and Richard Granger. Incremental learning from noisy data. *Machine Learning*, 1:317–354, 2004.

[SMdlR+14] Prasha Shrestha, Suraj Maharjan, Gabriela Ramírez de la Rosa, Alan Sprague, Thamar Solorio, and Gary Warner. Using string information for malware family identification. In Ana L.C. Bazzan and Karim Pichara, editors, *Advances in Artificial Intelligence – IBERAMIA 2014*, pages 686–697, Cham, 2014. Springer International Publishing.

[SS15] P.V. Shijo and A. Salim. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, 46:804–811, 2015. Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India.

[SSD] SSDSI. One way anova.

[VB18] Giovanni Vigna and Davide Balzarotti. When malware is Packin' heat. In *Enigma 2018 (Enigma 2018)*, Santa Clara, CA, January 2018. USENIX Association.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde,

Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[Vir] VirusTotal. Virustotal. [Online; accessed 27-July-2024].