Universidade Federal de Pernambuco

Walber de Macedo Rodrigues

Dynamic Ensemble of Classifiers and Security Relevant Methods of Android's API:
An Empirical Study

Recife

2022

Walber de Macedo Rodrigues

Dynamic Ensemble of Classifiers and Security Relevant Methods of Android's API:
An Empirical Study

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Aprendizagem de Máquina e Mineração

Orientador: George Darmiton da Cunha Cavalcanti

Recife

2022

Catalogação na fonte
Bibliotecária: Mônica Uchôa, CRB4-1010

R696d Rodrigues, Walber de Macedo.
Dynamic ensemble of classifiers and security relevant methods of android's API: an empirical study / Walber de Macedo Rodrigues. – 2022.
93 f.: il., fig., tab.

Orientador: George Darmiton da Cunha Cavalcanti.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn. Programa de Pós-graduação em Ciência da Computação. Recife, 2022.
Inclui referências e apêndices.

1. Security relevant methods. 2. Métodos de ensemble. 3. Sistema de múltiplos classificadores. 4. Ensenmble dinâmico. I. Cavalcanti, George Darmiton da Cunha (Orientador). II. Título.

681.3 CDD (23. ed.) UFPE- CCEN 2022 - 75

Walber de Macedo Rodrigues

“Dynamic Ensemble of Classifiers and Security Relevant Methods of Android’s API: An Empirical Study”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Inteligência Computacional

Aprovado em: 10/02/2022.

In memory of my grandfather, Walter Colaço Rodrigues, always as the energy in our hearts.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. George Cavalcanti, for his academic guidance, disponibility, and patience, during the difficulties of writing a master's degree, exacerbated by the past years.

A special thanks to all my family for the support, advices, and foundation provided during the years. In special, my gratitude to my mother, Maria Elisabete de Macedo Rodrigues, who is always supportive and sympathetic on day-to-day issues, and to my father, Walter Vieira Rodrigues, who helped discussing part of the mathematical analysis in this work. Without your support and academic encouragement through the years, concluding this work would not be possible.

I also would like to thank my friends at the RobôCIn and at the Projeto Samsung/CIn-UFPE. In particular Roberto Costa Fernandes for the support, and to Felipe Nunes Walsley, for the support, and help in elaborating experiments with embedding algorithms.

Finally, I would like to acknowledge the *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq) for the financial support received.

RESUMO

O sistema operacional Android disponibiliza funções e métodos de manuseio de dados sensíveis para proteger os dados dos usuários. Dados sensíveis são todo tipo de dados que podem identificar o usuário, como localização de GPS, dados biométricos e informações bancárias. A literatura de segurança Android propõe extrair *features* binárias de um método classificar-lo em uma das classes de *Security Relevant Methods*, agregando informação de o método manuseia dados sensíveis. Entretanto, existe uma lacuna na literatura onde não são avaliados algoritmos de *Ensemble* Dinâmico. Os algoritmos de *Ensemble* Dinâmico são estado da arte para Sistemas de Múltiplos classificadores, que por sua vez, não atacam objetivamente o tipo específico de *features* binárias. Assim sendo, este trabalho endereça a lacuna em relação a algoritmos de *Ensemble* Dinâmicos aplicados ao problema de classificação de *Security Relevant Methods*. Nossas análises motram que, ao contrário do que é inicialmente posto pela literatura, SVM não é o melhor classificador para esse problema, sendo MLP, *Random Forest*, *Gradient Boosted Decision Trees* e META-DES usando Random Forest como geração do *pool* os melhores resultados. Também constatamos que, em geral, algoritmos de *Ensemble* Dinâmico possuem uma desvantagem em relação aos classificadores monolíticos. Ademais, essa desvantagem é exarcebada em algoritmos que utilizam classificadores baseados em distância, como o OLP. Quando utlizamos o algoritmo de *embedding Triplet Loss*, observamos um aumento de performance para o kNN e OLP, mas não de outras técnicas de Ensemble Dinâmico, mostrando que um conjunto de *features* binárias tem impacto mais significativo sobre esses algoritmos.

Palavras-chaves: *Security Relevant Methods*. Métodos de Ensemble. Sistema de Múltiplos Classificadores. *Ensenmble* Dinâmico.

ABSTRACT

The Android operating system provides functions and methods to handle sensitive data to secure users' data. Sensitive data is every data that can identify the user, such as GPS location, biometric data, and banking data. The Android security literature proposes extracting binary features from a method and classifying the method into one of the Security Relevant Method's classes, adding information about how the method handles sensitive data. However, there is a gap in the literature where Dynamic Ensemble algorithms are not evaluated. Dynamic Ensemble techniques are state of the art on Multiple Classifiers Systems, which do not explicitly address the problem of a dataset of binary features. Thus, this work tackles the gap related to Dynamic Ensemble applied to Security Relevant Methods classification. Our analyzes show that, unlikely initially stated in the literature, SVM is not the best classifier for this problem, being MLP, Random Forest, Gradient Boosted Decision Trees, and META-DES using Random Forest as pool generation gives the best results. We also find that, in general, Dynamic Ensemble algorithms have a disadvantage compared to monolithic classifiers. Furthermore, this disadvantage is exacerbated in algorithms that use distance-based classifiers, such as OLP. When using the Triplet Loss embedding algorithm, we observed an increase in performance for kNN and OLP, but not for other Dynamic Ensemble techniques, showing that a set of binary features has a more significant impact on these algorithms.

Keywords: Security Relevant Methods. Ensemble Methods. Multiple Classifier Systems. Dynamic Ensemble.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
BCE	Binary Cross Entropy
BHF	Binary Hashing Function
DCS	Dynamic Classifier Selection
DES	Dynamic Ensemble Selection
DS	Dynamic Selection
DSEL	Dynamic Selection data set
IMEI	International Mobile Equipment Identity
kDN	K-Disagreeing Neighbors
kNN	K-Nearest Neighbors
kNNE	K-Nearest Neighbor Equality
KNORA-U	K-Nearest Oracle Union
KNORA-E	K-Nearest Oracle Elimination
MCS	Multiple Classifier Systems
MLP	Multilayer Perceptron
OLA	Overall Local Accuracy
OLP	Online Local Pool
PCA	Principal Component Analysis
RoC	Region of Competence
SGH	Self-generating Hyperplanes
SRM	Security Relevant Methods
SS	Static Selection
SSIM	Structural Similarity Index
SVD	Single-Value Decompositon
SVM	Support Vector Machine

LIST OF SYMBOLS

Γ	Greek letter Gamma
Σ	Greek letter Sigma
α	Greek letter alpha
θ	Greek letter theta
ψ	Greek letter psi
$\lvert\cdot\rvert$	Set cardinality
$\cup$	Set union
$\wedge$	Logical and
$\in$	Belongs to
$\leftarrow$	Variable assignment

CONTENTS

1 INTRODUCTION

Machine learning is a tool applicable to numerous problems, such as securing user data in smartphones. In order to guarantee data privacy in smartphone apps, the security literature uses methods that help the app developer track undesired data flow. Data Flow is the process of tracking which method and variables were used to transmit out of the device, create or store, modify, or unmodified data Wu et al. (2016).

Tools that perform data flow tracking analyze the source code of an app statically or dynamically. Static analysis interprets the source code and extracts how the app handles data without executing it, unlike dynamic analysis, which executes the source code in a safe environment Li et al. (2017).

In addition to source code, both analysis methods require a list of Security Relevant Methods (SRM). These methods are functions of the Android Application Programming Interface (API) that operate on sensitive data. Sensitive data is any private data related to a user or data that can be used to identify a person, such as photos, International Mobile Equipment Identity (IMEI), biometric data, GPS localization, and banking information. As it is up to the users to allow an app to access sensitive data, they can misinterpret if an application is trustworthy or not Rasthofer, Arzt and Bodden (2014).

As methods of code analysis have an essential role in flagging possible threats to be corrected by the developer, methods such as proposed by Rasthofer, Arzt and Bodden (2014) help to improve the analysis coverage by labeling previously unknown API methods. Nevertheless, treats can remain undetected by these methods. Thus, other methods are necessary to detect undetected threats during development.

Other techniques, such as proposed by Arp et al. (2014), focus on detecting apps with malicious behavior rather than helping the developer identify unwanted data flow. Although malware detection and SRM classification are different, some techniques share similar feature sets.

These feature sets are extracted directly from the app by analyzing its source code or behavior. This feature set comprehends binary features, representing the existence or not of specific characteristics, such as keywords, method return type, or data flow between parameters and the return variable.

We observed that works on SRM and malware classification explore deeply monolithic classifiers and static ensemble selection. Although Cruz, Sabourin and Cavalcanti (2018) reports that dynamic ensemble selection of classifiers outperforms monolithic classifiers and static ensembles, SRM classification does not explore these techniques.

Unlike in the literature on Dimmensionaly Reduction, we found that the literature on dynamic ensemble techniques does not present evidence of algorithm performance on pure binary feature datasets. The algorithms focus on solving classification problems grounded on continuous random variables. The binary-valued data is overlooked despite its applicability. Thus, an analysis of each step of MCS algorithms is required to understand how datasets of binary features impact these algorithms.

To fill the gaps in SRM classification literature and MCS, we focus on three main questions:

- How do binary features affect the performance of Dynamic Ensemble algorithms?
- How do Dynamic Ensemble techniques perform compared to Static Ensemble and monolithic classifiers in SRM Classification?
- What modifications can increase performance, either in preprocessing the dataset or modifying Dynamic Ensemble techniques?

In order to answer these questions, we need to understand how dynamic ensemble algorithms are affected by a binary feature dataset. Therefore, we analyze the aspects of a dataset of binary features and how it can theoretically impact the OLP algorithm.

Then, we proceed to evaluate the algorithms over the SRM dataset, comparing MCS and monolithic classifiers. We use Bagging, Boosting, and Random Forest as pool generation methods to increase knowledge of MCS algorithms' performance in this problem.

Analyzing the impact of binary features in MCS can also be measured when we embed the dataset into a real-valued space. In some applications, using an embedding algorithm can positively impact the performance of algorithms by removing variables that penalize specific classifiers. Deep Metric Learning, such as Triplet Loss, uses supervised learning to create a better-separated embedding to improve classification or clustering problems, as reported by Kaya and Bilge (2019).

Our results and analysis show that Dynamic Ensemble has worse performance than monolithic classifiers. The combination of META-DES and Random Forest has the least observable difference compared to monolithic classifiers. Moreover, embedding the data into a better-separated space does not improve the performance of MCS, except for the OLP. We interpreted these results as a limitation of the pool of classifiers to generate a good pool of classifiers to combine and form the MCS algorithms to select the most competent classifiers in the pool.

In addition to this analysis, we modify the OLP RoC, now considering the similarity between instances in the embedded space of a neural network trained with the metric learning algorithm, Triplet Loss. Our findings show that despite its improvements over the original OLP algorithm, it still does not outperforms monolithic classifiers.

The rest of the document is structured as follows: Chapter 2 introduces Android Security, fundamentals of Hamming spaces, ensemble methods, and embedding algorithms. In Chapter 3 we assess the impact of binary features over ensemble methods. In Chapter 4 we present the experimental procedure used during the experiments, presented in the Chapter 5. Finally, we draw our conclusions in Chapter 6.

2 BACKGROUND AND RELATED WORK

In this chapter, we present concepts related to SRM classification, Hamming space, Ensemble Methods, and Embedding algorithms. This build-up is necessary to ground the scope of the work and its contributions.

2.1 ANDROID SECURITY

First, we need to define sensitive data to understand its importance in Android security. Sensitive data is any data that can be used to identify a user, or any user's private information, such as photos, IMEI, biometric data, GPS localization, bank information, and private messages. Non-sensitive data is any dynamic information that does not identify the user. Non-sensitive data is often public or shared, such as application source code.

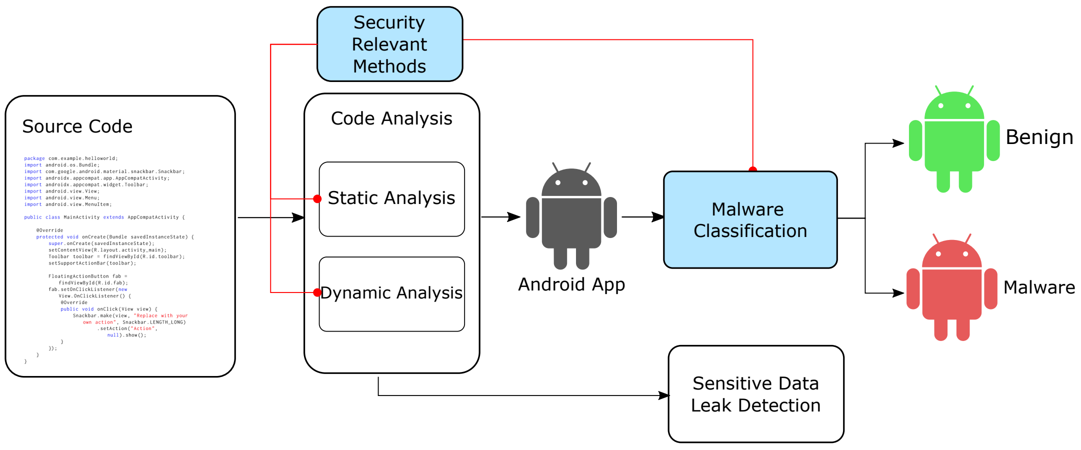
It is common to modularize snippets of code recurrently used into functions or methods. When a set of snippets are used to agglutinate functionalities to another software, this set gets called API. APIs are a powerful tool to developers, as APIs standardize how resources are being accessed, creating a standard and simple interface to resources and functionalities.

On mobile systems, like Android, it is necessary to call specific methods of its API to generate sensitive data. For example, by instantiating a `Camera` object in an app's code, which handles camera functionalities, and evoking the `Camera.takePicture` function to take a picture, the application uses the camera to take a picture. It is important to note that any application can operate with user-sensitive data, and to manage that, the Android system has a Permission System that enables the user to manage which apps are using sensitive data.

Before executing the app for the first time, the app informs the user of a list of permissions required to execute correctly. Then, the user can select which permissions the app is allowed to use. This Permission System is one of the most critical security systems in Android as it permits the user to manage which data apps can use. However, this system cannot make the user control how and where data is processed.

Malware is an app with illegitimate intentions while handling user-sensitive data. As with any other app, the malware uses the Android Permission System. However, the malware has the intention to harm the user, often disguising itself as a good-intentioned app.

Two main methods are used to detect malware or unexpected behavior in an app, Code



```
package com.example.helloworld;
import android.os.Bundle;
import com.google.android.material.snackbar.Snackbar;
import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab =
            findViewById(R.id.fab);
        fab.setOnClickListener(new
            View.OnClickListener() {
             @Override
             public void onClick(View view) {
                 Snackbar.make(view, "Replace with your
                     own action", Snackbar.LENGTH_LONG)
                           .setAction("Action",
                               null).show();
             }
        });
    }
}
```

Figure 1 – Process of sensitive data leak and malware classification. The steps which machine learning is employed are highlighted in blue.

Analysis and Malware Classification. Code Analysis uses the app source code to analyze interactions and leaves it to the developer to decide if any of these interactions are intended or not. Meanwhile, Malware Classification analyzes the app and classifies it into benign or malware. Figure 1 shows when in each stage of software development, both Malware and SRM Classification help to keep user data safe.

Before we proceed to methods that enforce and analyze if an application is purposefully leaking private data, we must first understand how the system generates private data and how a malicious app can leak it. In the following section, we will introduce Security Relevant Methods.

2.1.1 Security Relevant Methods

Let us back to our example of the `Camera.takePicture` function. This function generates or operates on sensitive data, which are called SRM. Knowledge of the SRM is essential to detect data leaks, as they carry information about how data is processed and if the method generates sensitive data. In this section, we will discuss the SRM classes: Sources, Sinks, Validators, and Authenticators and their impact.

The first class of SRM is called a Source method. Arzt et al. (2014) define this method as a source of sensitive data. These methods are available on the Android API to every app developer, as many of these sensitive data are essential to some features, such as localization in a map, messaging apps, or social media apps.

As we defined a source of sensitive data, the next step to characterize data leakage is to identify how data can leave the device. A Sink method, as defined by Li et al. (2015), provides the functionality to send data outside the device. This functionality includes sending data through an internet connection, writing files, or any method that makes an app capable of communicating with another app, remote host, or device. Thus, as the data can only leave by using a Sink, the unwanted relation between a Source and a Sink defines a data leak.

However, only considering Source and Sink methods as SRM classes, despises methods do not generate user data directly but are used to obfuscate or cipher sensitive data. In order to address the limitations of the Source-Sink model, another class was recently approached by Spoto et al. (2019), the Validator method. This method transforms user-sensitive data into non-sensitive data or treats possible harmful sequences of characters.

Spoto et al. (2019) argues that Validator methods are essential to analyze if an app is leaking sensitive data. Since Validator results do not yield sensitive data, not considering this behavior should result in false data leak reports. For example, if a code variable contains sensitive data, its contents are processed by a Validator, and after that, it reaches a Sink method, the analysis method should not report a data leak.

Finally, Piskachev and Bodden (2019) defines Authenticators as methods that have the functionality to change, elevate, or lower the application's privilege. These methods have the power to grant or revoke access of an app to system functionalities, remote data, or sensitive data.

Even though it is critical to know which methods yield sensitive data, Rasthofer, Arzt and Bodden (2014) noticed that developers or specialists label just a tiny portion of the APIs functions. As manual annotation is costly, Rasthofer introduces machine learning methods to classify API methods into SRM classes.

2.1.2 Malware Classification

A broad range of machine learning fields is applied to Android malware classification, analyzing network traffic and payload Wang et al. (2019), Pang et al. (2019), Narudin et al. (2016), Natural Language Processing to identify instruction patterns of the disassembled version of the application Yang et al. (2017), Zhang et al. (2019).

When studying malware classification, we can observe that the main contributions of recent works are the proposition of new features and different ensembles of algorithms. Regarding the features, we can divide between unstructured and structured data usage. We define structured data as tabular information, such as numerical, boolean, or categorical. Meanwhile, unstructured data is data that is represented in formats such as images, text, or audio samples.

Since unstructured data is out of the scope of this work, we will focus on works that utilize structured data.

Martín, Lara-Cabrera and Camacho (2019) conducted in-depth research over malware classification and observed that the most used features are sensitive API calls, used Permissions, and Intents. Liu et al. (2020) conducted a similar study and showed that 44 out of 100 works on malware classification consider API calls to classify malicious apps, and 63 out of 100 works use API calls or Permissions as features to malware classification.

Martín, Lara-Cabrera and Camacho (2019) and Liu et al. (2020) show that API calls are one of the most popular features used in the literature to classify malware. Recall Section 2.1, where we discussed that every app could have access to user-sensitive data. It is natural to think that API can be used to classify an app as malware; there is variability in the usage of this feature.

For example, Badhani and Muttoo (2019) uses the tag of a API package rather than analyzing the underlying API methods, as the methods can vary from different API versions. We will discuss further how features are extracted and the format used during classification.

To sumarize both SRM and malware classification, Table 1 shows that in the literature, homogeneous ensembles are explored by Zhu et al. (2018), Yerima, Sezer and Muttik (2015), and Sheen, Anitha and Natarajan (2015), which reports results for Random Forest, Rotation Forest, and monolithic classifiers.

The homogeneous ensemble combines multiple classifiers of a single type, such as multiple decision trees or multiple neural networks. Meanwhile, heterogeneous ensembles combine classifiers of different types, such as combining decision trees and neural networks.

For heterogeneous ensembles, Badhani and Muttoo (2019) introduces a combination of supervised and unsupervised learning, where a clustering algorithm splits uniform and diverse clusters. Diverse clusters comprise data labeled with multiple classes, and their samples are labeled by a heterogeneous ensemble, while uniform consists of only one class per cluster, and the cluster class labels instances. Whereas, Martín, Lara-Cabrera and Camacho (2019) performs the most detailed study of ensemble classifiers for Malware Classification so far, exploring both homogeneous and heterogeneous ensembles.

The next topic will introduce how to detect possible data leaks.

Reference	Monolithic Classifiers	Static Classifier Ensemble	Dynamic Classifiers Ensemble
SRMClassification			
Piskachev and Bodden (2019)	✓		
Sas, Bessi and Arcelli Fontana (2018)	✓	✓	
Rasthofer, Arzt and Bodden (2014)	✓		
MalwareClassification			
Taheri et al. (2020)	✓		
Martín, Lara-Cabrera and Camacho (2019)		✓	
Badhani and Muttoo (2019)		✓	
Zhu et al. (2018)		✓	
Zhu et al. (2017)	✓		
Sheen, Anitha and Natarajan (2015)	✓	✓	
Yerima, Sezer and Muttik (2015)	✓	✓	

Table 1 – Classifiers used in the literature.

2.1.3 Code Analysis

Previously we introduced what sensitive data is, how it is created, and how it could leak by using API methods. Now, we will discuss how a list of labeled API functions into SRM classes is useful.

There are two kinds of source code analysis, Static Analysis, and Dynamic Analysis. Static Analysis is a set of tools that examines the application code without executing it. The tool analyzes all the code structures and how data is processed in this process. Li et al. (2017) point that Static Analysis is mainly used to detect data leakage, privilege exploitation of benign apps, permission misuse, and energy efficiency.

On the other hand, Wong and Lie (2017) says that Dynamic Analysis is strictly dependent on the execution path, limiting to specific inputs that trigger an app's malicious behavior. Furthermore, Fernandes, Paupore and Rahmati (2016) uses Dynamic Analysis to enforce that sensitive data will not reach a sink method, executing the application in a safe environment with limitations over sink API access.

Although sending determined information to a host or another device can be the application intention, Arzt et al. (2014) note that a developer can use a third-party library to send this information to a malicious host. In this case, Static or Dynamic Analysis enables the developer to track unwanted behavior by the library.

Rasthofer, Arzt and Bodden (2014) proposes to expand the number of labeled SRM to improve the detection rate for Static and Dynamic Analysis methods. By employing supervised learning to classify if an API function generates, consumes sensitive data, or neither, the authors manually classify some functions from the Android API and extract 213 features. These are binary features and are classified as Syntactic, Semantic, and Data Flow features. These features intend to classify if a method is a Source or Sink and categorize them into source categories, explaining what information source methods provide.

The work of Sas, Bessi and Arcelli Fontana (2018) is similar to Rasthofer, Arzt and Bodden (2014) in the sense that both classify security-relevant methods. Moreover, they extend the classification process to general Java APIs and add a new Mixed method class, which is either a Source or a Sink of data and uses a slightly different feature set.

Piskachev and Bodden (2019) include Validators and Authenticators to provide the analysis methods a broader detection of vulnerabilities. Also, as previously observed in Sas, Bessi and Arcelli Fontana (2018), a method can perform two different tasks, either acting as data Source and Sink. However, Piskachev and Bodden (2019) uses binary classification to infer if a method belongs to a class. The system also analyzes a new codebase that a developer is working on and accepts user classification for new or unlearned methods.

Having studied how SRM classification impacts the detection of data leakage in Code Analysis methods and Malware Classification algorithms, we are going to discuss how features are extracted to perform SRM classification.

2.1.4 Feature Extraction

Feizollah et al. (2015), and Liu et al. (2020) conducted extensive studies of machine learning applied to malware detection published from 2010 to 2014 and from 2012 to 2019, respectively. The taxonomy proposed by Feizollah et al. (2015) classifies features into Static features, Dynamic features, Hybrid features, and Application Metadata features.

As stated by Feizollah et al. (2015), Static features relate to information extracted statically from the app without effectively executing it. These features are available in the Android Manifest file or the source code. The Android Manifest is a file that holds meta-information from an app, present in every Android application.

Dynamic features relate to information inferred from the software during execution, such as system calls, network traffic, system components, and user interaction.

Hybrid features are created when Static and Dynamic features are combined. Furthermore, Application Metadata features are information provided by the app developer found on the store where it is published, which involves application description, identification of the app creator, and category.

Static, Dynamic, and consequently, Hybrid features are interpreted as categorical features. For example, if an app is constrained only to use the phone camera, it will have a specific camera permission tag in the Manifest file. Considering that there are N different permissions, a reasonable way to extract this information from the Manifest File is to assign a numerical value to each N feature and fill a table with the respective permission.

However, as an app can require multiple permissions, a list containing each permission is necessary to track which permissions an app requires. Note that a variable size list could be undesired in a structured data environment. Thus, to avoid that, the table must contain a column for each permission, transforming categorical into binary features. This process can also be employed to extract some Dynamic features, tracking which Systems Calls or System Components are used during the application execution.

When analyzing the API methods, the source of the features is extracted differently from malware classification. Features extracted from SRM are divided into Syntactic, Semantic, and Data Flow features.

Syntactic features are related to the programming language, such as class and function modifiers, return type, variables, and arguments. Semantic features are naming patterns extracted from the methods and are defined by a specialist, which lists what information is helpful to classification. For example, a method starting with `Get` can characterize a Source method. Rasthofer, Arzt and Bodden (2014) points out that Syntactic and Semantic features help to track naming and coding patterns in the APIs.

Finally, Data Flow features explain how data is processed during the method call, for example, if an inner function call uses any of the method's parameters. Rasthofer, Arzt and Bodden (2014) considers Data Flow features as part of Syntactic and Semantic features, but Piskachev and Bodden (2019) reports that this feature class has its characteristics. Despite that, Data Flow features could be extracted dynamically. We observe that in SRM classification, these features are extracted statically.

Rasthofer, Arzt and Bodden (2014) proposes a feature set containing binary features reflecting the presence or not of keywords in the method's name, modifiers on the method such as *public* or *private*, method belonging to specific classes, parameter type, or parameter flow to the method's result. The complete list of features is present in Appendix A, Table 12.

In our analysis of Binary Datasets in Android Security, we observe the usage of Static,

Reference	Static features	Dynamic features	Hybrid features
SRMClassification			
Piskachev and Bodden (2019)	✓		
Sas, Bessi and Arcelli Fontana (2018)	✓		
Rasthofer, Arzt and Bodden (2014)	✓		
MalwareClassification			
Taheri et al. (2020)	✓		
Martín, Lara-Cabrera and Camacho (2019)	✓	✓	✓
Badhani and Muttoo (2019)	✓		
Zhu et al. (2018)	✓		
Zhu et al. (2017)		✓	
Sheen, Anitha and Natarajan (2015)	✓		
Yerima, Sezer and Muttik (2015)	✓		

Table 2 – Set of binary features and types of classifiers used in each work.

Dynamic, and Hybrid features for Malware Classification and only Static Features for SRM Classification, as shown in Table 2.

Although feature selection plays a significant role in machine learning, few works employ feature selection to obtain a more reliable feature set. Taheri et al. (2020) employs feature selection by using the most relevant features from a Random Forest classifier, showing that feature selection reduces the classifier's performance. On the other hand, Sheen, Anitha and Natarajan (2015) shows that Chi-Square, Relief, and Information Gain can be used to gain accuracy in some datasets.

Observing that malware and SRM classification uses binary features, we reserve the following section to study the specificities of these features and their natural mathematical space, the Hamming space.

2.2 THE HAMMING SPACE

The Hamming space $\mathcal{H}$ is the set of finite strings containing a sequence of n 0's and 1's. The maximum number of different strings is 2^n, as each element in the string is binary and can only assume two values. We will write each string $h \in \mathcal{H}$, also called binary string or binary code, as the *n*-tuple in Equation 2.1.

$$h = (x_1, x_2, ..., x_n) \qquad x_i = \{0, 1\}. \tag{2.1}$$

We can relate strings in the Hamming space to the computer's binary representations of numbers. For example, the natural number 10 can be represented as the binary string 1010

with length $n = 4$, using the Algorithm 1. The transformation of data into a binary string is called Binary Hashing Function (BHF), we describe the BHF of a decimal number.

Algorithm 1 Natural Number to Binary String

```
 1: procedure SimpleBHF($x$)
Require: $x \geq 0$
 2:     $n \leftarrow \lceil log_2(x) \rceil$                  ▷ ⌈ ⌉ is the ceil function
 3:     $h \leftarrow \{0\}^n$
 4:     while $n > 0$ do
 5:         $h[n-1] \leftarrow x\%2$                   ▷ % stands for modulo operator
 6:         $x \leftarrow x/2$           ▷ / stands for the integer division function
 7:         $n \leftarrow n - 1$
 8:     end while
 9:     return $h$
10: end procedure
```

The similarity metric between two instances in the Hamming space is the editing distance between two binary strings. This distance function is called the Hamming distance. The Hamming distance D_h is defined by the Equation 2.2 and represents how many elements are different between two binary codes.

$$D_h(x, y) = \sum_{i=0}^{n} |x_i - y_i| \tag{2.2}$$

As stated by Taheri et al. (2020), the Minkowski distance and the Hamming distance are equivalent for binary codes. The Minkowski distance D_m, shown in Equation 2.3 is the generation of the Euclidean distance, when $p = 2$.

$$D_m(x, y) = \sqrt[p]{\sum_{i=0}^{n} |x_i - y_i|^p} \qquad p \geq 1. \tag{2.3}$$

BHF are particularly useful when they are designed to learn how to create a mapping while maintaining the original neighborhood structure of the original data. Like transforming natural numbers to binary strings, the transformation of data, such as images, to binary code can also be performed using a BHF. The applicability is related to the reduction of complexity during a search of similar examples in a dataset.

For example, let us take the pixel-wise similarity between two images of size 1024×1024 pixels. In this case, the similarity is the mean difference between every pixel in the two images. When computing the similarity between the images, $1024 \times 1024 \times 3$ operations compute the difference between each pixel, then more 1024 sums are performed to sum all the differences, in a total of $(1024 \times 1024 \times 3) + 1024$ operations performed, which is translated into the complexity of n^2.

When transforming the image into a binary string, the similarity between two strings is the size of the string n. Thus, BHF are a powerful tool to reduce complexity in similarity search, as computation of similarity between binary instances becomes more computationally efficient.

As far as we are concerned, the Dynamic Selection (DS) literature does not cover the classification of instances with pure binary features. Meanwhile, tasks that perform similarity search on high-dimensionality data are favored when transformed to Hamming space, as neighborhood estimation becomes less computationally costly.

2.3 ENSEMBLE METHODS

Machine Learning is the branch of Computer Science that investigates algorithms that learn from data. It is common to think of a single model that classifies data in a table, detects objects in an image, or clusters data.

We can divide machine learning models into two main groups, monolithic classifiers and Ensemble Methods, also known as MCS. Classifiers such as Perceptron, Decision Tree, kNN, SVM and MLP are considered monolithic classifiers, as they do not use or combine multiple classifiers. Their objective is to learn the whole dataset by using a single classifier.

For Kuncheva (2014), the philosophy to employ an ensemble of classifiers is to create a more accurate classifier in detriment to a higher computational cost when compared to the single base model. Instead of searching for the best classifier and features, the objective is to find the best set of classifiers and the best combination method.

Zhang and Ma (2012) justifies this philosophy statically, arguing that it is common for a low-biased classifier to have a high variance. So, a classifier with a good performance on a specific training set can have a poor performance on a different set. Thus, a combination of classifiers reduces the variability while maintaining a good performance.

Britto, Sabourin and Oliveira (2014) reviews MCS techniques and proposes a taxonomy, breaking MCS into three parts: Pool Generation, Selection and Integration. The Pool Generation is the step reserved to train the pool of classifiers, followed by selecting the classifier set, called the Selection phase. And then, the Integration phase combines the classifiers selected.

2.3.1 Pool Generation

As introduced previously, the Pool Generation, or Overproduction phase, creates the classifiers used during the Selection phase. In this step, the objective is to create a set of diverse classifiers. Kuncheva (2014) and Zhang and Ma (2012) state that diverse classifiers are the critical point that improves classification performance, as these differences reflect missing and hitting different instances. For Kuncheva (2014), diversity represents variability in the errors of the base classifiers.

The pool of classifiers can assume two forms, a homogeneous pool or a heterogeneous pool. A heterogeneous pool combines a set of different classification algorithms. In this case, diversity is achieved by using different algorithms that generate different classification regions.

Meanwhile, a homogeneous pool consists of a single class of classifiers as the base for the pool. Britto, Sabourin and Oliveira (2014) states that diversity, in this case, is achieved by training each classifier with different instances in the dataset, changing the initial parameters of the classifiers, or using different features.

From the pool of classifiers arises the concept of the Oracle. The Oracle is a theoretical model, defined by Kuncheva (2002), which defines a classifier that always selects the classifier that correctly predicts the test instance from the pool. This model is observed by Cruz, Sabourin and Cavalcanti (2018) as an upper bound of MCS, as it is not possible to know the label of a test instance. However, this model is considered too optimistic by Souza et al. (2017) as DS schemes have a significant performance gap between them and the Oracle.

As Ensemble Methods fundamentally uses a set of classifiers, we will now explore four pool generation methods: Bagging, Boosting, Random Forest, and Self-generating Hyperplanes.

The Bagging algorithm splits a dataset into N smaller sets with replacement. Breiman (1996) introduced this method, intended to be used together with an unstable, or weak, classifier, where small changes in the training data create a significant change in the predictions. Then, combining these classifiers creates a more robust and accurate classifier.

Freund and Schapire (1997) proposed the Boosting algorithm, where rather than dividing the dataset into smaller sets, the wrongfully classified instances from a classifier are used to train another classifier. Freund and Schapire (1997) proposed this technique for two-class classification and regression problems and was further improved for multi-class problems by Zhu et al. (2009).

The Random Forest overproduction algorithm, proposed by Ho (1998), is the algorithm used by the Random Forest classifier to create its pool of classifiers. The Random Forest combines Bagging Breiman (1996) and a Random Subspace algorithm Ho (1998) that randomly selects features at each level of the Decision Tree. In other words, different features are used at each level of the tree.

SGH, proposed by Souza et al. (2017), has the objective to generate a pool of classifiers, ensuring that every instance in the training set is correctly classified. The SGH pool generation approaches the problem of relying on the selection method to correctly select the local Oracles by creating a pool of Oracles. By using SGH and comparing with the expected results by the Oracle model, they argue that Dynamic Classifier Selection (DCS) algorithms use local data to select the best classifier, and the Oracle is a global measure. They suggested that the rate that a DCS technique selects the correct classifier for a given instance was a better upper bound.

The SGH method is iterative and uses the perceptron as the base classifier. The preference for perceptrons comes from the theory of the pool of weak classifiers, Ko, Sabourin and Britto (2008) demonstrates that Dynamic Ensemble Selection (DES) techniques marginally improve accuracy.

The Algorithm 2 describes the SGH. The algorithm computes the centroid of each class, selects the two centroids with the biggest distance, and separates them by a linear classifier. In other words, a perceptron is placed between the two centroids. Then, the perceptron is added to the pool of classifiers and used to label every instance of the training set. Each correctly classified instance is removed from the current training instances, and these steps are performed iteratively until the training set is empty.

After understanding the mechanisms behind pool generation, the question of how to select the best classifiers arises.

2.3.2 Selection

The selection method defines a heuristic used to select the most suited classifiers. If the algorithm will select a single classifier DCS, or an ensemble of classifiers DES. Cruz, Sabourin and Cavalcanti (2018) states that the selection can be performed statically, during training time, or dynamically, during inference time.

Cruz, Sabourin and Cavalcanti (2018) describes three main categories of Ensemble Methods by its selection approach: Static Selection, Dynamic Classifier Selection, and Dynamic

Algorithm 2 Pseudocode of SGH

```
 1: procedure SGH($\Gamma$)
Require: Training dataset $\Gamma \leftarrow \{x_1, x_2, ..., x_N\}$
 2:     $Pool \leftarrow \{\}$
 3:     while $\Gamma \neq \{\}$ do
 4:         $R \leftarrow getCentroids(\Gamma)$              ▷ Calculate centroid of each class
 5:         $r_1, r_2 \leftarrow selectCentroids(R)$         ▷ Select the most distant centroids
 6:         $p \leftarrow placeHyperplane(r_1, r_2)$         ▷ Place a hyperplane between $r_1$ and $r_2$
 7:         $i \leftarrow 1$
 8:         while $i < N$ do
 9:             if $p(x_i) = label(x_i)$ then
10:                 $\Gamma \leftarrow \Gamma - x_i$         ▷ Remove correctly classified instances
11:             end if
12:         end while
13:         $Pool \leftarrow Pool \cup p$
14:     end while
15:     return $Pool$
16: end procedure
```

Ensemble Selection, which we can further combine into Static Selection (SS) and DS. SS selects the ensemble of classifiers during the training phase and then uses this ensemble to label the test instances. Meanwhile, DS each unknown example will be used to select a classifier or an ensemble of classifiers.

An example of such a heuristic is to select the classifier with higher accuracy in the pool, evaluated in a set of instances, ideally, different from the training. Then, it is used to label the test instances. This heuristic that we just described is the Single Best method, a DCS static method, as it selects a single classifier during training.

2.3.3 Integration

The last step on a MCS is the Integration step. Also called Aggregation, the predictions of multiple selected classifiers are combined into a single prediction in this phase. Cruz, Sabourin and Cavalcanti (2018) breaks the integration approaches into Non-trainable, Trainable, and Dynamic weighting.

Non-trainable integration techniques are simple rules that are used to fuse multiple model outputs without extra complexity. Kittler et al. (1998) presents rules such as the Sum, Product, Maximum, Minimum, Median, and the Majority Voting of the outputs. The Majority Voting rule is one of the most used integration algorithms. By using the mode of the ensemble's prediction, it is an effective way to combine independent classifiers, as reported by Cruz, Sabourin and Cavalcanti (2018).

Trainable approaches combine multiple outputs by another learning algorithm using the training data and are known to be better than non-trainable approaches, as stated by Cruz, Sabourin and Cavalcanti (2018). Finally, the Dynamic weighting algorithms attribute dynamic weights to each classifier, estimated by the classifier's performance in the neighborhood of the test instance.

2.3.4 Dynamic Selection

Previously, in Section 2.3.2, we gave an example of a SS technique, where the classifier is selected during training and later used for test instances. Now, we will introduce basic concepts to clarify DS techniques.

In Dynamic Selection (DS), the selection of competent classifiers is performed during inference time, selecting a single or a set of classifiers based on the test instance. Note that either a single or an ensemble of classifiers can be selected dynamically. Thus, the main difference between static and dynamic selection methods is to select the most fitted set of classifiers for a specific instance.

Britto, Sabourin and Oliveira (2014) report the viability of such techniques by directly comparing DS algorithm with the selection of the best classifier in the pool and a static combination of the best classifiers, showing that DS techniques are superior. Other works also demonstrate the superior performance in the DS literature, such as Cavalin, Sabourin and Suen (2013) and Ko, Sabourin and Britto (2008).

In the review made by Cruz, Sabourin and Cavalcanti (2018), the authors present a taxonomy that divides DS techniques into three main steps, the definition of the region of competence, the determination of the selection criteria, and the determination of the selection technique.

2.3.4.1 Region of Competence

Didaci and Giacinto (2004) defines that a classifier has both a "region of competence" and a "region of lack of competence". The region of competence is an area in the feature space in which a classifier has a low error rate. Meanwhile, the region of lack of competence is where the classifier has its worse performance. However, how do we define the region of competence or lack of competence for an algorithm?

In his review, Cruz, Sabourin and Cavalcanti (2018) observes that the selection of the local region is essential to DS techniques, as these algorithms are sensitive to the distribution of the local region. Also, they present four methods of RoC definition: using kNN, employing clusterization algorithms, using decision, or by a potential function.

The most used method is the kNN approach, as observed by Cruz, Sabourin and Cavalcanti (2018). This method uses the test instance to select k nearest instances in DSEL. Each classifier in the pool has its competence evaluated in the test instance's neighborhood. This process is described by Didaci and Giacinto (2004), used on algorithms such as Overall Local Accuracy (OLA), proposed by Woods, Jr and Bowyer (1997), and KNORA-U and KNORA-E, proposed by Ko, Sabourin and Britto (2008).

The first step of clustering methods is to cluster the data in DSEL. For example, Kuncheva (2000) uses K-means as the clusterization algorithm. Each centroid represents patterns in the feature space, enabling the computation of similarity between test instances and DSEL features. The competence of classifiers is measured for all instances in every cluster. Then, the cluster representing the test instance is calculated, and the most competent classifiers in the cluster are used to label the test instance.

Proceeding to potential functions, Cruz, Sabourin and Cavalcanti (2018) observes that this method differs from clustering methods and kNN methods, as rather than using a subset of DSEL, is used the whole DSEL. First, the probability of a base classifier correctly labeling the instances in the DSEL is calculated. Then, a function of the similarity between the test instance and each DSEL instance scales the probability. An example for this algorithm is the potential function described by (WOLOSZYNSKI; KURZYNSKI, 2011), defined in the Equation 2.4, where ψ_l is the l classifier in the pool, x the test instance, x_k the k instance in the DSEL, C is a function that assigns 1 and -1 if a classifier correctly or incorrectly classifies an instance, and $dist$ is the euclidean distance.

$$c(\psi_l, x) = \frac{\sum_{k=1}^{N} C(\psi_l, x_k) exp(-dist(x, x_k)^2)}{\sum_{k=1}^{N} exp(-dist(x, x_k)^2)} \tag{2.4}$$

Finally, the decision space approach, rather than using the instances in DSEL, uses the output profile of the base classifiers, which is the set of predictions of a classifier of the instances in DSEL. The output profile of the test instance is compared with the output profiles in the

DSEL to calculate the RoC. A method that uses this technique is the META-DES, proposed by Cruz et al. (2015), discussed later in this document.

2.3.4.2 Selection Criteria

The selection criteria define how to estimate the competence level of the classifier for a test instance. Cruz, Sabourin and Cavalcanti (2018) divides the selection criteria into two groups: individual-based and group-based measures.

Individual-based measures evaluate the competence of each base classifier in the pool independently. Cruz, Sabourin and Cavalcanti (2018) reports that this category can be further divided into seven groups based on the type of information used to evaluate the classifiers. The groups identified by the authors are Ranking, Accuracy, Probabilistic, Behavior, Oracle, Data complexity, and Meta-learning.

On the other hand, group-based measures are composed of algorithms that consider the ensemble's final composition. This composition can be measured differently, considering the pool's diversity, data handling, or ambiguity. These measures have the objective of selecting classifiers that will improve the ensemble. Aksela (2003) lists diversity measurements, for example, the correlation between the errors of the classifiers, the ratio between different and same errors, and the weighted count of errors and correct results that are used to select classifiers with complementary information.

Before we proceed to specific algorithms, we must discuss two required concepts, the Oracle model and Instance Hardness. Recalling the concept of Oracle, introduced in Section 2.3.1, it is a theoretical model which always selects from the pool the classifier that correctly predicts the test instance.

Instance Hardness, as presented by Smith, Martinez and Giraud-Carrier (2014), is a measure of the misclassification likelihood of an instance, manifesting how hard it is to be labeled. Walmsley et al. (2018) traces a parallel between instance hardness and noisy samples and applies it to modify the Bagging algorithm, avoiding noisy samples in the datasets that train the pool of classifiers.

The kDN score of an instance is defined by the Equation 2.5, representing the ratio of the number of neighbors that have a different label than the test instance. In Equation 2.5, kNN is the set of nearest neighbors of an instance x, k is the number of neighbors, $label$ is the class label of an instance, and the modulo sign $||$ represent the size of a set.

$$kDN(x) = \frac{\mid \{i \mid i \in kNN(x) \wedge label(i) \neq label(x)\} \mid}{k} \tag{2.5}$$

2.3.4.3 Selection Method

After our exposition of the required fundamentals, let us continue the discussion of the selection methods. We will discuss five techniques corresponding to the individual accuracy (OLA), individual oracle (KNORA-U, KNORA-E and OLP) and individual meta-learning (META-DES) taxonomy defined by Cruz, Sabourin and Cavalcanti (2018).

From the OLA, Algorithm 3, we note its simple decision process. The algorithm selects the best classifier in the pool based on its accuracy in the RoC. Line 5 computes and stores the accuracy for a classifier c_i, and then the best classifier is selected in line 7.

Algorithm 3 Overall Local Accuracy algorithm

```
1: procedure OLA($x_q, \Gamma, C, k$)
Require: Query pattern $x_q$
Require: DSEL set $\Gamma = \{x_1, x_2, ..., x_N\}$
Require: Pool of classifiers $C = \{c_1, c_2, ...c_M\}$
Require: Neighborhood size $k$
2:     $\theta \leftarrow kNN(x, \Gamma)$                              ▷ Gets the k-nearest neighbors
3:     $\alpha \leftarrow \{\}$                                         ▷ Define an empty set
4:     for all $c_i \in C$ do
5:         $\alpha_{i,q} \leftarrow |\{\theta_j|\theta_j \in \theta \wedge label(\theta_j) = c_i(\theta_j)\}|/k$ ▷ Where || is the size function of a set
   and $label$ is the correct sample label
6:     end for
7:     $c* \leftarrow c_{best}, best = \text{argmax}\, \alpha_{i,q}$
8:     return The most competent classifier $c*$
9: end procedure
```

Now, we will discuss the selection criteria that use the Oracle model, the KNORA-E and KNORA-U. Unlike OLA, KNORA-U and KNORA-E select an ensemble of classifiers, which are combined using majority voting scheme.

The KNORA-U, described in Algorithm 4, selects from the pool all the classifiers that correctly label at least one instance of the RoC. Meanwhile, the KNORA-E, described in Algorithm 5, selects only the classifiers that correctly label all the instances in the RoC.

Let us continue our discussion of selection criteria with OLP. The algorithm has three steps, the RoC Evaluation, the Local Pool Generation, and the Generalization. The RoC Evaluation estimates if the test instance is hard enough to be classified with a pool of Perceptrons, the

Algorithm 4 K-Nearest Oracle Union algorithm

```
 1: procedure KNORAU($x_q, \Gamma, C, k$)
Require: Query pattern $x_q$
Require: DSEL set $\Gamma = \{x_1, x_2, ..., x_N\}$
Require: Pool of classifiers $C = \{c_1, c_2, ...c_M\}$
Require: Neighborhood size $k$
 2:     $C' = \{\}$                              ▷ The pool of competent classifiers
 3:     $\theta \leftarrow kNN(x, \Gamma, k)$                  ▷ Gets the k-nearest neighbors
 4:     for all $c_i \in C$ do
 5:         $\alpha_{i,q} \leftarrow |\{\theta_j | \theta_j \in \theta \wedge label(\theta_j) = c_i(\theta_j)\}|/k$ ▷ Where || is the size function of a set
    and $label$ is the correct sample label
 6:         if $\theta_j > 0$ then
 7:             $C' \leftarrow C' \cup c_i$
 8:         end if
 9:     end for
10:     return The pool $C'$
11: end procedure
```

Algorithm 5 K-Nearest Oracle Elimination algorithm

```
 1: procedure KNORAE($x_q, \Gamma, C, k$)
Require: Query pattern $x_q$
Require: DSEL set $\Gamma = \{x_1, x_2, ..., x_N\}$
Require: Pool of classifiers $C = \{c_1, c_2, ...c_M\}$
Require: Neighborhood size $k$
 2:     while $k > 0$ do
 3:         $C' = \{\}$                          ▷ The pool of competent classifiers
 4:         $\theta \leftarrow kNN(x, \Gamma, k)$              ▷ Gets the k-nearest neighbors
 5:         for all $c_i \in C$ do
 6:             $\alpha_{i,q} \leftarrow |\{\theta_j | \theta_j \in \theta \wedge label(\theta_j) = c_i(\theta_j)\}|/k$ ▷ Where || is the size function of a
    set and $label$ is the correct sample label
 7:             if $\theta_j = 1$ then
 8:                 $C' \leftarrow C' \cup c_i$
 9:             end if
10:         end for
11:         if $|C'| = 0$ then
12:             $k = k - 1$
13:         else
14:             Break
15:         end if
16:     end while
17:     return The pool $C'$
18: end procedure
```

Local Pool Generation creates the pool of perceptrons and the Generalization phase combines the pool using majority voting.

First, the RoC Evaluation phase selects the instance's neighborhood using kNN. The objective of extracting the region is to evaluate if the RoC is hard enough to be classified by a pool of classifiers. The RoC is considered to be hard if it has at least one instance with kDN higher than a threshold. If the instance's RoC is hard, the algorithm proceeds to the local pool generation. If not, a simple kNN labels the instance.

The generation of the local pool follows the steps in Algorithm 6. The size M of the local pool dictates the number of iterations required to build the pool. At each iteration, the size of the local region increases by 2, shown in line 4, selected by the K-Nearest Neighbor Equality (kNNE) algorithm.

Proposed by Sierra et al. (2011), kNNE improve the kNN by giving equal chances for all classes. In more detail, the algorithm takes the average distance between the test instance and every k nearest neighbors of each class. Then, the class with the smaller average distance is the label.

Going back to the local region, to build a pool of classifiers, OLP uses the SGH, creating a pool of local oracles. The classifiers in this pool are evaluated in the training dataset using OLA and added to the global classifiers pool.

Algorithm 6 Online Local Pool algorithm

```
 1: procedure $\mathrm{OLP}(x_q, \Gamma, C, k)$
Require: Query pattern $x_q$
Require: Training set $\Gamma = \{x_1, x_2, ..., x_N\}$
Require: Neighborhood size $k$
Require: Local pool size $M$
 2:     $C' = \{\}$                                   ▷ The pool of competent classifiers
 3:     for $m \in 1, 2, ..., M$ do
 4:         $k_m \leftarrow k + 2 \times (m - 1)$
 5:         $\theta_m \leftarrow kNNE(x_q, k_m, \Gamma)$              ▷ Gets the k-nearest neighbors
 6:         $C_m \leftarrow SGH(\theta_m)$
 7:         $c_{m,n} \leftarrow OLA(x_q, \Gamma, C_m, k_m)$
 8:         $C' \leftarrow C' \cup \{c_{m,n}\}$
 9:     end for
10:     return The pool $C'$
11: end procedure
```

Lastly, the meta-learning techniques present a different paradigm from previous techniques. Cruz et al. (2015) approaches the problem of estimating the competence of classifiers by using a meta-learning algorithm. Recalling the MCS architecture of overproduction, selection, and integration, the META-DES divides into three phases, Overproduction, Meta-learning, and Generalization.

The overproduction phase is similar to the previous methods: generating a pool of classifiers. However, the selection method is a classifier trained in the meta-learning phase rather than a simple rule.

Algorithm 7 Meta-learning phase of META-DES, adapted from Cruz et al. (2015)

```
 1: procedure METADES($\Gamma$)
Require: $\Gamma \leftarrow \{x_1, x_2, ..., x_N\}$                          ▷ Training dataset
Require: Pool of classifiers $C = \{c_1, c_2, ...c_M\}$
Require: Neighborhood size $k$
 2:     $\mathcal{T}^* \leftarrow \{\}$
 3:     for all $x_i \in \Gamma$ do
 4:         Compute the consensus of the pool $H(x_i, C)$
 5:         if $H(x_i, C) < h_c$ then
 6:             $\theta_i \leftarrow kNN(x_i, k, \Gamma)$                 ▷ Gets the k-nearest neighbors
 7:             $\tilde{x}_i \leftarrow ouputprofile(x_i, C)$             ▷ Compute the output profile of $x_i$
 8:             $\phi_i \leftarrow findSimilarProfiles(\tilde{x}_i, C, K_p, \Gamma)$   ▷ Finds the $K_p$ similar ouput profiles
 9:             for all $c_j \in C$ do
10:                 $v_{i,j} = metaFeatureExtractor(\theta_i, \phi_i, c_i, x_i)$
11:                 if $c_i$ correctly classifies $x_i$ then
12:                     $\alpha_{i,j} = 1$          ▷ Means that $c_i$ is a competent classifier for $x_i$
13:                 else
14:                     $\alpha_{i,j} = 0$          ▷ Means that $c_i$ is not a competent classifier for $x_i$
15:                 end if
16:                 $\mathcal{T}^* = \mathcal{T}^* \cup (v_{i,j}, \alpha_{i,j})$
17:             end for
18:         end if
19:     end for
20:     Divide $\mathcal{T}^*$ into 25% for validation and 75% for training
21:     Train a classifier $\lambda$ using $v$ as feature space and $\alpha$ as class
22:     return The meta classifier $\lambda$
23: end procedure
```

The Algorithm 7 describes the meta-learning phase. The meta-learning is used in scenarios where the classifier pool has low confidence in its prediction, measured by a confidence score. Thus, to improve the prediction, a meta-classifier is trained using classifiers that had a confidence score lower than a threshold.

Line 10 shows that five meta-instances are extracted for every instance and classifier: the neighbor's hard classification, the posterior probability, the overall local accuracy, the output profiles classification, and the classifier's confidence. This set of features is used in combination with $\alpha_{i,j}$, which labels if the classifier c_j is a competent classifier of the instance x_i.

Finally, in the generalization phase, the meta-features are extracted using the similar output profiles of the instance x_i in the RoC using the DSEL. Then the meta-features are used to select the competent classifiers in the pool, which are combined using majority voting.

2.4 EMBEDDING ALGORITHMS

Embedding algorithms are a class of models that creates mappings between two feature spaces. These mappings are used to reduce dimensionality and improve data complexity. This section will introduce Linear, Nonlinear, and Distance Metric Learning embeddings.

2.4.1 Linear Embeddings

Initially proposed by Pearson (1901), PCA is a helpful technique to reduce data dimensionality and decorrelate features. PCA works by selecting the eigenvectors with higher eigenvalues, effectively selecting the components with the highest variances. In other words, the algorithm finds a lower-dimensional space that minimizes the distance between the data points and the projections.

This algorithm removes linear dependent features by creating new orthogonal dimensions where the original linear dependent features are comprised of a single dimension, used to improve classifier performance on specific applications Bishop and Nasrabadi (2006).

As pointed out by Collins et al. (2001), the original PCA algorithm does not contemplate binary or category features, as the PCA's formulation is to minimize a squared loss function or project data onto a lower dimensionality space while maintaining the variance, which is more suitable to Euclidean variables.

To tackle the previous formulation issue, Tipping (1998) proposes a probabilistic formulation for PCA, where the latent variables could be defined as independent Bernoulli distributions. Similarly, Collins et al. (2001) generalizes for any member of the exponential family of distributions, for example, Poisson, Bernoulli, Normal and Exponential distributions.

However, Landgraf and Lee (2020) points out that the definition of Collins et al. (2001) is more approximate to Single-Value Decompositon (SVD) that o PCA as it aims at a low-rank factorization of the natural parameters matrix. So, the authors propose an algorithm that computes the principal components by using a linear combination of the data.

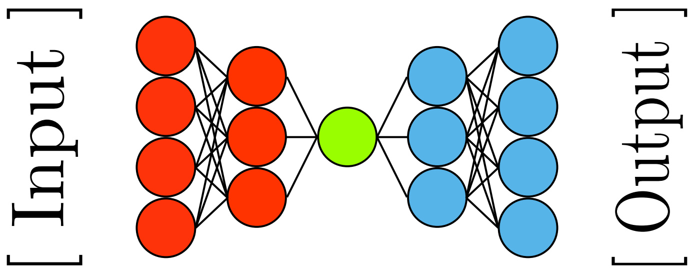


Figure 2 – Architecture of an Autoencoder. In red, the encoder neuros, in green the bottleneck neuron and in blue the decoder neurons.

2.4.2 Nonlinear Embeddings

As defined by Goodfellow, Bengio and Courville (2016), Autoencoders are neural networks with symmetric architecture with a bottleneck that tries to copy their input to the output. The Autoencoder bottleneck is an intermediate layer that compresses data into a smaller dimension. After training, the output of this bottleneck is used as embedding.

The Autoencoder can be seen as a non-linear alternative to PCA. As the bottleneck compresses the data points, the Autoencoder generalizes the input data in a lower dimension. An essential characteristic of autoencoders to Sakurada and Yairi (2014) is the capability of representing structure and patterns in a smaller set of features.

Figure 2 shows the architecture of an autoencoder, where a circle represents each neuron. We can break the network into three parts, the encoder, the bottleneck, and the decoder. The encoder part of the network is represented in red and performs the data compression. The bottleneck is the green neuron and the layer where the data compression is highest. The output of the bottleneck is used as embedding. Finally, the decoder uses the embedded information and tries to reconstruct the original data.

2.4.3 Metric Learning Embeddings

Finally, Metric Learning Embeddings are algorithms that learn how to separate data into a new space. This separation is performed by agglutinating similar instances and separating different instances. However, how do we define similar and different instances?

Measure similarity is an ill-posed problem that varies between problems. For example, the similarity between two images can be measured by using $L1$ distance, which measures the pixel-wise difference between two images, or using more complex metrics such as Structural Similarity

Index (SSIM) proposed by Wang et al. (2004), which measures differences in luminance, contrast, and structure in an image.

Metric Learning embeddings learn similarity or dissimilarity between two instances in a dataset by approximating instances of the same class and separating instances of different classes in the embedding space. As observed in the survey of Kaya and Bilge (2019), classifiers such as kNN, SVM, and Naïve Bayes do not perform any transformation to the dataset, which can be represented in a better way. This transformation is performed by linear approximators, such as PCA and non-linear approximators, such as neural networks. Non-linear approximators are extremely useful for combining features, extracting patterns in data, and projecting that into a lower-dimensional space.

The Triplet Loss function, introduced by Weinberger and Saul (2009), is used to train neural networks which use the data label to create an embedding space. The network uses the original features as input and feeds through a bottleneck, which reduces the number of dimensions. Unlike an Autoencoder, the network trained with the Triplet Loss learns how to separate the classes in the embedded space better than copying the input in the output.

Equation 2.6 mathematically describes the Triplet Loss function. The variables A, P, and N represent an anchor, positive and negative instances extracted from the training set. The positive point has the same label as the anchor, while the negative anchor does not. The neural network is represented by f, and α is the margin value between classes.

This loss function works by maximizing the distance between A and N while minimizing the distance between A and P. The variable α is a tunable hyperparameter that penalizes the network if the margin between classes is below this threshold.

$$L(A, P, N) = ||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha \tag{2.6}$$

The margin α is fundamental to avoiding negative instances and not invading the neighborhood of positive instances. This definition is similar to the SVM margin, as pointed by Weinberger and Saul (2009). Kaya and Bilge (2019) points out that margin can help find hard, semi-hard, or easy groups of instances.

Kaya and Bilge (2019) defines hard instances are negative samples that lie at a distance smaller than an anchor and the closest positive instance. Semi-hard instances are negative instances that lie within the range of the closest positive sample and the margin. In contrast, easy instances are negative instances located further than the distance between the anchor and the closest positive sample, plus the margin.

Selecting the instances during training is also essential. Wu et al. (2017) shows that while selecting hard instances during training yields noisy training, and training progress halts if negative samples are uniformly distributed and are randomly selected, a distance weighting sampling improves the performance of trained networks.

3 ASSESSING THE IMPACT OF BINARY FEATURES ON MULTIPLE CLASSIFIER SYSTEMS

Our objective in this chapter is to discuss the impacts of datasets with binary features on DS algorithms.

3.1 HAMMING SPACE AND MULTIPLE CLASSIFIER SYSTEMS

We discussed in Section 2.2 the applicability of Hamming features by reducing complexity in neighborhood estimation in high dimensionality spaces. However, despite the advantages of a more efficient space for comparisons between instances, this binary space yields multiple instances sharing the same Hamming distance. Zhang et al. (2013) defines this as ambiguity. This issue is reported in the literature of high-dimensionality search by Gui et al. (2021) and Zhang et al. (2013).

As stated by Gui et al. (2021), the Hamming space presents challenges to estimating the neighborhood of an instance. This issue arises from the binary characteristic of the features, which results in data points sharing the same Hamming distance. Zhang et al. (2013) state that binary features negatively impact kNN due to the number of instances that can share the same distance and data sparsity.

Materializing the ambiguity problem, let us assume a binary string of size 4, if we choose the instance $h_0 = 1010$, we can create a set s of 4 other instances that are equidistant from h_0, with $d = 1$: $s = \{0010, 1110, 1000, 1011\}$.

We can generalize the number of different data points sharing the same Hamming distance as the following: to create all the strings with distance d to h_0, we select d digits from h_0 to flip and permute the positions of which bits from h_0 are going to be flipped. Figure 3 shows all the possible unique binary codes considering $h_0 = 1010$. Also, note that permutating the different or equal positions leads to repeated strings. Thus, we can remove the repetitions in the permutation without losing unique strings.

Thus, the number C of binary codes where its Hamming distance to h is d is defined by Equation 3.1, where the instance h has n features.

$$C = \frac{n!}{(n-d)!d!} \tag{3.1}$$

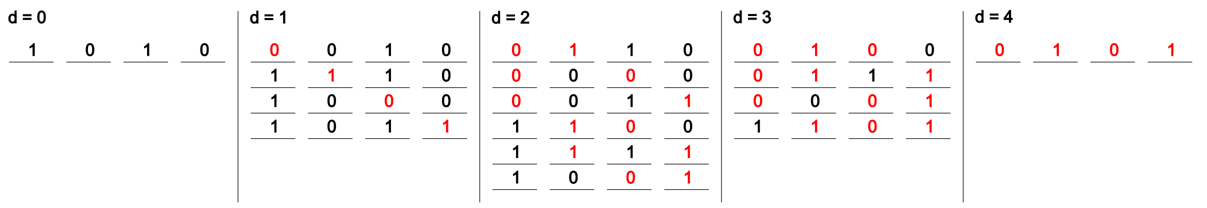


Figure 3 – Permutation of different digits in a binary string. The different digits are highlighted in red. d represents the Hamming difference between the strings and $h_0 = 1010$, or, the number of different digits in the string compared to h_0.

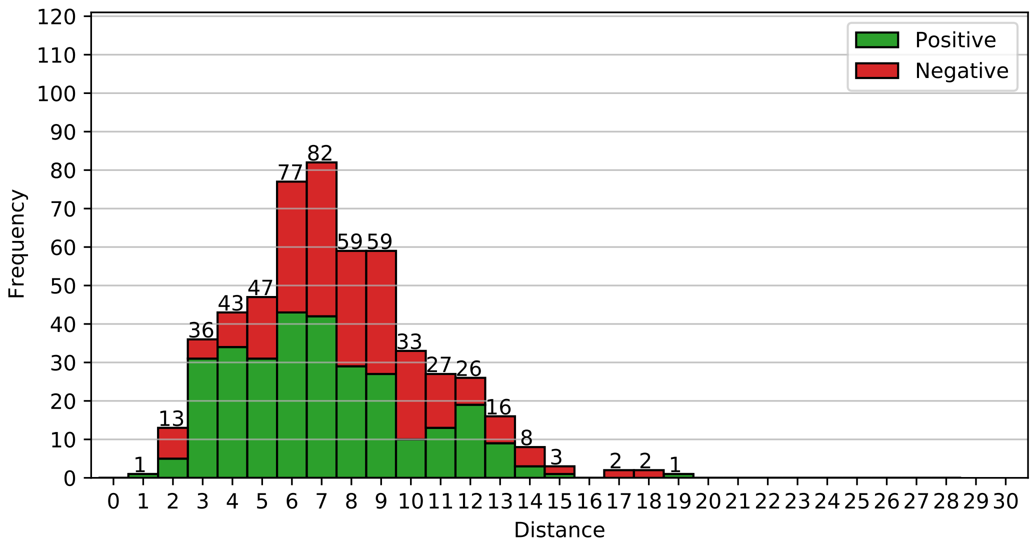


Figure 4 – Example of ambiguous situation, x-axis is the distance between samples and y-axis the number of instances in each distance bin. Red and green are negative and positive samples compared to an anchor.

As we showed in Equation 3.1, the number of codes sharing the same Hamming distance increases as the code size increases, far exceeding the k nearest neighbors, usually 5 or 7.

A practical example of this ambiguity is displayed in the histogram in Figure 4. Here, we select an anchor instance from the binary dataset, used to calculate the Hamming distance between it and the rest. Then, the number of samples with the same label as the anchor, positive samples, are displayed in green and red. The negative samples have different labels compared to the anchor. Here, the instances selected by the kNN play a significant role in classifying the sample correctly, as the majority of the samples are equally distant to the anchor.

The ambiguity is related to the binary disposition of the features associated with kNN. When analyzing DS, we observe that the majority of the techniques use neighborhood algorithms to define the RoC of an instance, and as stated by Cruz, Sabourin and Cavalcanti

(2018), the distribution of the RoC impact DS performance. However, can datasets containing purely binary features impact DS?

3.1.1 Binary Features and Online Local Pool Algorithm

Let us take as an example the OLP, Souza et al. (2019). The algorithm decides which method will be used to classify an instance based on the instance kDN. If the instance hardness of an instance is high enough, a pool of classifiers is generated with SGH. Otherwise, kNN performs the classification.

We must observe the impact of binary features on OLP, first, during the decision process of how to choose between the SGH pool and kNN classification, and second, how the features impact the pool generation process.

Before we analyze these topics, we must recall how the OLP algorithm works. The OLP uses an instance hardness threshold IH_{th} to decide if the neighbors of the instance are hard enough to require a pool of classifiers to label an instance. This threshold is compared to the kDN score of the test instance over all other instances in the RoC.

As we discussed in Section 2.3.4.3, in the OLP, if the instance hardness is high enough, the RoC is selected, and a pool of classifiers is trained using the SGH algorithm. SGH, introduced by Souza et al. (2017) is a pool generation algorithm that iteratively creates hyperplanes until each instance is classified correctly by at least one classifier.

Although kDN is well suited to decide if an instance is hard enough in R^n space when applied to a dataset of binary features, using kDN can lead to issues when generating a pool of classifiers with SGH. As the Hamming distance evaluates how many bits are the difference between two binary strings, and the kDN measures the ratio of mismatching labels in a region, an instance can be both hard by the kDN score and linearly separable. Thus, when the SGH generates a pool, this pool will contain only one classifier, making kDN inadequate to measure how difficult the RoC of an instance is.

The example in Figure 5 shows that the maximum number of unique 2-dimensional instances with Hamming distance of 2 to a test instance is 1.

Figure 5 shows an example of a hard instance by kDN while being linearly separable. The example shows a RoC with 5 instances, 3 red triangles placed on $(0,0)$ and 2 blue x-crosses placed at $(1,1)$. The kDN is the maximum for a RoC with $k=5$, and at the same time, the classes are linearly separable by the classifier C. This is also true for other combinations for

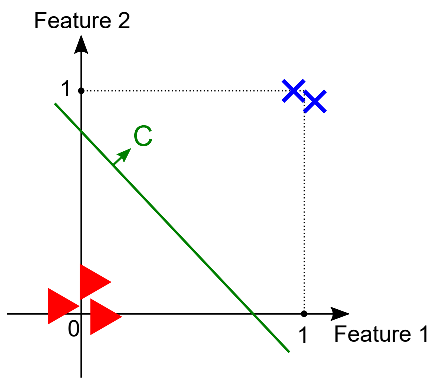


Figure 5 – Linear separation of five hard instances.

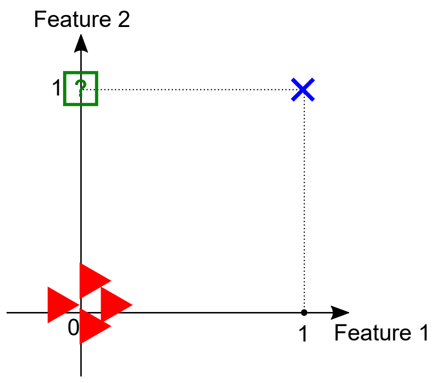


Figure 6 – Linear separation of five easy instances.

the features 1 and 2, like $(0, 0)$ and $(1, 0)$ or $(0, 0)$ and $(1, 0)$.

Figure 6 shows the case where the kDN is smaller than a certain threshold, considering the instance as easy. In this case, the unknown green instance would be classified as a red triangle using the kNN algorithm.

In these cases, kNN uses a heuristic to select the instances to classify the unknown point, which depends on the algorithm's implementation. For example, (PEDREGOSA et al., 2011) uses the instances' order in the dataset and selects the first instances. This issue can also be extended from classification to the definition of the RoC for DS. As the RoC is the neighborhood of a test sample related to training samples, it equally suffers from this ambiguity.

4 METHODOLOGY

Malware Classification and SRM Classification relies on creating a binary dataset of features, which is a new characteristic in DS literature. Thus, to understand how these algorithms perform under these circumstances, we conduct experiments to compare MCS performance related to monolithic classifiers and how preprocessing these binary features with embedding algorithms impacts the performance of such classifiers. In this section, we present the methodology used during our experiments.

4.1 OUR APPROACH

4.1.1 The Dataset

Our experiments have focused on SRM classification. To be precise, we use the extended feature set proposed by Rasthofer, Arzt and Bodden (2014), which contains 213 static binary features extracted from Android API methods. This extended feature set consists of Semantic, Syntactic, and Data Flow features and includes Android Permissions required for the method. The authors evaluate both feature sets, but only the feature set without permissions is available in the repository[1]. The dataset provided by the authors contains 144 features and 793 instances.

We executed code provided by Rasthofer, Arzt and Bodden (2014) with the same API version, but we could not replicate the same amount of dataset instances. To increase the number of data, we merged several APIs, available here[2], removed instances with missing features and duplicated instances, resulting in 640 instances.

4.1.2 Evaluation Procedure

As we discussed in Section 2.1, differently from Malware Classification, in SRM Classification, we observe that only Sas, Bessi and Arcelli Fontana (2018) analyzes ensemble learning.

Thus, to fill the gap of knowledge for these classifiers, we evaluate monolithic, static, and dynamic ensembles of classifiers in SRM dataset consisting of pure binary features. This evaluation intends to create a baseline for all the models using the binary feature space. Then,

1 https://github.com/secure-software-engineering/SuSi
2 https://github.com/walbermr/SuSi/tree/develop/android-platforms

Decision Tree	max height = unlimited	criterion = Gini Impurity
Naive Bayes	algorithm: Bernoulli	
KNN	$k = 7$	
SVM	kernel = linear	$C = 1.0$
MLP	hidden layers size = (100)	max iterations: 1000
Random Forest	pool size = 100	max height = unlimited
Gradient BoostedDecision Trees	max pool size = 100	

Table 3 – Hyperparameters used in monolithic classifiers.

Single Best	score function = accuracy			
Static Selection	percentage to select = 0.5	score function = accuracy		
OLA	$k = 7$	DSEL size = 0.5		
KNORA-U	$k = 7$	DSEL size = 0.5		
KNORA-E	$k = 7$	DSEL size = 0.5		
META-DES	$k = 7$	DSEL size = 0.5	output profiles = 5	selection threshold = 0.5

Table 4 – Hyperparameters used for MCS. k represents the size of the RoC, DSEL size is the percentage of data used to train the selection algorithm.

we also include Embedding Algorithms such as PCA, Logistic PCA, Autoencoder, and Metric Learning using Triplet Loss.

Table 3 shows the hyperparameter used for each monolithic classifier. Other hyperparameters are set as the default of *scikit-learn* Pedregosa et al. (2011).

To perform the comparisons between MCS and monolithic classifiers, we included Single Best, Static Selection, OLA, KNORA-U, KNORA-E, META-DES, OLP, and the Oracle model, using the standard hyperparameters of DESLib, Cruz et al. (2020). The hyperparameters are presented in Table 4, aditionally, for OLA, KNORA-U, KNORA-E, and META-DES, if the instance hardness of the test instance is lower than 0.3, the kNN is used as classifier, if it is higher, the pool of classifiers is used.

The algorithms to train the pool of classifiers are the Bagging, Boosting, and Random Forest. For the Bagging, each generated bootstrap had the size of the dataset, generating 100 classifiers in the pool. The boosting algorithm used is the AdaBoost, using the default hyperparameters, like the Random Forest, which also uses 100 classifiers. The maximum size of the Decision Tree is set to unlimited, meaning that the tree can increase its height until it fits every data point.

In our tests, we included two types of base classifiers to train the pool of classifiers used by the MCS, Decision Tree, and Perceptron, using the standard hyperparameters of the *scikit-learn* library Pedregosa et al. (2011). This procedure is described by Cruz et al. (2015), where a pool of weak classifiers are used in the pool, as a weak classifier generates more diversity when the dataset slightly changes.

Now, we will discuss the embedding algorithms hyperparameters. The autoencoder architecture with three fully-connected layers, the first is a 213 input to emb_dimm output, followed by a layer with emb_dimm input and emb_dimm output, the last layer a emb_dimm input to 213 output, where emb_dimm is the embedding dimension. The last layer has a sigmoid activation function, and the other layers have a $ReLU$ activation. We use adam as an optimizer and 10^{-3} as the learning rate, and due to the nature of the training data, Binary Cross Entropy (BCE) as loss function, described in Equation 4.1.

$$BCE(x, y) = y_n \cdot log x_n + (1 - y_n) \cdot log(1 - x_n) \tag{4.1}$$

Finally, the network trained with the Triplet Loss also uses a three-layer network, with the first layer 213 input to 100 size output, 100 size input to 100 output, and 100 input to emb_dimm output. The first and second layers use $ReLU$ activation, and the last layer has no activation. The training loss uses the Triplet Loss, described in Section 2.4.3, with a $alpha = 1.0$. We use adam as an optimizer and 10^{-3} as the learning rate with a decay of 10% for every 8 epochs.

Regarding the instance sampling method, we used random sampling. As discussed in Section 2.4.3, Wu et al. (2017) points out that random sampling could negatively impact the training architecture. However, random sampling is enough to analyze its impact on the problem to draw a baseline for metric learning algorithms' performance on binary data.

To adhere to the experimental procedure defined in Benavoli, Corani and Mangili (2016), we perform a stratified hold-out repeated 30 times. Raschka (2018) presents this procedure as a method to evaluate model stability. To statistically verify the results, we use the Kruskal-Wallis test, proposed by Kruskal and Wallis (1952) to evaluate if exists a statistically significant difference between a pair of models. This methodology is used by Cruz et al. (2015) to compare pairs of classifiers on the same dataset using a hold-out repeated 20 times per dataset.

As Rasthofer, Arzt and Bodden (2014), we use Precision and Recall to evaluate the classifiers, and we add F1 and Accuracy. Precision is the ratio between the correctly predicted instances of a class and the total class predictions, and Recall is the ratio between correctly predicted instances of a class and all the instances of this class. F1 combines Precision and Recall giving a more general intuition of the performance of the classifiers, and accuracy measures how many samples are correctly classified. Precision, recall, F1, and accuracy are described in Equation 4.1.2, where TP, FP, TN, and FN, stand for true positive, false positive, true negative, and false negative, respectively.

$$precision = \frac{TP}{TP + FP} \quad (4.2)$$

$$F1 = \frac{2 \times recall \times precision}{recall + precision} \quad (4.4)$$

$$recall = \frac{TP}{TP + FN} \quad (4.3)$$

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (4.5)$$

4.2 EXPERIMENTAL PROCEDURE

We conducted the experiments using a stratified hold-out of 30 partitions of the dataset. Each partition has a proportion of 80% to train and 20% to test. To train the DSEL algorithm for dynamic ensemble algorithms, we further divide the training set into 50% to train the pool of classifiers and 50% to train the selection algorithm.

The first experiment, Section 5.1.1, is relative to classifier comparison in order to fill the SRM literature gap related to the lack of an evaluation of dynamic ensemble algorithms. The experiments directly compare monolithic, static, and dynamic ensembles of classifiers.

In the second experiment, Section 5.1.3, we apply preprocessing algorithms to the binary features. This experiment has an objective to understand how much the set of binary features can be improved and how embedding algorithms can improve MCS techniques. This experiment is conducted with an ablation study, changing the number of dimensions created by the embeddings and evaluating the performance of DES techniques in this new feature space.

5 EXPERIMENTS

This chapter describes the experiments performed and their parameters and then analyzes their results. Each experiment intends to answer a specific question and fill gaps in the literature relative to MCS using binary features.

5.1 RESULTS AND EXPERIMENTS

5.1.1 First Experiment

In Table 5 we present the results for monolithic classifiers and observe that MLP has the best result for all the metrics. When we compare the results reported by Rasthofer, Arzt and Bodden (2014), we can observe that both MLP, Random Forest, and Gradient Boosted Decision Trees classifiers all outperform SVM. The Kruskall-Wallis test shows that exists significant difference between SVM and Random Forest ($p-value = 0.002$), Gradient Boosted Decision Trees ($p - value = 0.002$), and with MLP (0.017), considering a 95% confidence interval.

Our first evaluation for MCS uses the method described by Cruz et al. (2015) to train the pool of classifiers, where the pool of base classifiers consists of 100 Perceptrons trained using the Bagging technique. It is demonstrated in the literature that weak classifiers achieve

Model	Precision	Recall	F1 Score	Accuracy
Decision Tree	0.8446(0.0309)	0.8418(0.0307)	0.8413(0.0311)	0.8418(0.0307)
Naive Bayes	0.8234(0.0308)	0.8219(0.0310)	0.8208(0.0309)	0.8219(0.0310)
kNN	0.8256(0.0267)	0.8050(0.0247)	0.7939(0.0267)	0.8050(0.0247)
SVM	0.8676(0.0281)	0.8667(0.0282)	0.8659(0.0285)	0.8667(0.0282)
MLP	0.8851(0.0304)	0.8838(0.0306)	0.8832(0.0312)	0.8838(0.0306)
Random Forest	0.8947(0.0250)	0.8915(0.0259)	0.8903(0.0267)	0.8915(0.0259)
Gradient Boosted Decision Trees	0.8916(0.0225)	0.8903(0.0224)	0.8899(0.0226)	0.8903(0.0224)

Table 5 – Results for monolithic, where red indicates the best model, green the second-best model, and blue the third-best.

Model	Precision	Recall	F1 Score	Accuracy
Single Best	0.8157(0.0316)	0.8127(0.0300)	0.8099(0.0317)	0.8127(0.0300)
Static Selection	0.8438(0.0310)	0.8410(0.0312)	0.8386(0.0318)	0.8410(0.0312)
OLA	0.8252(0.0325)	0.8229(0.0329)	0.8213(0.0339)	0.8229(0.0329)
KNORA-U	0.8447(0.0302)	0.8420(0.0307)	0.8399(0.0312)	0.8420(0.0307)
KNORA-E	0.8627(0.0293)	0.8607(0.0295)	0.8596(0.0304)	0.8607(0.0295)
META-DES	0.8638(0.0295)	0.8607(0.0300)	0.8584(0.0308)	0.8607(0.0300)
OLP	0.7244(0.0434)	0.6828(0.0509)	0.6794(0.0538)	0.6828(0.0509)
Oracle	0.9782(0.0138)	0.9776(0.0144)	0.9775(0.0144)	0.9776(0.0144)

Table 6 – Results for MCS Bagging Perceptron, where red indicates the best model, green the second-best model, and blue the third-best.

better results for DES techniques create variability in the pool, and Bagging helps stabilize weak classifiers during training Cruz et al. (2015).

So, we compared MCS algorithms using a pool of 100 Perceptrons trained using Bagging. Table 6 shows that META-DES and KNORA-E are the two best classifiers, the Kruskal-Wallis test shows that there is no difference between META-DES and KNORA-E ($p-value = 0.970$), but the difference between META-DES and KNORA-U ($p - value = 0.023$) is significant.

We can also observe that the OLP is outperformed by META-DES, this difference is statistically relevant ($p - value = 3.041 \times 10^{-11}$), although Souza et al. (2019) shows that OLP and META-DES should have similar performance. Our analysis in Section 3.1 shows that the intrinsic ambiguity of the Hamming space impacts kNN-based algorithms and, in particular, the OLP.

We can further explore MCS models by using a weak Decision Tree classifier. In this case, the Table 7 shows that KNORA-E has better results overall. The Kruskal-Wallis test shows no significant difference between KNORA-E and META-DES ($p - value = 0.847$), and shows significant difference between KNORA-E and OLA ($p - value = 1.192 \times 10^{-3}$).

Table 8 shows the results for MCS algorithms using Perceptron as base classifiers, using Boosting during the overproduction phase. We can observe that META-DES, KNORA-E and KNORA-U are the three best classifiers, but the difference between them is not statistically relevant. This can be observed in the pairwise comparison between KNORA-E with META-DES

Model	Precision	Recall	F1 Score	Accuracy
Single Best	0.7966(0.0344)	0.7933(0.0328)	0.7906(0.0327)	0.7933(0.0328)
Static Selection	0.8240(0.0383)	0.8209(0.0377)	0.8184(0.0379)	0.8209(0.0377)
OLA	0.8356(0.0292)	0.8338(0.0301)	0.8326(0.0306)	0.8338(0.0301)
KNORA-U	0.8293(0.0383)	0.8266(0.0373)	0.8245(0.0379)	0.8266(0.0373)
KNORA-E	0.8639(0.0352)	0.8614(0.0353)	0.8603(0.0356)	0.8614(0.0353)
META-DES	0.8614(0.0295)	0.8600(0.0294)	0.8588(0.0297)	0.8600(0.0294)
Oracle	0.9828(0.0098)	0.9821(0.0105)	0.9821(0.0104)	0.9821(0.0105)

Table 7 – Results for MCS Bagging Decision Tree, where red indicates the best model, green the second-best model, and blue the third-best.

Model	Precision	Recall	F1 Score	Accuracy
Single Best	0.8230(0.0356)	0.8172(0.0346)	0.8138(0.0357)	0.8172(0.0346)
Static Selection	0.7775(0.0413)	0.7540(0.0560)	0.7333(0.0772)	0.7540(0.0560)
OLA	0.8237(0.0371)	0.8214(0.0357)	0.8190(0.0364)	0.8214(0.0357)
KNORA-U	0.8288(0.0365)	0.8234(0.0338)	0.8188(0.0353)	0.8234(0.0338)
KNORA-E	0.8270(0.0365)	0.8254(0.0353)	0.8226(0.0364)	0.8254(0.0353)
META-DES	0.8396(0.0290)	0.8296(0.0285)	0.8238(0.0305)	0.8296(0.0285)
Oracle	0.9988(0.0027)	0.9988(0.0028)	0.9988(0.0028)	0.9988(0.0028)

Table 8 – Results for MCS Boosting Perceptron, where red indicates the best model, green the second-best model, and blue the third-best.

$(p - value = 0.778)$ and with OLA $(p - value = 0.558)$.

A single base classifier is enough to classify the training dataset in some cases. So, the Boosting algorithm stops the pool generation process, creating a pool of 1 classifier when Decision Tree is used as the base classifier. This problem is not an issue for a Perceptron, as its parameters are defined randomly. Meanwhile, the Decision Tree does not have this characteristic, as the training procedure solely depends on the dataset randomness and the classifier hyperparameters. As MCS requires a set of classifiers, the usage of Decision Trees with Boosting to this problem is not feasible.

When using Random Forest as the pool generation algorithm, we observe in Table 9

Model	Precision	Recall	F1 Score	Accuracy
Single Best	0.7810(0.0311)	0.7799(0.0305)	0.7766(0.0309)	0.7799(0.0305)
Static Selection	0.8520(0.0293)	0.8455(0.0305)	0.8413(0.0325)	0.8455(0.0305)
OLA	0.8241(0.0333)	0.8224(0.0323)	0.8210(0.0326)	0.8224(0.0323)
KNORA-U	0.8577(0.0267)	0.8510(0.0275)	0.8472(0.0286)	0.8510(0.0275)
KNORA-E	0.8611(0.0284)	0.8565(0.0308)	0.8544(0.0320)	0.8565(0.0308)
META-DES	0.8858(0.0261)	0.8828(0.0266)	0.8815(0.0271)	0.8828(0.0266)
Oracle	0.9976(0.0039)	0.9975(0.0040)	0.9975(0.0040)	0.9975(0.0040)

Table 9 – Results MCS Random Forest, where red indicates the best model, green the second-best model, and blue the third-best.

that META-DES is the best DES for all metrics. The Kruskal-Wallis pairwise test between META-DES and KNORA-E ($p-value = 1.194 \times 10^{-3}$) and META-DES and KNORA-U ($p-value = 5.277 \times 10^{-5}$), shows that a statistical significance backs these results.

Comparing Table 5 and Table 7, we can observe that META-DES and KNORA-U combined with Random Forest as pool generation over Bagging using Decision Tree as base classifier.

KNORA-U has a worse selection method, all the classifiers that correctly classify at least one instance in the RoC will be selected for aggregation. Thus, KNORA-U is more sensitive to the local performance of the underlying classifiers. So, increasing the variability in the pool of classifiers benefits the algorithm. As META-DES uses a meta-learning approach, it also benefits from the increase in pool variability.

Finally, when comparing all the classifiers, we observe in Table 10 that the monolithic classifiers have, overall, the three best models, with Random Forest outperforming all other techniques. However, for Accuracy, these results are not statistically significant. Comparing Random Forest with MLP ($p-value = 0.431$), Gradient Boosted Trees ($p-value = 0.497$) and META-DES Random Forest ($p-value = 0.666$), considering a $p-value$ of 0.001.

Also, note that, for all pool generation methods, the accuracy of the Oracle model is higher than 97%. Even though (SOUZA et al., 2017) considers this information as too optimistic about being used as an upper bound of MCS, as it is performed globally and the MCS techniques only use local data, this indicates that the pool generation methods are capable of creating competent classifiers.

Model	Precision	Recall	F1 Score	Accuracy
MLP	0.8851(0.0304)	0.8838(0.0306)	0.8832(0.0312)	0.8838(0.0306)
Random Forest	0.8947(0.0250)	0.8915(0.0259)	0.8903(0.0267)	0.8915(0.0259)
Gradient Boosted Decision Trees	0.8916(0.0225)	0.8903(0.0224)	0.8899(0.0226)	0.8903(0.0224)
Bagging Perceptron				
KNORA-U	0.8447(0.0302)	0.8420(0.0307)	0.8399(0.0312)	0.8420(0.0307)
KNORA-E	0.8627(0.0293)	0.8607(0.0295)	0.8596(0.0304)	0.8607(0.0295)
META-DES	0.8638(0.0295)	0.8607(0.0300)	0.8584(0.0308)	0.8607(0.0300)
Decision Tree				
OLA	0.8356(0.0292)	0.8338(0.0301)	0.8326(0.0306)	0.8338(0.0301)
KNORA-E	0.8639(0.0352)	0.8614(0.0353)	0.8603(0.0356)	0.8614(0.0353)
META-DES	0.8614(0.0295)	0.8600(0.0294)	0.8588(0.0297)	0.8600(0.0294)
Boost Perceptron				
KNORA-U	0.8288(0.0365)	0.8234(0.0338)	0.8188(0.0353)	0.8234(0.0338)
KNORA-E	0.8270(0.0365)	0.8254(0.0353)	0.8226(0.0364)	0.8254(0.0353)
META-DES	0.8396(0.0290)	0.8296(0.0285)	0.8238(0.0305)	0.8296(0.0285)
Random Forest				
KNORA-U	0.8577(0.0267)	0.8510(0.0275)	0.8472(0.0286)	0.8510(0.0275)
KNORA-E	0.8611(0.0284)	0.8565(0.0308)	0.8544(0.0320)	0.8565(0.0308)
META-DES	0.8858(0.0261)	0.8828(0.0266)	0.8815(0.0271)	0.8828(0.0266)

Table 10 – Results aggregating all three best classifiers, where red indicates the best model, green the second-best model, and blue the third-best.

5.1.2 Conclusion

From the results presented, we note that rather than previously presented in the literature, there are better alternatives rather than SVM to SRM classification, for instance, monolithic classifiers as Random Forest, Gradient Boosted Decision Trees, MLP, and DES, as META-DES with Random Forest.

Also, we conclude that MCS algorithms are impacted by the SRM Classification dataset. We assert that based on the experiments results with multiple DCS and DES, and what is reported by the MCS literature.

First, we compared Perceptron and Decision Tree as base classifiers using Bagging. The comparisons show that neither Perceptron nor Decision Tree trained using Bagging had satisfactory results.

We ground this on the results of different pools evaluated on DES: Bagging, Boosting, and Random Forest, which had no significant impact on the majority of MCS. Except for combining Random Forest as pool generation and META-DES. This combination results in a performance similar to monolithic classifiers, with no perceived statistical difference. Observing the results of the Oracle model, we can observe that the pool generation algorithms create competent classifiers. However, the only algorithm capable of selecting these classifiers was the META-DES and Random Forest.

The variability of the binary features can explain these results. Compared to a real-valued feature, selecting different instances from the dataset is enough to create classifier diversity, unlikely with binary features, backed by Random Forest's results as pool generation. Also, the meta-features and a meta-estimator in META-DES play a fundamental role in selecting the most competent classifiers. We can conclude that by evaluating the pool of classifiers in the RoC, either by their accuracy or finding the local oracles, such as OLA, KNORA-E, and KNORA-U, in a dataset of binary features, is not enough to select the most competent classifiers.

5.1.3 Second Experiment

Keeping in mind the ambiguity issues presented in Section 3, in this section, we aim to explore the impact of binary features on the classifiers. Here, we evaluate the performance of monolithic classifiers and MCS in the embedded space created by 4 embedding algorithms, PCA, Logistic PCA, Autoencoder, and Triplet Loss.

The embedding algorithms are then used to transform the binary feature space into continuous features. We use the embedded features to analyze if the sparsity of the hamming features impacts the algorithms. We use the classifiers' accuracy measured on the binary feature set and compare it with the accuracy when trained with each embedding.

The analysis is performed by conducting an ablation study, evaluating how each embedding algorithm performs when presented with a reduced training set and varying the number of dimensions. We vary the training dataset size from 80%, down to 20% in steps of 10% and use 2, 3, 5, 10, 15, 20, 25, 50, 100, 150, and 200 as embedding dimensions.

As PCA decorrelate the features, we first calculated the rank of the features before the embedding. While performing the ranking, we found that, of the 213 features, 178 are linearly independent, consequently uncorrelated. This number of linear independent features suggests that the classifiers will perform worse until the embedding created by PCA reaches 178 features. However, the ranking does not project which features contribute the most to classification.

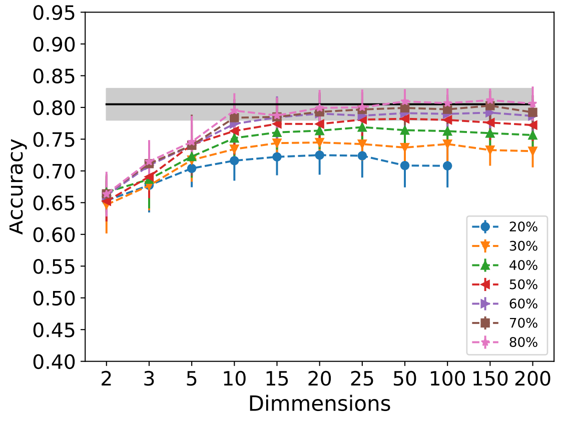
5.1.3.1 Monolithic

First, let us discuss the results for kNN, MLP, Random Forest and SVM using PCA as embedding algorithm, shown in Figure 7. We can observe that using PCA results in no significant improvement over the original feature set. Also, as PCA removes the linear dependency from data, increasing the number of dimensions improves the results until a certain point. Despite the 178 linear independent features, we can observe for the 4 models that 100 dimensions are enough to represent the original dataset.

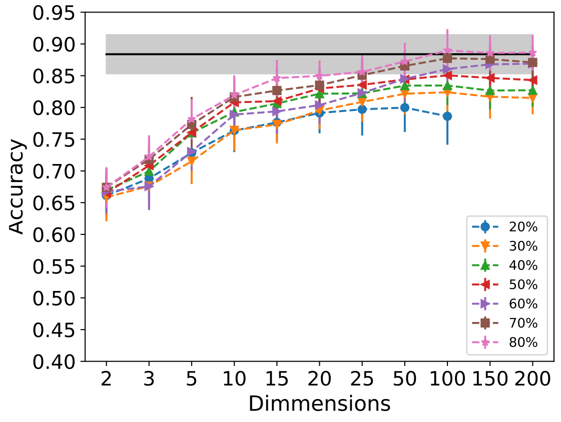
The Logistic PCA results are presented in Figure 8. Similarly to PCA's results, we do not observe a clear improvement over the original feature set. However, compared to PCA, Logistic PCA has a slightly improvement on low-dimension embeddings. However, we can observe degradation in the SVM's performance for embeddings with more than 25 dimensions.

Furthermore, note that the results for MLP are more sensitive to the embedding created by the Logistic PCA algorithm. We note either an improvement or a reduction in the classifier's accuracy at lower dimensions. However, we could not observe a pattern in these results. For example, the proportions of 30% and 60% as training perform worse than 20% and 40%.

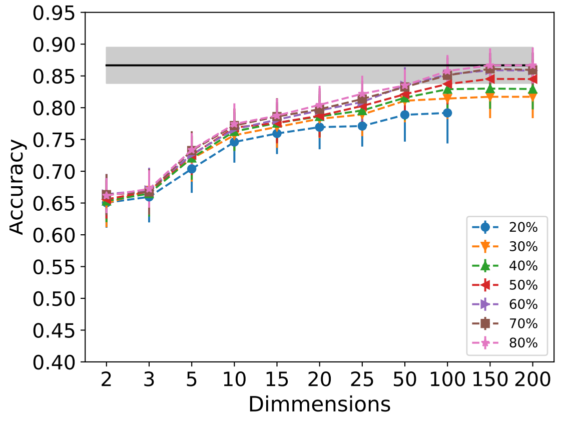
On the other hand, the Autoencoder proved to be worse Figure 9 shows that for every dimension and dataset size, it always worse than using the binary features.



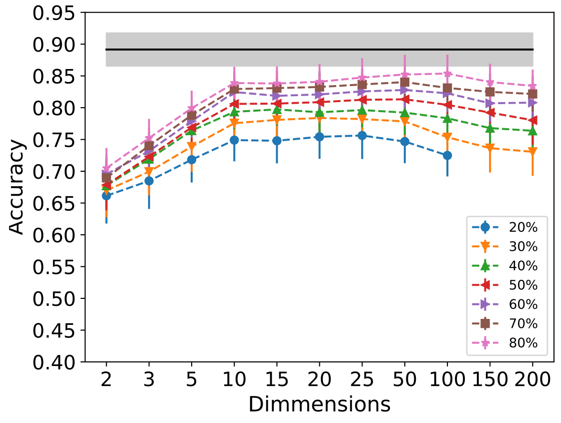
(a) kNN



(b) MLP



(c) SVM



(d) Random Forest

Figure 7 – Accuracy for kNN (Figure 7a), MLP (Figure 7b), SVM (Figure 7c) and Random Forest (Figure 7d) using PCA. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.

It is also noticeable a decrease in the MLP performance when the Autoencoder uses 30% of the instances as training. The accuracy is heavily impacted, especially at 100 dimensions. However, these results are specific to the MLP. Thus, we consider this a training instability for this particular case.

As for Triplet Loss, Figure 10 shows that the embedding reduces the performance for MLP and Random Forest, Figure 10b and Figure 10d. Meanwhile, the kNN results shows a gain in accuracy, Figure 10a.

The observed results are better than the baseline binary features, black line, it is also better than the PCA, Figure 7a and Autoencoder, Figure 9a. Using Triplet Loss, the kNN improved its performance for all dimensions, varying from 50% to 80% of the dataset used as training.

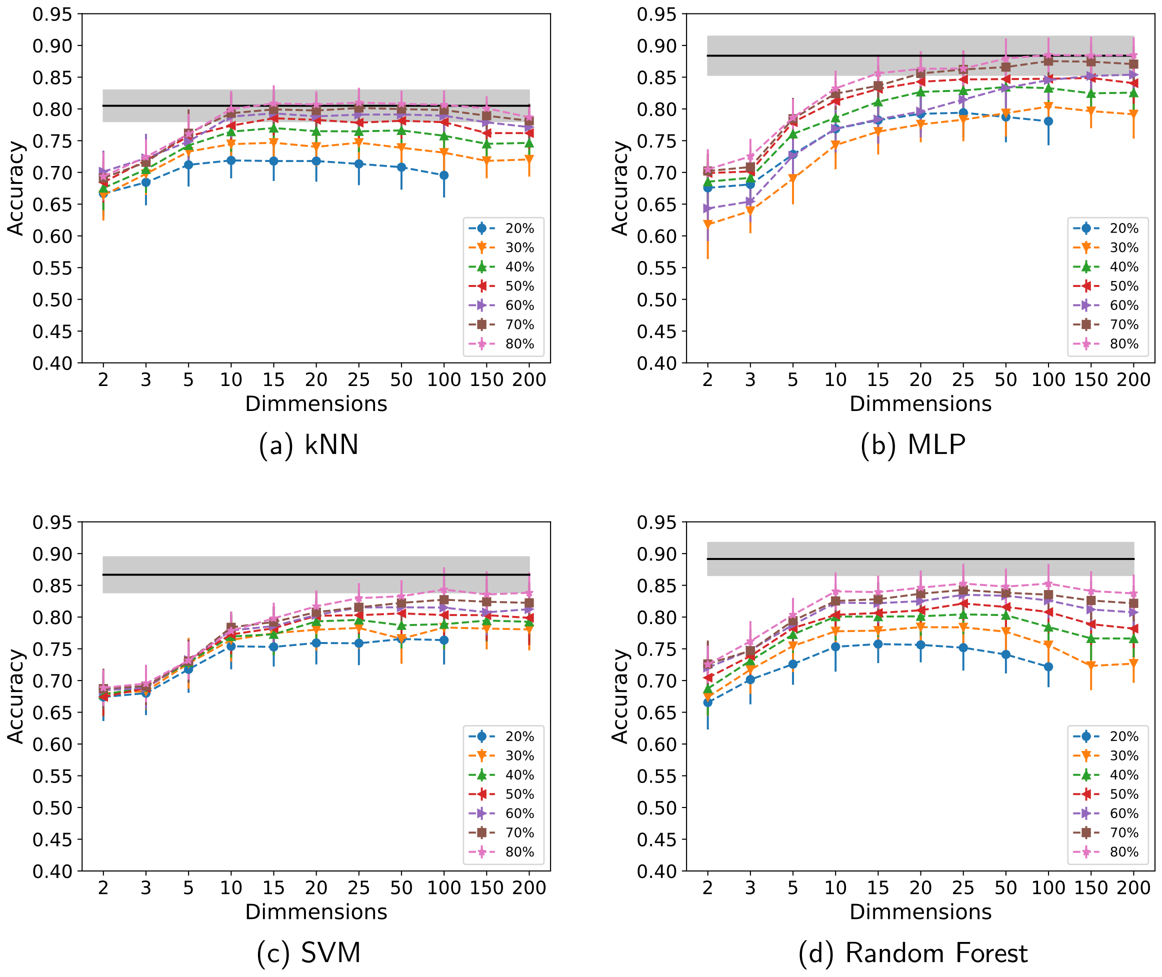
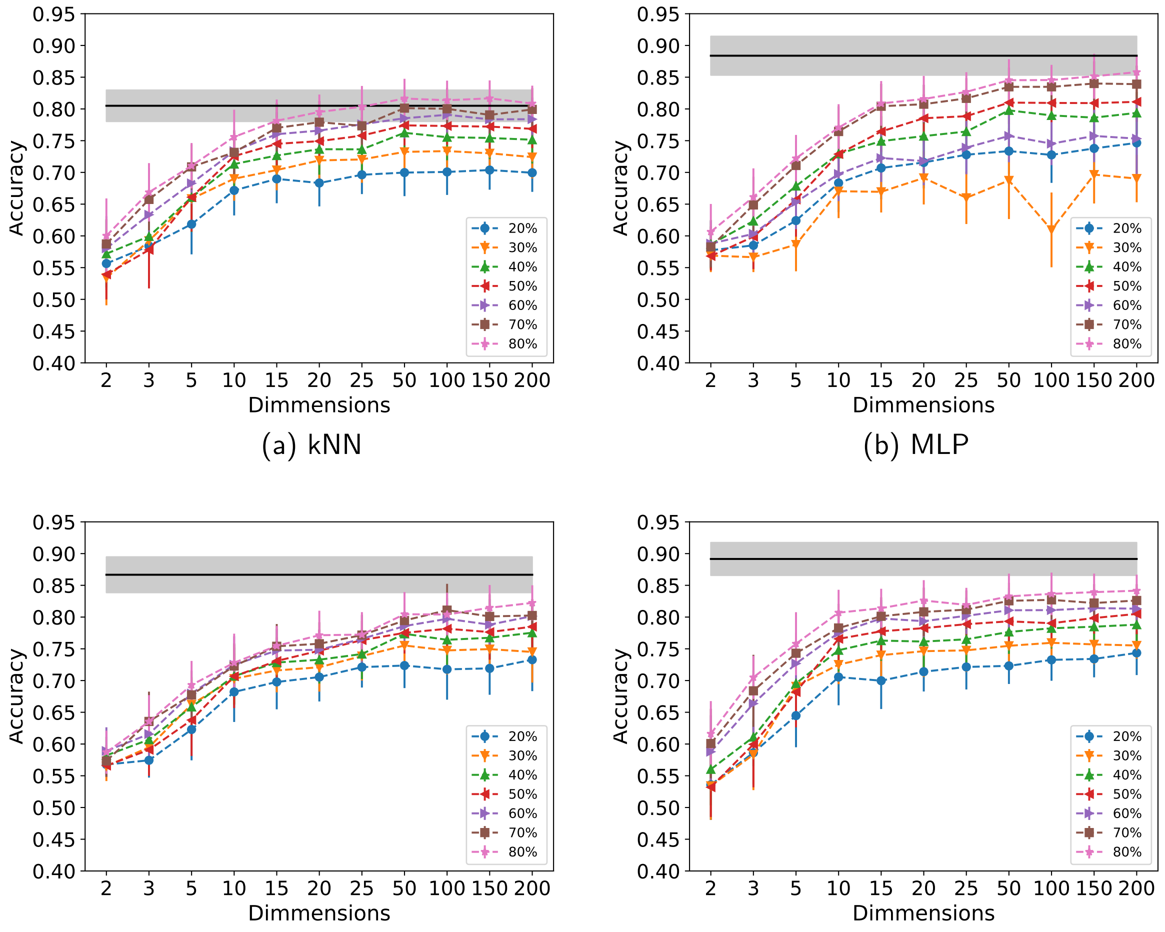


Figure 8 – Accuracy for kNN (Figure 8a), MLP (Figure 8b), SVM (Figure 8c) and Random Forest (Figure 8d) using Logistic PCA. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.

We also observe that the number of dimensions generated by the Triplet Loss algorithm does not impact the accuracy. The accuracy difference between embedding dimensions can be assigned to random initialization of the network.

5.1.3.2 Dynamic Ensemble Selection

Now, we focus our analysis on DES, comparing KNORA-E, KNORA-U, META-DES, and OLP using Bagging as pool generation algorithm, considering Perceptron and Decision Tree as base classifiers. The embedded space is used to train the base classifiers and the MCS algorithms.

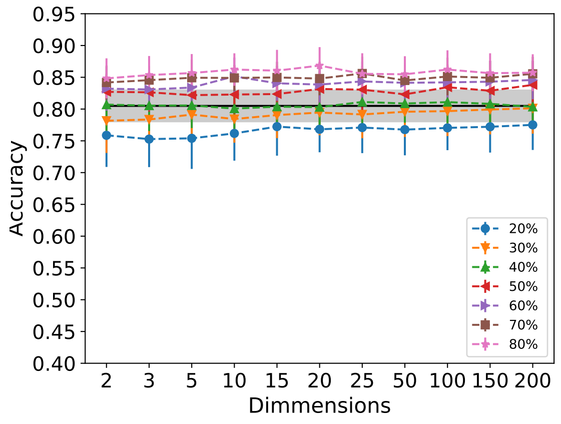


(a) kNN (b) MLP (c) SVM (d) Random Forest

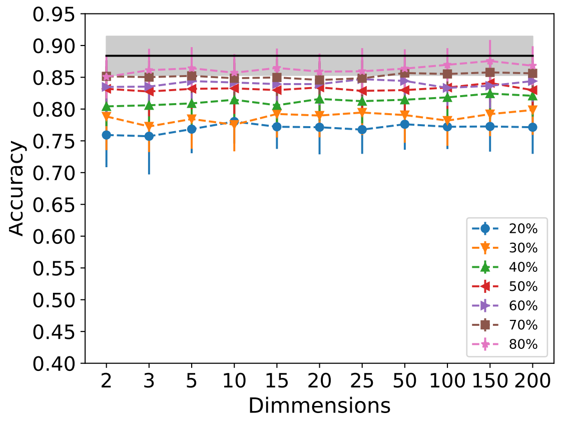
Figure 9 – Accuracy for kNN (Figure 9a), MLP (Figure 9b), SVM (Figure 9c) and Random Forest (Figure 9d) using Autoencoder. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.

The results for DES using Perceptron as base classifier with PCA are displayed in Figure 11, and the results for DES using Decision Tree as base classifier with PCA are displayed in Figure 12. We can observe that KNORA-E, KNORA-U, and META-DES have resulted in line with monolithic classifiers, where the accuracy keeps increasing until the embedding reaches 100 dimensions. Moreover, OLP presents a small increase in accuracy with more than 100 dimensions.

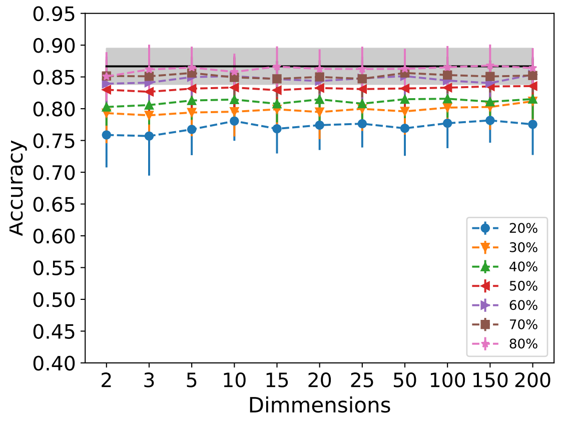
The results for Logistic PCA for DES using Baggin Perceptron and OLP, and Decisionn Tree are presented in Figure 13 and Figure 16, respectively. Here, the results are similar to the PCA's, where the performance increases with the number of dimmensions, and reaches a peak on 100 dimmensions.



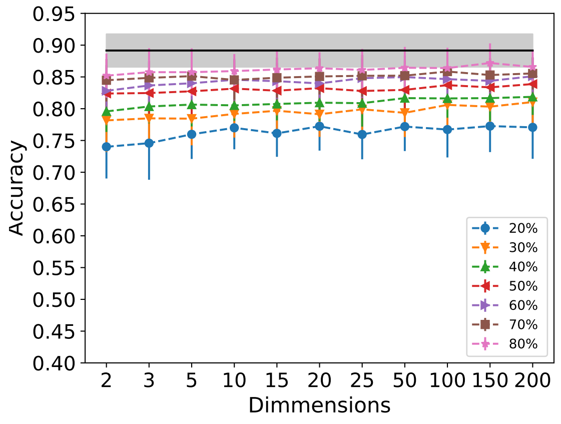
(a) kNN



(b) MLP



(c) SVM



(d) Random Forest

Figure 10 – Accuracy for kNN (Figure 10a), MLP (Figure 10b), SVM (Figure 10c) and Random Forest (Figure 10d) using Triplet Loss. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.

Using Autoencoder to generate the embedding is shown in Figure 15, for DES with Perceptron as base classifier and Figure 16, for Decision Tree as base classifier. There is also a lower accuracy than the baseline with binary features in these cases. Except for OLP, there is an increase in accuracy when using Autoencoder with more than 15 dimensions.

Evaluating the results obtained for Triplet Loss, Figure 17 shows the results for DES with Perceptron as base classifier and Figure 18 the results for DES with Decision Tree, we do not observe the same improvements as the ones for kNN.

Despite the KNORA-E and KNORA-U using kNN to define the RoC, the kNN's ambiguity on Hamming space do not have a big impact on the final classifier performance. Note that, in Figure 17a and Figure 18a, KNORA-U has a small accuracy improvement when using the

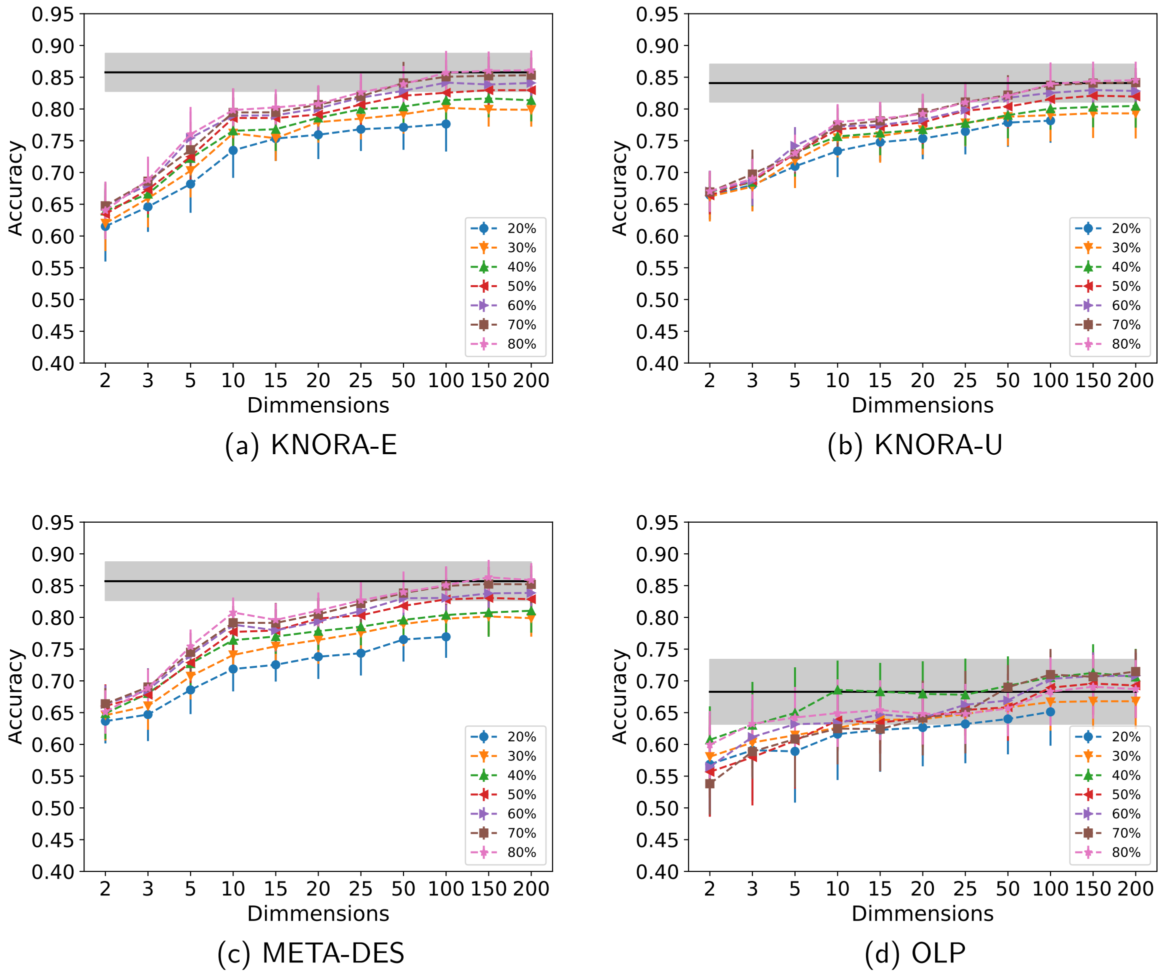
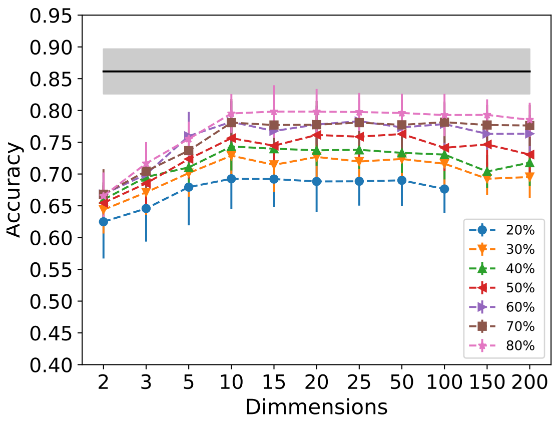


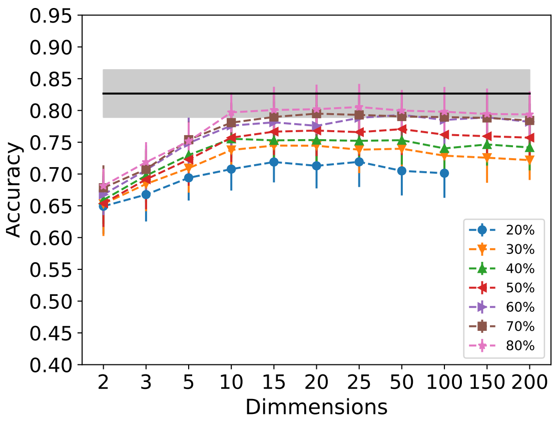
Figure 11 – Accuracy for KNORA-E (Figure 11a), KNORA-U (Figure 11b) and META-DES (Figure 11c) using Bagging and Perceptron, and OLP using SGH (Figure 11d), using PCA as embedding. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.

Triplet Loss embedding with 80% and 70% of the dataset as training.

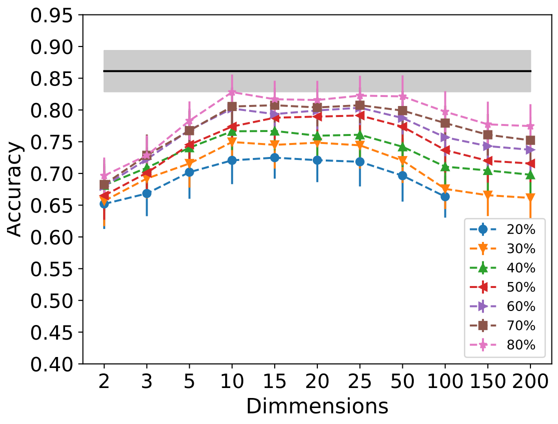
However, we observe a significant impact in OLP when paired with a Triplet Loss embedding, which consistently improves the classification accuracy using a training set higher than 20% for all dimensions. This improvement can be explained by the usage of kNN as the classifier when the instance is not hard enough.



(a) KNORA-E

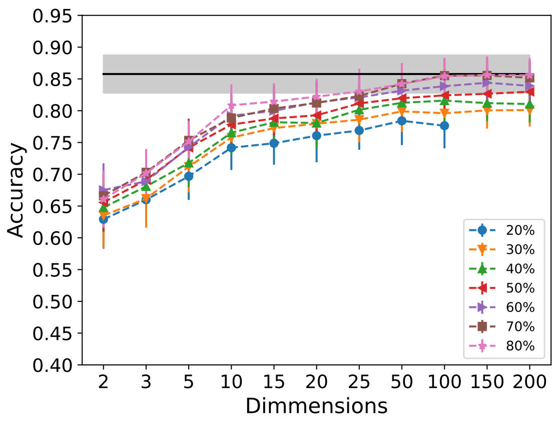


(b) KNORA-U

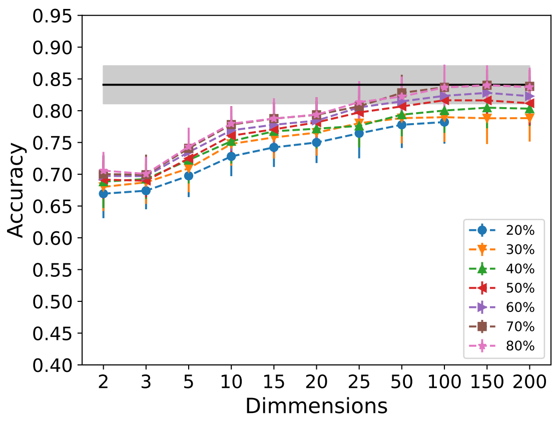


(c) META-DES

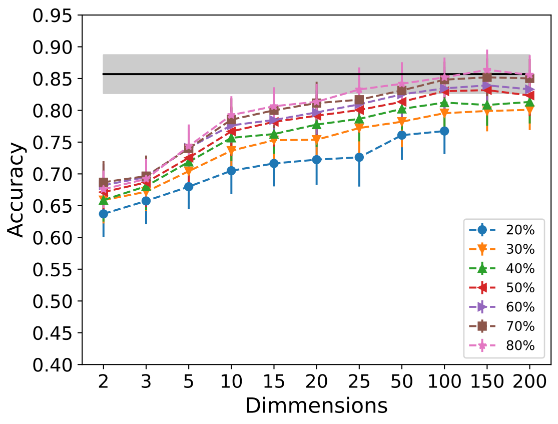
Figure 12 – Accuracy for KNORA-E (Figure 12a), KNORA-U (Figure 12a) and META-DES (Figure 12a) using PCA and Decision Tree as base classifier. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.



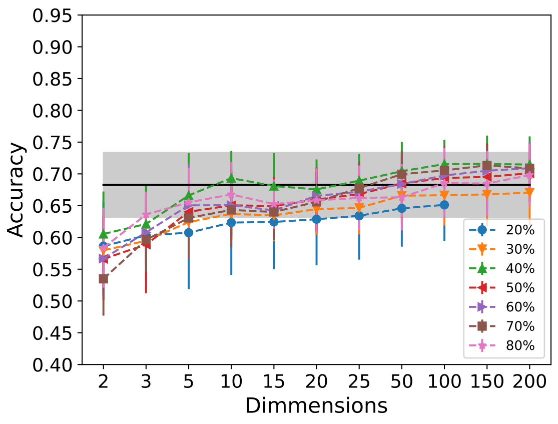
(a) KNORA-E



(b) KNORA-U

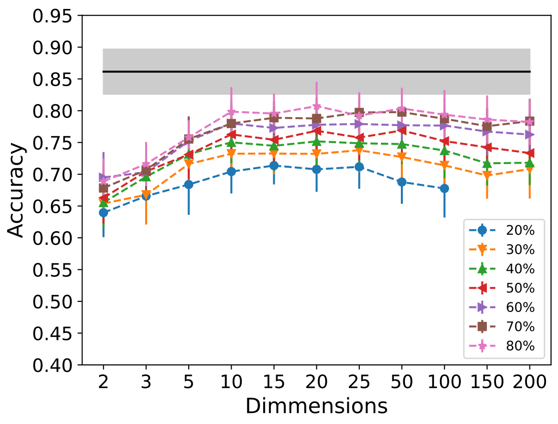


(c) META-DES

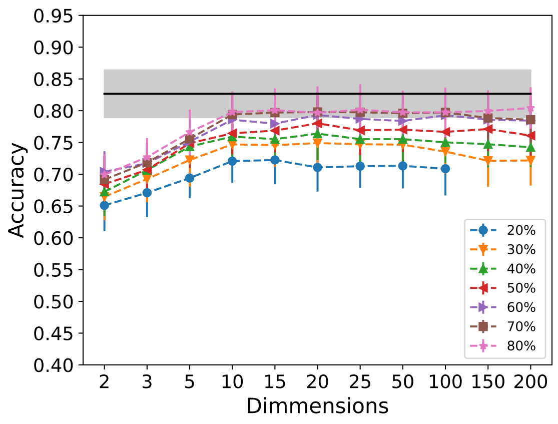


(d) OLP

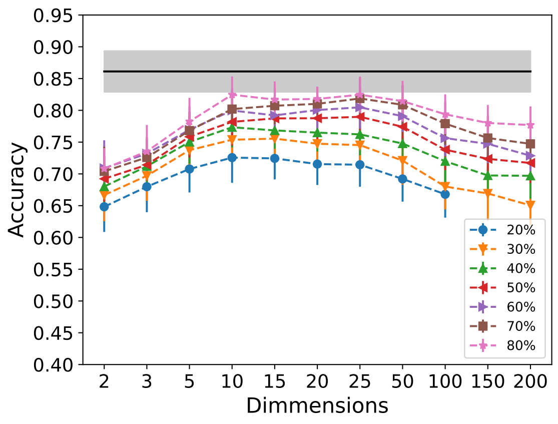
Figure 13 – Accuracy for KNORA-E (Figure 13a), KNORA-U (Figure 13b) and META-DES (Figure 13c) using Bagging and Perceptron, and OLP using SGH (Figure 13d), using Logistic PCA as embedding. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.



(a) KNORA-E

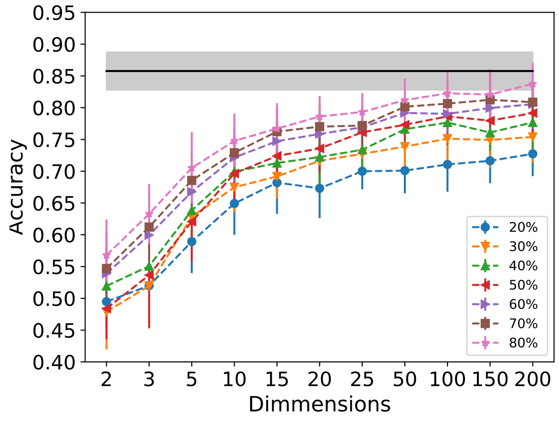


(b) KNORA-U

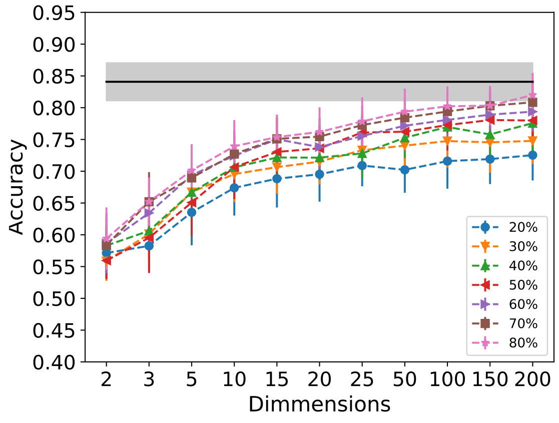


(c) META-DES

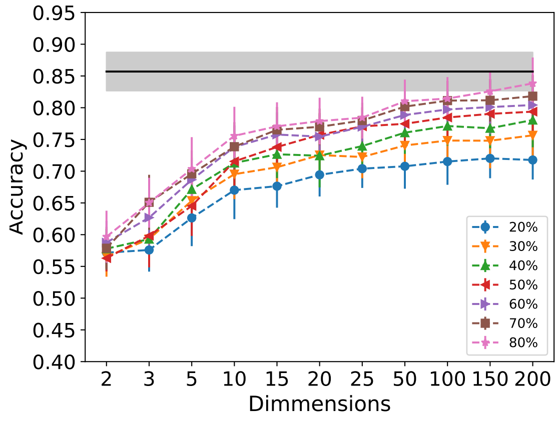
Figure 14 – Accuracy for KNORA-E (Figure 14a), KNORA-U (Figure 14a) and META-DES (Figure 14a) using Logistic PCA and Decision Tree as base classifier. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.



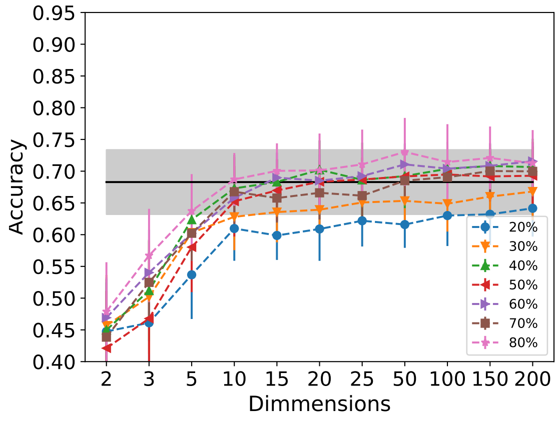
(a) KNORA-E



(b) KNORA-U

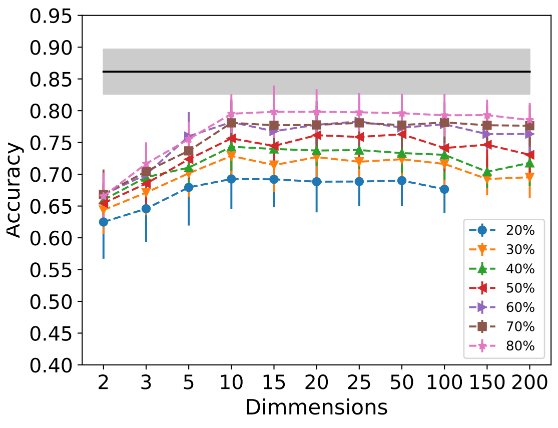


(c) META-DES

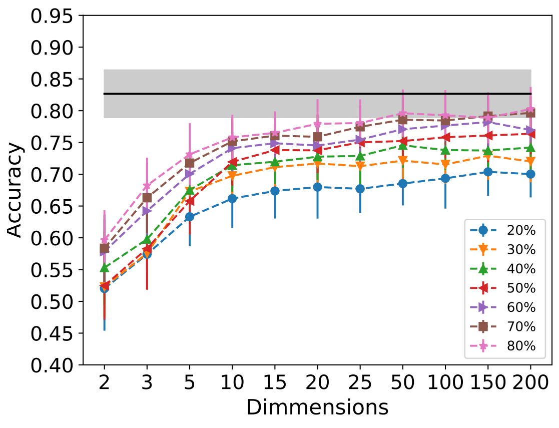


(d) OLP

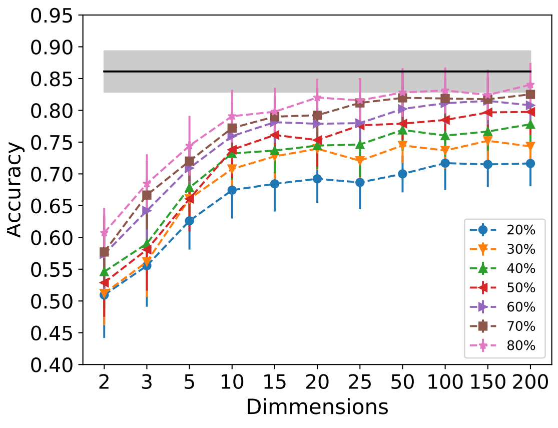
Figure 15 – Accuracy for KNORA-E (Figure 15a), KNORA-U (Figure 15b), META-DES (Figure 15c) using Bagging and Perceptron, and OLP using SGH (Figure 15d), with Autoencoder as embedding. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.



(a) KNORA-E

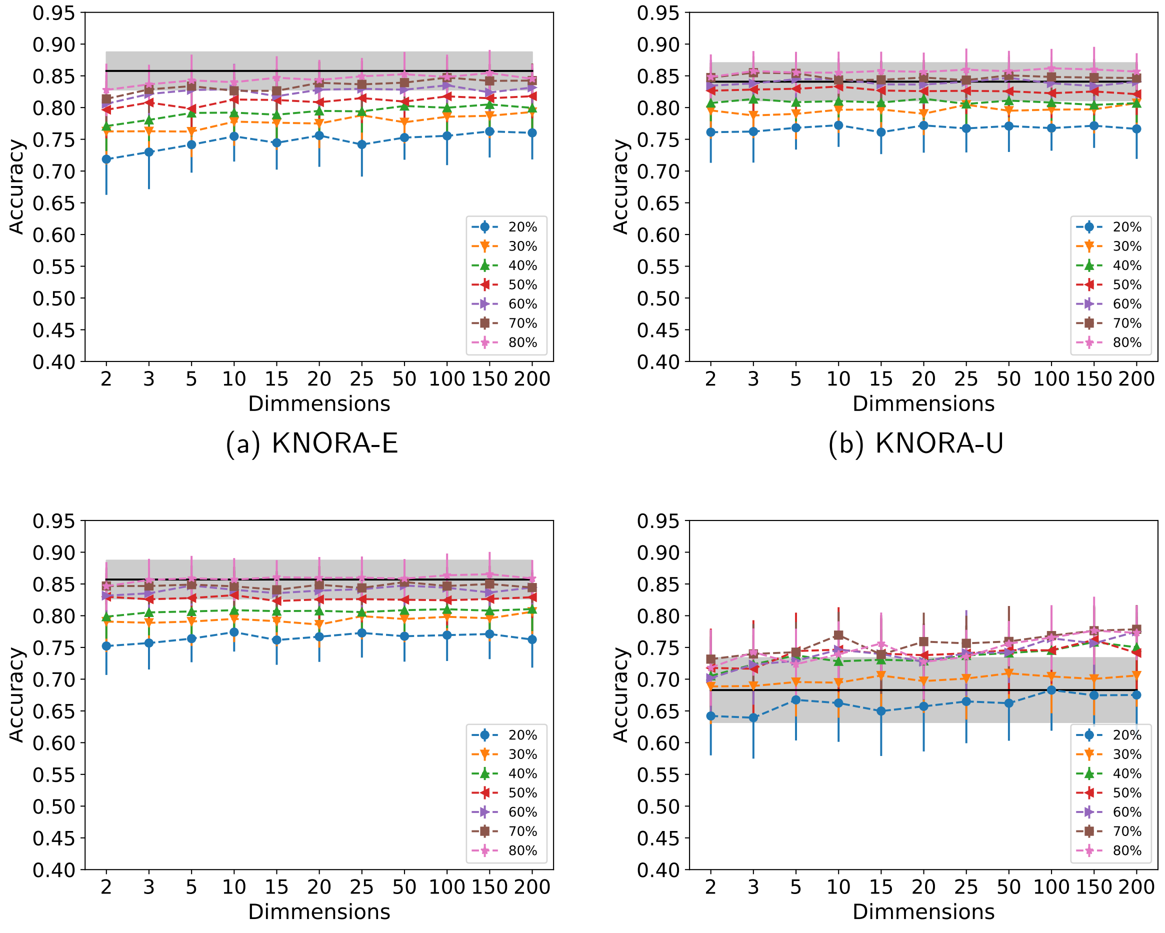


(b) KNORA-U



(c) META-DES

Figure 16 – Accuracy for KNORA-E (Figure 16a), KNORA-U (Figure 16b) and META-DES (Figure 16c) using Autoencoder and Decision Tree as base classifier. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.



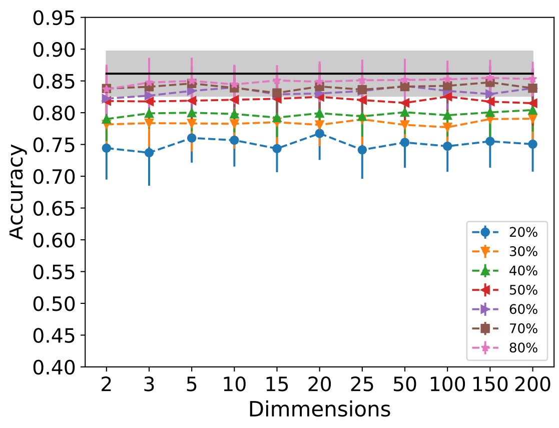
(a) KNORA-E

(b) KNORA-U

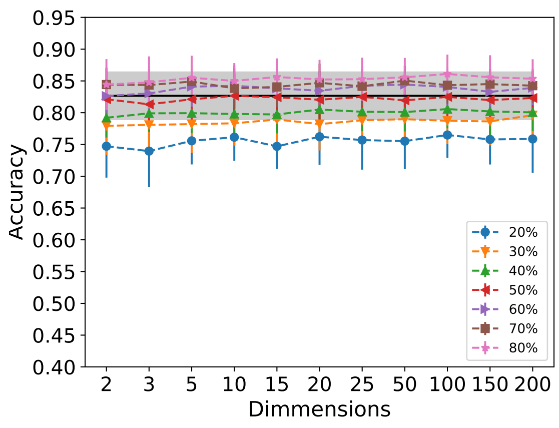
(c) META-DES

(d) OLP

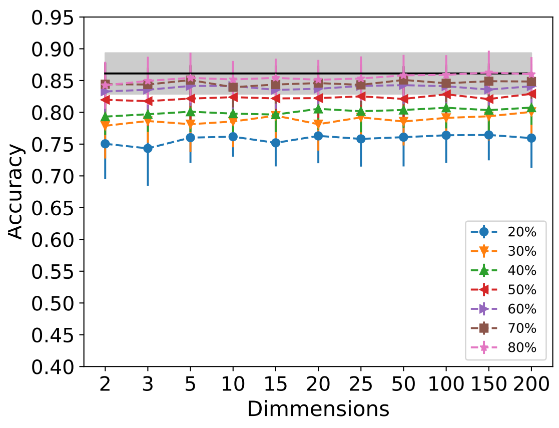
Figure 17 – Accuracy for KNORA-E (Figure 17a), KNORA-U (Figure 17b) and META-DES (Figure 17c) using pool of Perceptrons, and OLP with SGH (Figure 17d), using Triplet Loss as embedding. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.



(a) KNORA-E



(b) KNORA-U



(c) META-DES

Figure 18 – Accuracy for KNORA-E (Figure 18a), KNORA-U (Figure 18b) and META-DES (Figure 18c) using Triplet Loss and Decision Tree as base classifier. The black line is the accuracy for each model using binary features and the grey region is the standard deviation. Each coloured line correspond to the traning set size in the legend. Each point is the model accuracy and vertical bar its accuracy standard deviation.

5.1.3.3 Embeeding Evaluation

Recalling our study of the impact of binary features on MCS, in Section 3, we can compare the embeddings created by PCA, Autoencoder, and Triplet Loss. The specific instance used here is the same in Section 3.

Note that the embedding algorithms transform the binary features into real-valued features. In the Hamming distance case, each bin in the histogram represented every possible distance. However, with real-valued features, each bin represents a range of distances.

Consequently, some modifications were necessary to compare the binary case and the embeddings. We used the L2 distance to evaluate instance similarity, which is the natural distance in the Euclidean space and has no numerical difference in the Hamming space, as we studied in Section 2.2. The distance distribution is divided by its mean to keep it centered at $distance = 1$. This modification standardizes the mean of the distribution to 1, easing the comparison between embeddings algorithms. Additionally, we reduced the maximum x-axis range to 3 and increased the bins' granularity by 20.

First, Figure 19 shows the distances between a specific instance and the rest of the dataset in the embedding created by PCA. We can observe that the distribution still presents ambiguity, such as the original feature set.

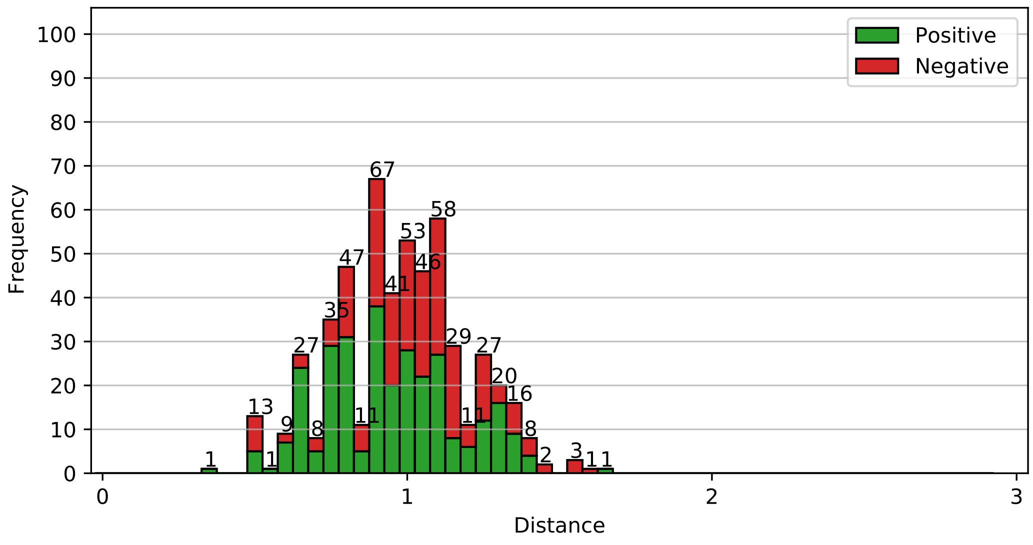


Figure 19 – Distribution of the mean-normalized similarity between an instance and the rest of the dataset using PCA. The x-axis is the distance between samples, and the y-axis is the number of instances in each distance bin. Red and green are negative and positive samples compared to an anchor.

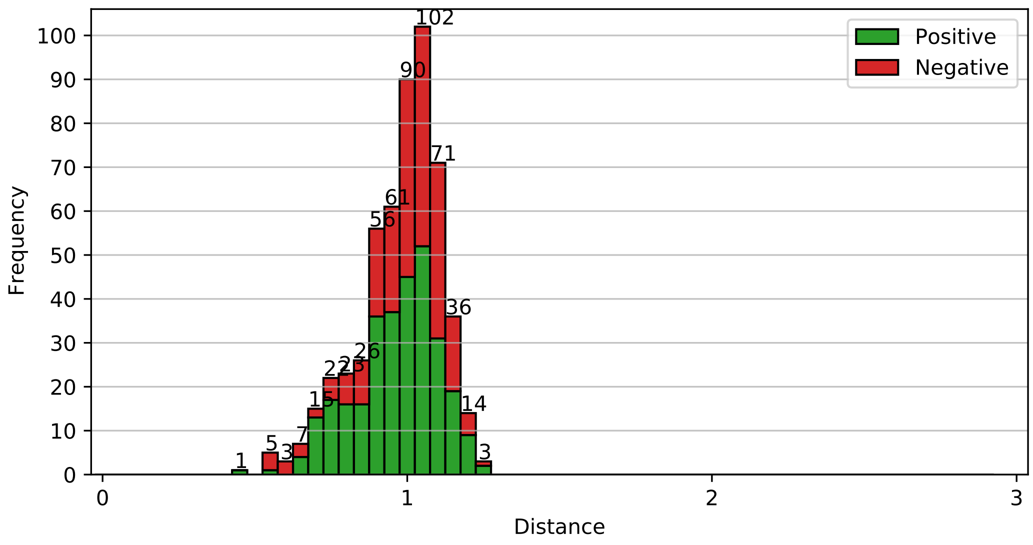


Figure 20 – Distribution of the mean-normalized similarity between an instance and the rest of the dataset using Autoencoder. The x-axis is the distance between samples, and the y-axis is the number of instances in each distance bin. Red and green are negative and positive samples compared to an anchor.

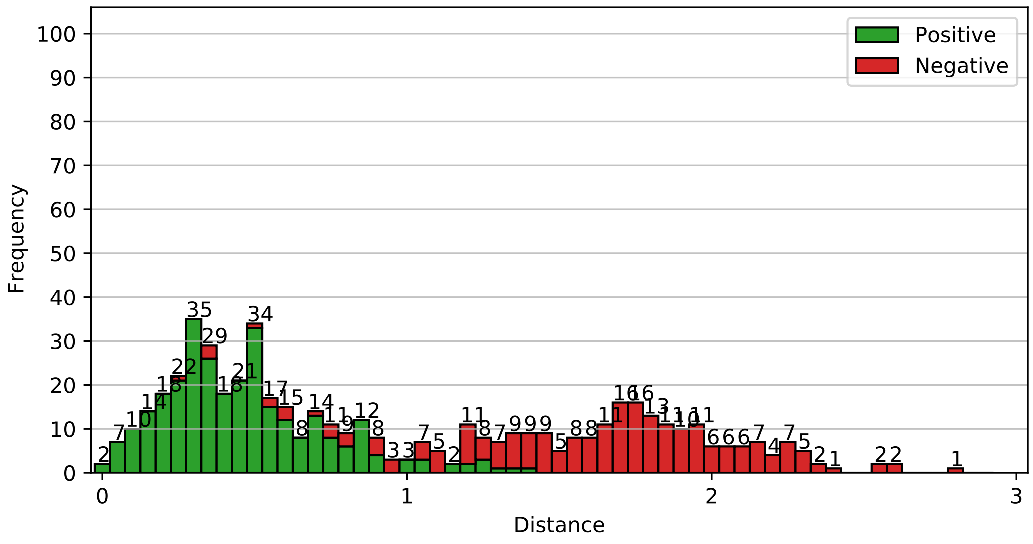


Figure 21 – Distribution of the mean-normalized similarity between an instance and the rest of the dataset using Triplet Loss. The x-axis is the distance between samples, and the y-axis is the number of instances in each distance bin. Red and green are negative and positive samples compared to an anchor.

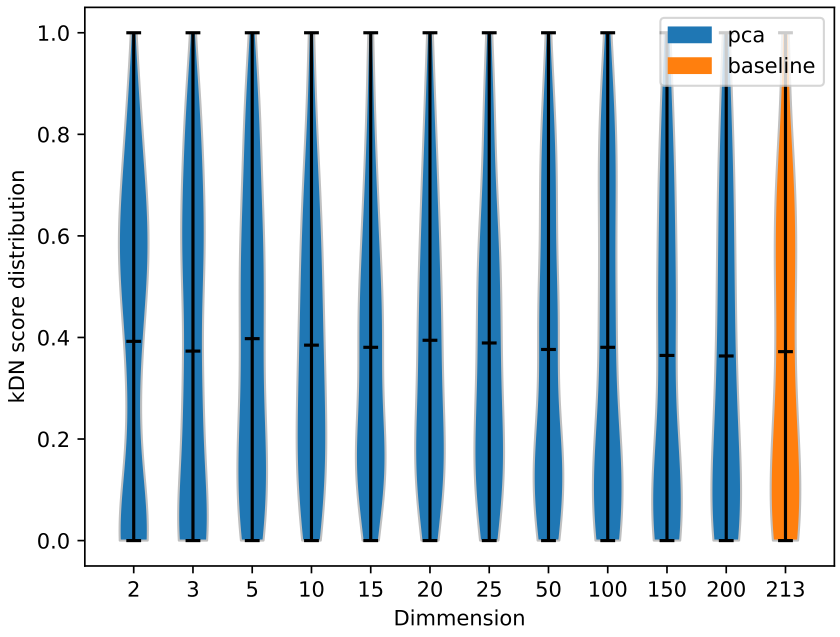


Figure 22 – Distribution of kDN for different embedding dimmensions for PCA, in blue, and the baseline binary features in orange. The higher tick in each graph shows the maximum value, the middle shows the mean value and the lower shows the minimum value of kDN.

When we extrapolate the observed distribution to the whole dataset, we used the violin plot to show the distribution of kDN. The x-axis shows a violin for each embedding dimension. The violin form represents how the kDN is distributed, and the tick shows the mean kDN. Dimension 213 shows the baseline distribution.

Figure 22, and Figure 23, shows the kDN score distribution of PCA and Autoencoder respectively, over multiple dimmensions. The distributions are in line with the expectations observed during the evaluation of the embeddinds with kNN, where there was no improvement in accuracy.

Meanwhile, as Triplet Loss showed an improvement with kNN, we expected that the mean kDN score would be lower than the previous methods. However, we still observe a maximum kDN of 1.0, indicating that even with an easier embedding, there are instances that all its neighbors have a different label, likely pointing to a noisy instance.

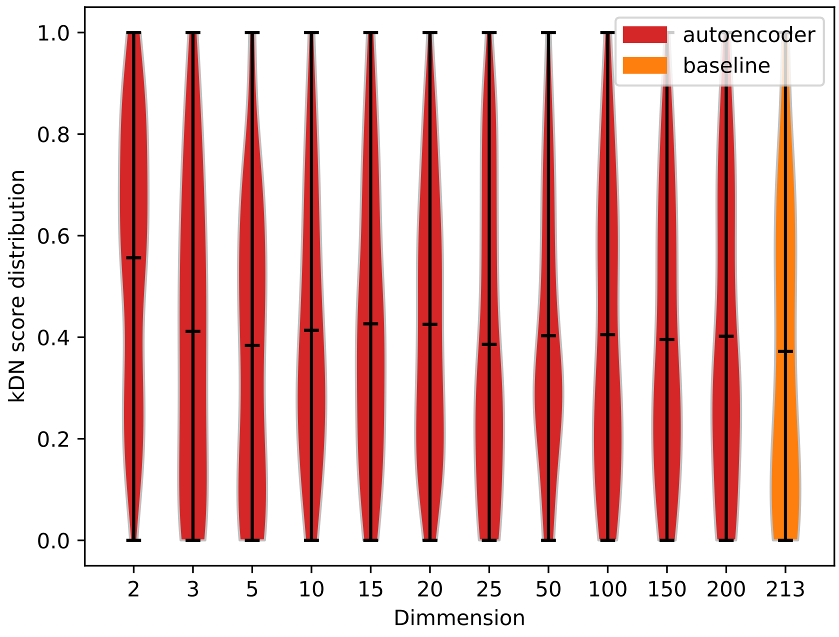


Figure 23 – Distribution of kDN for different embedding dimmensions for Autoencoder, in red, and the baseline binary features in orange. The higher tick in each graph shows the maximum value, the middle shows the mean value and the lower shows the minimum value of kDN.

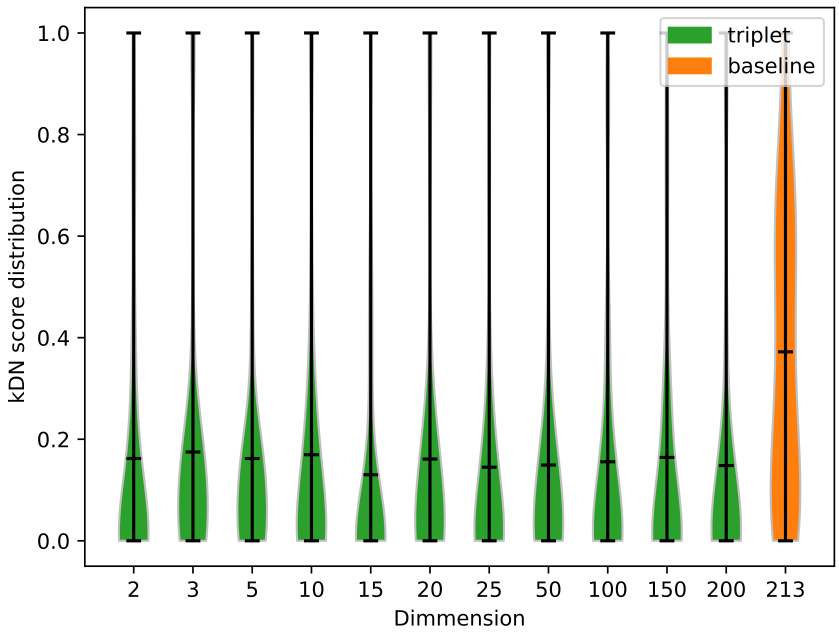


Figure 24 – Distribution of kDN for different embedding dimmensions for Autoencoder, in green, and the baseline binary features in orange. The higher tick in each graph shows the maximum value, the middle shows the mean value and the lower shows the minimum value of kDN.

5.1.4 Conclusion

In our experiments, we conclude that distance-based classifiers, such as kNN and OLP, are the most impacted by binary features, as the distance does not carry information about each feature. As 178 of the binary features are uncorrelated, dimensionality reduction using PCA and Autoencoder will reduce the classifier performance if the number of dimensions is lower than 178. We also note that Logistic PCA has better performance than regular PCA on lower dimensions for monolithic classifiers.

Note that Triplet Loss embedding has a smaller mean kDN, thus, improving techniques that use kNN, which is observed in our results, pairing kNN and Triplet Loss. We also observe that the Triplet Loss creates a better separation between classes while creating an embedding, which improves the performance of distance-based classifiers.

The results using Triplet Loss also show that META-DES, KNORA-E and KNORA-U are less impacted by the binary feature set. Even with a real-valued space with smaller kDN, there is no improvement in the performance compared with the binary feature set.

We also note that the OLP consistently improves its results when paired with Triplet Loss. Our analysis points out that the Hamming distance impacts the kNN classifier, validated by the improved results of OLP and kNN. Thus, using kNN as an underlying classifier is a weakness of the OLP on binary features.

5.2 DISCUSSION REGARDING OLP

Regarding the topic of binary features impacting MCS, discussed in Chapter 3, our rationale is to tackle the disadvantage of using distance-based algorithms in binary features.

The usage of distance-based algorithms, such as kNN, removes the knowledge embedded into each binary feature, replaced by a geometrical interpretation of data. Moreover, as we discussed, this approach has a natural disadvantage when applied to the Hamming space due to the high collision rate, creating an ambiguity between the instances.

In Section 5.1.3.3, we observe that Triplet Loss is an embedding algorithm that improves class separation in the embedded space. The better separation results in performance improvement to kNN, and OLP.

In our discussion about the OLP, let us recall the decision process of the algorithm. It uses the kNN to evaluate if a decision region is hard enough to use a pool of classifiers. An instance is considered hard enough if any sample in the instance's RoC exceeds a certain kDN threshold. Otherwise, a kNN labels the instance, implying losing knowledge about the underlying class distribution for each feature due to geometrically interpretation of data by kNN.

As discussed in Section 3.1.1, considering binary features datasets, using kDN to evaluate if an instance is hard enough can lead to linearly separable instances. Linear separable regions are not a problem, as the perceptron can learn this separation. However, it raises the question of whether it is worthy of training a pool using SGH or a kNN is enough.

Thus, we add a Triplet Loss embedding MLP, within RoC Evaluation phase, to embed the instance to evaluate if the instance's kDN exceeds the threshold. If the kDN is higher than the threshold, the classification algorithm uses a pool to label the instance. Otherwise, the instance is labeled using kNN, using the embedding space. The embedding space is used solely when the kNN is used. Thus, during RoC definition and classification of instances with hardness lower than the threshold.

A MLP trained with the Triplet Loss embeds the data into a feature space, considering the instance's labels. This property reduces the geometrical issue related to binary features, transforming them into a space with lower mean kDN, as presented in Section 5.1.3.3. This new space is pre-separated by the embedding MLP, creating a region in which easier instances are less likely to impact the kNN algorithm, leaving the more problematic instances to be labeled by the pool of classifiers.

Table 11 shows results of DES algorithms using Perceptron as base classifier trained with Bagging, including OLP and the modified OLP (Triplet OLP) using Triplet Loss during the hardness evaluation of an instance. In the table, we observe that it does not perform as the other DES algorithms.

Moreover, we observe a significant improvement over the default OLP on all metrics. These results on accuracy are statistically significant ($p-value = 1.307 \times 10^{-8}$), considering a level of significance of 0.001. However, this modification is not enough to improve the OLP to achieve results comparable to the best MCS so far, META-DES with Random Forest as pool generation ($p-value = 2.111 \times 10^{-10}$).

Model	Precision	Recall	F1 Score	Accuracy
Single Best	0.8157(0.0316)	0.8127(0.0300)	0.8099(0.0317)	0.8127(0.0300)
Static Selection	0.8438(0.0310)	0.8410(0.0312)	0.8386(0.0318)	0.8410(0.0312)
OLA	0.8252(0.0325)	0.8229(0.0329)	0.8213(0.0339)	0.8229(0.0329)
KNORA-U	0.8447(0.0302)	0.8420(0.0307)	0.8399(0.0312)	0.8420(0.0307)
KNORA-E	0.8627(0.0293)	0.8607(0.0295)	0.8596(0.0304)	0.8607(0.0295)
META-DES	0.8638(0.0295)	0.8607(0.0300)	0.8584(0.0308)	0.8607(0.0300)
OLP	0.7244(0.0434)	0.6828(0.0509)	0.6794(0.0538)	0.6828(0.0509)
Triplet OLP	0.7915(0.0408)	0.7846(0.0468)	0.7814(0.0506)	0.7846(0.0468)

Table 11 – Results for MCS classifiers using perceptron as base classifiers, including the modified OLP (Triplet OLP), where red indicates the best model, green the second-best model, and blue the third-best.

6 CONCLUSION AND FUTURE WORKS

This work had the objective to tackle the gap in the literature of SRM classification regarding Dynamic Ensembles due to its crucial role in securing private data, the known improvements of Dynamic Ensembles in classification problems justify our analysis. We discussed the theoretical limitations of using binary features, particularly with the OLP algorithm.

The binary features in SRM classification correspond to the presence or lack of a particular string, property, or attribute in the Android's API methods. Then, a classifier uses these features to classify the method as a source or sink of sensitive data or does not operate in sensitive data.

We show that, unlike in previous works in SRM classification, SVM is outperformed by Random Forest, Gradient Boosted Decision Trees, MLP and META-DES combined with Random Forest as the pool generation algorithm. Thus, using any of the mentioned classifiers lead to an improved list of SRM.

Our analysis demonstrated that these features directly impact distance-based classifiers, such as kNN. As the underlying distribution of each binary feature is disregarded to give a geometrical interpretation of data, the high collision rate between distinct instances leads to ambiguity in the neighborhood of a test instance. The high collision rate is intrinsic to the Hamming space, as the amount of collisions scales with the number of binary features. High collision rates are deleterious to kNN as heuristics to solve the collisions can harm the algorithm performance. We confirm this concern in our experiments, as the kNN had the worse performance of monolithic classifiers.

The concern over kNN can be extended to MCS, as KNORA-E, KNORA-U, META-DES, OLP, OLA uses kNN to create the RoC of an instance. Unlikely with a pure kNN, the pool of classifiers is capable of labeling an instance correctly, observed by the Oracle result. However, the majority of DS fails to select the most competent classifiers, performing worse than monolithic. Combining META-DES and Random Forest as pool generation methods is the only DES technique achieving similar results to MLP, Random Forest, and Gradient Boosted Decision Trees. Thus, simply considering the accuracy or finding the local oracles in the RoC for a dataset of binary features adds little information to the problem of selecting the best classifiers in the pool.

From the experiments with the embedding algorithms, we can conclude that dimensionality reduction negatively impacts the classifiers' performance, explained by 178 linearly independent features, according to our analysis using PCA. Additionally, despite the KNORA-E, KNORA-U, and META-DES using kNN to generate the RoC, none of these classifiers improve their performance when using the Triplet Loss embedding. Unlike kNN and OLP, where we observe the most improvement.

Finally, our discussion regarding OLP shows that there is a margin for improvements on DS techniques regarding binary features. Using a Triplet Loss embedding in the RoC Evaluation improves the results of OLP. By embedding the instances during RoC Evaluation, we avoid the linear separable RoC to be considered hard. Also, it tackles the ambiguity issue of kNN and binary features, as if the region is considered easy, the kNN works in a real-valued space. Although these modifications statistically improve the OLP results, it is worse than monolithic classifiers.

6.1 FUTURE WORKS

To fully assess the impact of binary features over DS techniques, it is required to isolate the limitations related to the pool generation algorithms and the DS methods themselves. Hence, a set of datasets of solely binary features is required to reduce the specificity of a single application.

It is known in the literature that the pool generation algorithm can limit the DCS performance. Souza et al. (2017) observes that Bagging and Random Subspaces do not guarantee an Oracle of 100% in the training set. Developing a specific pool generation technique, accounting for the specificity of a dataset of binary features, could further improve DS performance.

Our results show that combining META-DES and Random Forest is the best DES method for the problem. However, the high Oracle performance of Bagging, Boosting, and Random Forest, points to a possible limitation in the DES techniques when applied to binary features. A solution would be creating a method less reliant on the kNN as RoC generation method. Cruz, Sabourin and Cavalcanti (2018) reports other techniques to define the RoC, namely clustering and potential functions. However, clustering and potential functions relying on euclidean distances over binary features could also lead to the same ambiguity of kNN.

REFERENCES

AKSELA, M. Comparison of classifier selection methods for improving committee performance. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 2709, p. 84–93, 2003. ISSN 16113349.

ARP, D.; SPREITZENBARTH, M.; HÜBNER, M.; GASCON, H.; RIECK, K. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. n. February, 2014.

ARZT, S.; RASTHOFER, S.; FRITZ, C.; BODDEN, E.; BARTEL, A.; KLEIN, J.; Le Traon, Y.; OCTEAU, D.; MCDANIEL, P. FLOWDROID: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM SIGPLAN Notices*, v. 49, n. 6, p. 259–269, 2014. ISSN 15232867.

BADHANI, S.; MUTTOO, S. K. CENDroid—A cluster-ensemble classifier for detecting malicious Android applications. *Computers and Security*, Elsevier Ltd, v. 85, p. 25–40, 2019. ISSN 01674048.

BENAVOLI, A.; CORANI, G.; MANGILI, F. Should we Really use Post-hoc Tests Based on Mean-ranks? *Journal of Machine Learning Research*, v. 17, p. 1–10, 2016. ISSN 15337928.

BISHOP, C. M.; NASRABADI, N. M. *Pattern Recognition and Machine Learning*. 4. ed. [S.l.]: Springer, 2006. ISBN 9780387310732.

BREIMAN, L. Bagging Predictors. *Machine Learning*, v. 24, p. 123–140, 1996. ISSN 22279091.

BRITTO, A. S.; SABOURIN, R.; OLIVEIRA, L. E. Dynamic selection of classifiers - A comprehensive review. *Pattern Recognition*, v. 47, n. 11, p. 3665–3680, 2014. ISSN 00313203.

CAVALIN, P. R.; SABOURIN, R.; SUEN, C. Y. Dynamic selection approaches for multiple classifier systems. *Neural Computing and Applications*, v. 22, n. 3-4, p. 673–688, 2013. ISSN 09410643.

COLLINS, M.; SCHAPIRE, R. E.; AVENUE, P.; PARK, F. A Generalization of Principal Component Analysis to the Exponential Family. In: *Neural information processing systems*. [S.l.: s.n.], 2001. p. 23.

CRUZ, R. M.; SABOURIN, R.; CAVALCANTI, G. D. Dynamic Classifier Selection: Recent Advances and Perspectives. *Information Fusion*, Elsevier B.V., v. 41, p. 195–216, 2018. ISSN 15662535. Available at: <http://dx.doi.org/10.1016/j.inffus.2017.09.010>.

CRUZ, R. M.; SABOURIN, R.; CAVALCANTI, G. D.; Ing Ren, T. META-DES: A dynamic ensemble selection framework using meta-learning. *Pattern Recognition*, Elsevier, v. 48, n. 5, p. 1925–1935, 2015. ISSN 00313203. Available at: <http://dx.doi.org/10.1016/j.patcog.2014.12.003>.

CRUZ, R. M. O.; HAFEMANN, L. G.; SABOURIN, R.; CAVALCANTI, G. D. C. Deslib: A dynamic ensemble selection library in python. *Journal of Machine Learning Research*, v. 21, n. 8, p. 1–5, 2020. Available at: <http://jmlr.org/papers/v21/18-144.html>.

DIDACI, L.; GIACINTO, G. Dynamic Classifier Selection by Adaptive k-Nearest-Neighbourhood Rule. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 3077, p. 174–183, 2004. ISSN 16113349.

FEIZOLLAH, A.; ANUAR, N. B.; SALLEH, R.; WAHAB, A. W. A. A review on feature selection in mobile malware detection. *Digital Investigation*, Elsevier Ltd, v. 13, p. 22–37, 2015. ISSN 17422876. Available at: <http://dx.doi.org/10.1016/j.diin.2015.02.001>.

FERNANDES, E.; PAUPORE, J.; RAHMATI, A. FlowFence : Practical Data Protection for Emerging IoT Application Frameworks This paper is included in the Proceedings of the FlowFence : Practical Data Protection for. 2016.

FREUND, Y.; SCHAPIRE, R. E. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, v. 55, n. 1, p. 119–139, 1997. ISSN 00220000.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [s.n.], 2016. ISBN 0262035618. Available at: <http://www.deeplearningbook.org>.

GUI, J.; CAO, Y.; QI, H.; LI, K.; YE, J.; LIU, C.; XU, X. Fast kNN Search in Weighted Hamming Space with Multiple Tables. *IEEE Transactions on Image Processing*, v. 30, p. 3985–3994, 2021. ISSN 19410042.

HO, T. K. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 20, n. 8, p. 832–844, 1998. ISSN 01628828.

KAYA, M.; BILGE, H. S. Deep Metric Llearning: A Survey. *Symmetry*, v. 11, n. 9, p. 1066, 2019. ISSN 20738994. Available at: <https://www.mdpi.com/2073-8994/11/9/1066>.

KITTLER, J.; HATEF, M.; DUIN, R. P.; MATAS, J. On Combining Classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 20, n. 3, p. 226–239, 1998. ISSN 01628828.

KO, A. H.; SABOURIN, R.; BRITTO, A. S. From dynamic classifier selection to dynamic ensemble selection. *Pattern Recognition*, v. 41, n. 5, p. 1718–1731, 2008. ISSN 00313203.

KRUSKAL, W. H.; WALLIS, W. A. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, Taylor & Francis, v. 47, n. 260, p. 583–621, 1952.

KUNCHEVA, L. A theoretical study on six classifier fusion strategies. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 24, n. 2, p. 281–286, 2002. ISSN 01628828. Available at: <http://ieeexplore.ieee.org/document/982906/>.

KUNCHEVA, L. I. Clustering-and-selection model for classifier combination. In: *International Conference on Knowledge-Based Intelligent Engineering Systems and Allied Technologies*. [S.l.: s.n.], 2000. v. 1, p. 185–188. ISBN 0780364007.

KUNCHEVA, L. I. *Combining Pattern Classifiers: Methods and Algorithms*. [S.l.]: John Wiley & Sons, 2014. ISSN 0040-1706. ISBN 9786468600.

LANDGRAF, A. J.; LEE, Y. Dimensionality Reduction for Binary Data Through the Projection of Natural Parameters. *Journal of Multivariate Analysis*, Elsevier Inc., v. 180, p. 104668, 2020. ISSN 10957243. Available at: <https://doi.org/10.1016/j.jmva.2020.104668>.

LI, L.; BARTEL, A.; BISSYANDÉ, T. F.; KLEIN, J.; TRAON, Y. L.; ARZT, S.; RASTHOFER, S.; BODDEN, E.; OCTEAU, D.; MCDANIEL, P. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. *Proceedings - International Conference on Software Engineering*, v. 1, p. 280–291, 2015. ISSN 02705257.

LI, L.; BISSYANDé, T. F.; PAPADAKIS, M.; RASTHOFER, S.; BARTEL, A.; KLEIN, J.; TRAON, Y. L. Static Analysis of Android Apps : A Systematic Literature Review. *Information and Software Technology*, p. 1–22, 2017.

LIU, K.; XU, S.; XU, G.; ZHANG, M.; SUN, D.; LIU, H. A review of android malware detection approaches based on machine learning. *IEEE Access*, IEEE, v. 8, p. 124579–124607, 2020.

MARTÍN, A.; LARA-CABRERA, R.; CAMACHO, D. Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset. *Information Fusion*, v. 52, n. September 2018, p. 128–142, 2019. ISSN 15662535.

NARUDIN, F. A.; FEIZOLLAH, A.; ANUAR, N. B.; GANI, A. Evaluation of Machine Learning Classifiers for Mobile Malware Detection. *Soft Computing*, Springer Berlin Heidelberg, v. 20, n. 1, p. 343–357, 2016. ISSN 14337479. Available at: <http://dx.doi.org/10.1007/s00500-014-1511-6>.

PANG, Y.; PENG, L.; CHEN, Z.; YANG, B.; ZHANG, H. Imbalanced learning based on adaptive weighting and Gaussian function synthesizing with an application on Android malware detection. *Information Sciences*, Elsevier Inc., v. 484, p. 95–112, 2019. ISSN 00200255.

PEARSON, K. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, v. 2, n. 11, p. 559–572, 1901. ISSN 1941-5982.

PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, v. 12, p. 2825–2830, 2011.

PISKACHEV, G.; BODDEN, E. Codebase-Adaptive Detection of Security-Relevant Methods. p. 181–191, 2019.

RASCHKA, S. Model evaluation, model selection, and algorithm selection in machine learning. *arXiv preprint arXiv:1811.12808*, 2018.

RASTHOFER, S.; ARZT, S.; BODDEN, E. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. n. February, p. 23–26, 2014.

SAKURADA, M.; YAIRI, T. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In: *ACM International Conference Proceeding Series*. [S.l.: s.n.], 2014. v. 02-Decembe, p. 4–11. ISBN 9781450331593.

SAS, D.; BESSI, M.; Arcelli Fontana, F. Automatic detection of sources and sinks in arbitrary Java libraries. *Proceedings - 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018*, IEEE, p. 103–112, 2018.

SHEEN, S.; ANITHA, R.; NATARAJAN, V. Android based malware detection using a multifeature collaborative decision fusion approach. *Neurocomputing*, Elsevier, v. 151, n. P2, p. 905–912, 2015. ISSN 18728286. Available at: <http://dx.doi.org/10.1016/j.neucom.2014.10.004>.

SIERRA, B.; LAZKANO, E.; IRIGOIEN, I.; JAUREGI, E.; MENDIALDUA, I. K Nearest Neighbor Equality: Giving Equal Chance to All Existing Classes. *Information Sciences*, Elsevier Inc., v. 181, n. 23, p. 5158–5168, 2011. ISSN 00200255. Available at: <http://dx.doi.org/10.1016/j.ins.2011.07.024>.

SMITH, M. R.; MARTINEZ, T.; GIRAUD-CARRIER, C. An Instance Level Analysis of Data Complexity. *Machine Learning*, v. 95, n. 2, p. 225–256, 2014. ISSN 15730565.

SOUZA, M. A.; CAVALCANTI, G. D.; CRUZ, R. M.; SABOURIN, R. On the characterization of the Oracle for dynamic classifier selection. In: *Proceedings of the International Joint Conference on Neural Networks*. [S.l.]: IEEE, 2017. v. 2017-May, p. 332–339. ISBN 9781509061815.

SOUZA, M. A.; CAVALCANTI, G. D.; CRUZ, R. M.; SABOURIN, R. Online local pool generation for dynamic classifier selection. *Pattern Recognition*, Elsevier Ltd, v. 85, p. 132–148, 2019. ISSN 00313203. Available at: <https://doi.org/10.1016/j.patcog.2018.08.004>.

SPOTO, F.; BURATO, E.; ERNST, M. D.; FERRARA, P.; LOVATO, A.; MACEDONIO, D.; SPIRIDON, C. Static identification of injection attacks in Java. *ACM Transactions on Programming Languages and Systems*, v. 41, n. 3, 2019. ISSN 15584593.

TAHERI, R.; GHAHRAMANI, M.; JAVIDAN, R.; SHOJAFAR, M. Similarity-based Android Malware Detection Using Hamming Distance of Static Binary Features. *Future Generation Computer Systems*, Elsevier B.V., v. 105, p. 230–247, 2020. ISSN 0167-739X. Available at: <https://doi.org/10.1016/j.future.2019.11.034>.

TIPPING, M. E. Probabilistic Visualisation of High-dimensional Binary Data. *Advances in Neural Information Processing Systems*, v. 11, 1998.

WALMSLEY, F. N.; CAVALCANTI, G. D.; OLIVEIRA, D. V.; CRUZ, R. M.; SABOURIN, R. An Ensemble Generation Method Based on Instance Hardness. *Proceedings of the International Joint Conference on Neural Networks*, IEEE, v. 2018-July, p. 1–8, 2018.

WANG, S.; CHEN, Z.; YAN, Q.; YANG, B.; PENG, L.; JIA, Z. A Mobile Malware Detection Method Using Behavior Features in Network Traffic. *Journal of Network and Computer Applications*, Elsevier Ltd, v. 133, n. January, p. 15–25, 2019. ISSN 10958592. Available at: <https://doi.org/10.1016/j.jnca.2018.12.014>.

WANG, Z.; BOVIK, A. C.; SHEIKH, H. R.; SIMONCELLI, E. P. Image quality assessment: From error visibility to structural similarity. *IEEE transactions on image processing*, IEEE, v. 13, n. 4, p. 600–612, 2004.

WEINBERGER, K. Q.; SAUL, L. K. Distance metric learning for large margin nearest neighbor classification. *Journal of machine learning research*, v. 10, n. 2, 2009.

WOLOSZYNSKI, T.; KURZYNSKI, M. A probabilistic model of classifier competence for dynamic ensemble selection. *Pattern Recognition*, Elsevier, v. 44, n. 10-11, p. 2656–2668, 2011. ISSN 00313203. Available at: <http://dx.doi.org/10.1016/j.patcog.2011.03.020>.

WONG, M. Y.; LIE, D. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. n. February, p. 21–24, 2017.

WOODS, K.; JR, W. P. K.; BOWYER, K. Combination of Multiple Classifiers Using Local Accuracy Estimates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 19, n. 4, p. 405–410, 1997.

WU, C.-Y.; MANMATHA, R.; SMOLA, A. J.; KRAHENBUHL, P. Sampling matters in deep embedding learning. In: *Proceedings of the IEEE International Conference on Computer Vision*. [S.l.: s.n.], 2017. p. 2840–2848.

WU, S.; WANG, P.; LI, X.; ZHANG, Y. Effective detection of android malware based on the usage of data flow APIs and machine learning. *Information and Software Technology*, Elsevier B.V., v. 75, p. 17–25, 2016. ISSN 0950-5849. Available at: <http://dx.doi.org/10.1016/j.infsof.2016.03.004>.

YANG, X.; LO, D.; LI, L.; XIA, X.; BISSYANDÉ, T. F.; KLEIN, J. Characterizing Malicious Android Apps by Mining Topic-specific Data Flow Signatures. *Information and Software Technology*, v. 90, p. 27–39, 2017. ISSN 09505849.

YERIMA, S. Y.; SEZER, S.; MUTTIK, I. High accuracy android malware detection using ensemble learning. *IET Information Security*, v. 9, n. 6, p. 313–320, 2015. ISSN 17518717.

ZHANG, C.; MA, Y. *Ensemble Machine Learning: Methods and Applications*. [S.l.]: Springer, 2012. ISBN 9781441993250.

ZHANG, L.; ZHANG, Y.; TANG, J.; LU, K.; TIAN, Q. Binary Code Ranking with Weighted Hamming Distance. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. [S.l.: s.n.], 2013. p. 1586–1593. ISSN 10636919.

ZHANG, Y.; REN, W.; ZHU, T.; REN, Y. SaaS: A situational awareness and analysis system for massive android malware detection. *Future Generation Computer Systems*, Elsevier B.V., v. 95, p. 548–559, 2019. ISSN 0167739X. Available at: <https://doi.org/10.1016/j.future.2018.12.028>.

ZHU, D.; JIN, H.; YANG, Y.; WU, D.; CHEN, W. DeepFlow: Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data. *Proceedings - IEEE Symposium on Computers and Communications*, p. 438–443, 2017. ISSN 15301346.

ZHU, H. J.; YOU, Z. H.; ZHU, Z. X.; SHI, W. L.; CHEN, X.; CHENG, L. DroidDet: Effective and robust detection of android malware using static analysis along with rotation forest model. *Neurocomputing*, Elsevier B.V., v. 272, p. 638–646, 2018. ISSN 18728286. Available at: <https://doi.org/10.1016/j.neucom.2017.07.030>.

ZHU, J.; ZOU, H.; ROSSET, S.; HASTIE, T. Multi-class AdaBoost. *Statistics and its Interface*, v. 2, p. 349–360, 2009.

APPENDIX A – FEATURES AND FEATURE TYPES

Table 12 – List of all features extracted. Class 1 means a Semantic feature, Class 2 a Syntactic feature and 3 a Data-Flow feature.

Class	Feature
1	Base name of class package name: accounts
1	Base name of class package name: io
1	Base name of class package name: music
1	Base name of class package name: telephony
1	Base name of class package name: webkit
1	Method has parameters
1	Method is lone getter or setter
2	Method is part of a ABSTRACT class
2	Method is part of a FINAL class
2	Method is part of a PRIVATE class
2	Method is part of a PROTECTED class
2	Method is part of a PUBLIC class
2	Method is part of a STATIC class
2	Method is part of an anonymous class
2	Method is part of class android.app.Activity
2	Method is part of class android.app.BroadcastReceiver
2	Method is part of class android.app.ContentProvider
2	Method is part of class android.app.Service
2	Method is part of class android.content.ContentResolver
2	Method is part of class android.content.Context
2	Method is part of class that contains the name com.google.common.io
2	Method is part of class that contains the name java.io.

2	Method is part of class that ends with Context
2	Method is part of class that ends with Factory
2	Method is part of class that ends with Handler
2	Method is part of class that ends with Loader
2	Method is part of class that ends with Manager
2	Method is part of class that ends with Service
2	Method is part of class that ends with View
2	Method is thread runner
2	Method modifier is FINAL
2	Method modifier is PROTECTED
2	Method modifier is PUBLIC
2	Method modifier is STATIC
1	Method name ends with Messenger
1	Method name starts with <init>
1	Method name starts with add
1	Method name starts with apply
1	Method name starts with bind
1	Method name starts with clear
1	Method name starts with close
1	Method name starts with delete
1	Method name starts with disable
1	Method name starts with dispatch
1	Method name starts with do
1	Method name starts with dump
1	Method name starts with enable
1	Method name starts with finish
1	Method name starts with get

1	Method name starts with handle
1	Method name starts with insert
1	Method name starts with is
1	Method name starts with load
1	Method name starts with note
1	Method name starts with notify
1	Method name starts with onClick
1	Method name starts with open
1	Method name starts with perform
1	Method name starts with process
1	Method name starts with put
1	Method name starts with query
1	Method name starts with register
1	Method name starts with release
1	Method name starts with remove
1	Method name starts with request
1	Method name starts with restore
1	Method name starts with run
1	Method name starts with send
1	Method name starts with set
1	Method name starts with start
1	Method name starts with supply
1	Method name starts with toggle
1	Method name starts with unregister
1	Method name starts with update
2	Method returns constant
2	Method starts with on and has void/bool return type

2	Parameter is interface
3	Parameter to abstract sink
3	Parameter to sink method adjust
3	Parameter to sink method bind
3	Parameter to sink method broadcast
3	Parameter to sink method clear
3	Parameter to sink method com.android.internal.telephony.CommandsInterface
3	Parameter to sink method connect
3	Parameter to sink method create
3	Parameter to sink method delete
3	Parameter to sink method dial
3	Parameter to sink method disable
3	Parameter to sink method dispatch
3	Parameter to sink method dump
3	Parameter to sink method enable
3	Parameter to sink method enqueue
3	Parameter to sink method insert
3	Parameter to sink method notify
3	Parameter to sink method onCreate
3	Parameter to sink method perform
3	Parameter to sink method println
3	Parameter to sink method put
3	Parameter to sink method remove
3	Parameter to sink method replace
3	Parameter to sink method restore
3	Parameter to sink method save
3	Parameter to sink method send

3	Parameter to sink method set
3	Parameter to sink method setup
3	Parameter to sink method show
3	Parameter to sink method start
3	Parameter to sink method sync
3	Parameter to sink method transact
3	Parameter to sink method update
3	Parameter to sink method write
2	Parameter type contains android.content.contentresolver
2	Parameter type contains android.content.context
2	Parameter type contains android.content.intent
2	Parameter type contains android.database.cursor
2	Parameter type contains android.filterfw.core.filtercontext
2	Parameter type contains android.net.uri
2	Parameter type contains com.android.inputmethod.keyboard.key
2	Parameter type contains com.google.common.io
2	Parameter type contains event
2	Parameter type contains java.io.
2	Parameter type contains java.io.filedescriptor
2	Parameter type contains java.lang.string
2	Parameter type contains observer
2	Parameter type contains writer
1	Permission name is ACCESS COARSE LOCATION
1	Permission name is ACCESS FINE LOCATION
1	Permission name is ACCESS LOCATION EXTRA COMMANDS
1	Permission name is ACCESS NETWORK STATE
1	Permission name is ACCESS WIFI STATE

1	Permission name is ADD VOICEMAIL
1	Permission name is AUTHENTICATE ACCOUNTS
1	Permission name is BACKUP
1	Permission name is BLUETOOTH
1	Permission name is BLUETOOTH ADMIN
1	Permission name is BROADCAST STICKY
1	Permission name is CALL PHONE
1	Permission name is CALL PRIVILEGED
1	Permission name is CAMERA
1	Permission name is CHANGE CONFIGURATION
1	Permission name is CHANGE NETWORK STATE
1	Permission name is CHANGE WIFI STATE
1	Permission name is CLEAR APP USER DATA
1	Permission name is DEVICE POWER
1	Permission name is DISABLE KEYGUARD
1	Permission name is DUMP
1	Permission name is GET ACCOUNTS
1	Permission name is GET TASKS
1	Permission name is GLOBAL SEARCH
1	Permission name is INTERNET
1	Permission name is KILL BACKGROUND PROCESSES
1	Permission name is MANAGE ACCOUNTS
1	Permission name is MANAGE APP TOKENS
1	Permission name is MODIFY AUDIO SETTINGS
1	Permission name is MODIFY PHONE STATE
1	Permission name is MOUNT UNMOUNT FILESYSTEMS
1	Permission name is NFC

1	Permission name is READ CALENDAR
1	Permission name is READ CALL LOG
1	Permission name is READ CONTACTS
1	Permission name is READ EXTERNAL STORAGE
1	Permission name is READ HISTORY BOOKMARKS
1	Permission name is READ PHONE STATE
1	Permission name is READ SMS
1	Permission name is READ SOCIAL STREAM
1	Permission name is READ SYNC SETTINGS
1	Permission name is READ SYNC STATS
1	Permission name is READ USER DICTIONARY
1	Permission name is REBOOT
1	Permission name is RECEIVE BOOT COMPLETED
1	Permission name is RECEIVE SMS
1	Permission name is RECORD AUDIO
1	Permission name is RESTART PACKAGES
1	Permission name is SEND SMS
1	Permission name is SET DEBUG APP
1	Permission name is SET TIME ZONE
1	Permission name is SET WALLPAPER
1	Permission name is SET WALLPAPER COMPONENT
1	Permission name is STOP APP SWITCHES
1	Permission name is SYSTEM ALERT WINDOW
1	Permission name is UPDATE DEVICE STATS
1	Permission name is USE CREDENTIALS
1	Permission name is USE SIP
1	Permission name is VIBRATE

1	Permission name is WAKE LOCK
1	Permission name is WRITE CALENDAR
1	Permission name is WRITE CONTACTS
1	Permission name is WRITE EXTERNAL STORAGE
1	Permission name is WRITE HISTORY BOOKMARKS
1	Permission name is WRITE SETTINGS
1	Permission name is WRITE SMS
1	Permission name is WRITE SOCIAL STREAM
1	Permission name is WRITE SYNC SETTINGS
1	Permission name is WRITE USER DICTIONARY
2	Return type is android.database.Cursor
2	Return type is android.net.Uri
2	Return type is android.os.Parcelable
2	Return type is boolean
2	Return type is byte[]
2	Return type is com.android.internal.telephony.Connection
2	Return type is int
2	Return type is java.util.List
2	Return type is java.util.Map
2	Return type is void
3	Value from method get to sink method
3	Value from method parameter to native method
3	Value from source method create to return
3	Value from source method get to return
3	Value from source method is to return
3	Value from source method obtainMessage to return
3	Value from source method query to return

3	Value from source method writeToParcel to return
3	Method starting with insert invoked