UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

A Hybrid Machine Learning System for Vulnerability Detection in Web Applications

Miguel César de Albuquerque Oliveira

Mestrado em Ciências de Dados

Dissertação orientada por:
Prof. Doutor Ibéria Vitória de Sousa Medeiros

2023

Acknowledgements

I would like to express my heartfelt gratitude to all those who have made it possible for me to pursue this research project. Without your support, I would not have been able to embark on this journey of learning and discovery.

I want to begin by thanking my advisor Prof. Ibéria Medeiros for giving me this opportunity and for the continuous kindness and support throughout this project. I have learned a lot from our conversations, the guidance and positive environment that she created at all times. She gave me confidence to pursue my ideas and always gave me the right pointers to advance. I will always keep with me the feeling and enthusiasm she has shared with me for learning, sharing ideas and solving problems. I am very grateful to have had the opportunity to be your student.

I would also like to thank all of the colleagues I have met during my time at LASIGE and Master's program at the University. It was fantastic to meet you all and be part of a great community of talented students. My parents, Miguel and Fernanda Oliveira for supporting my decision to return to university to pursue a new field and equipping me with the values that have shaped my trajectory. I would also like to thank my sister Edna, your determination and resilience have inspired me. Lastly, my niece Luísa, for bringing me so much happiness and joy during this period of my life.

This work was partially supported by the national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference to PTDC/CC-INF/29058/2017, project SEAL, and LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020).

Resumo

Desde o início da revolução digital com a invenção do transistor em 1947 que marcou o primeiro dispositivo de transferência de dados, quase todas as áreas da atividade humana foram parcialmente ou totalmente deslocadas para o ciberespaço. Isto resultou em um risco significativo de ataques cibernéticos, com impactos em todos os domínios da sociedade humana. Com a pandemia de COVID-19, as organizações em todo o mundo se adaptaram ao trabalho remoto, tornando-as cada vez mais dependentes de aplicações web para operações diárias. O Relatório de Inteligência Global de Ameaças da NTT relatou que os ataques a aplicações web continuam em crescimento, representando 32% dos ataques cibernéticos globais. Esses ataques frequentemente exploram vulnerabilidades no código-fonte das aplicações, muitas das quais são resultado de programação deficiente. Para proteger as aplicações web, é essencial realizar uma programação adequada, implementando funções específicas para proteger pontos de entrada e pontos críticos, ao mesmo tempo em que se evitam funções que possam comprometer o fluxo de dados. Os atacantes, através de um ponto de entrada, são capazes de injetar código malicioso que navega através de uma série de instruções de código numa tentativa de alcançar um ponto sensível da aplicação. Para proteger as aplicações web, deve ser efectuada uma programação adequada. Isto inclui a utilização de funções específicas que protejam os pontos de entrada e os pontos sensíveis e evitem funções que possam comprometer o fluxo de dados.

Os trabalhos que usam aprendizagem automática têm como objetivo a deteção de vulnerabilidades no código-fonte, mas cada um deles utiliza apenas um único conjunto de dados para realizar a experiência utilizando analisadores estáticos e processamento de linguagem natural (PLN) ou uma combinação destes métodos. Além disso, quando a apredizagem automática é utilizada, a abordagem é avaliar o desempenho de um modelo supervisionado versus modelos não supervisionados.

O problema principal que esta tese aborda é o uso de uma arquitectura de apredizamgem automática híbrida para detectar automaticamente vulnerabilidades no código-fonte de aplicações web. No entanto, esta pergunta principal se desdobra em subperguntas de pesquisa que orientaram este trabalho e têm o objetivo de fornecer uma nova contribuição ao trabalho anterior realizado no campo.

O problema de detectar automaticamente vulnerabilidades no código-fonte pode ser desenvolvido ainda mais nas seguintes subperguntas:

- Quais são as melhores características para representar trechos de código-fonte e podemos derivar conjuntos de dados com essas características mas com diferentes níveis de granularidade?

- Qual é uma abordagem adequada de pré-processamento que pode ser aplicada a trechos de código-fonte do mundo real?

- Os modelos de Processamento de Linguagem Natural (Natural Language Processing, NLP) podem fornecer uma seleção de recursos não supervisionada com precisão?

- Qual é uma abordagem adequada de detecção de anomalias de aprendizagem automática (machine learning, ML) para código-fonte?

- Será que uma solução híbrida de ML será adequata para o problema?

Propomos uma arquitetura híbrida, denominada HYVUDE (*Hybrid Vulnerability Detection*), que combina técnicas de NLP e ML para detectar automaticamente vulnerabilidades no código-fonte de aplicações web escritas em PHP. O trabalho realizado nesta dissertação tem como objetivo fornecer um benchmark para o uso de novas abordagens na detecção de vulnerabilidades, em particular, incorporando um tipo de linguagem intermédia baseada em funções-chave definidas a priori e também usando modelos NLP para incorporação de palavras que obtêm características sem conhecimento prévio humano para defini-las. A arquitetura HYVUDE é composta por três componentes principais: extração de uma linguagem intermédia com e sem modelo NLP e classificação de vulnerabilidades com modelos de ML. A linguagem intermédia com e sem NLP é responsável por converter o código-fonte PHP em representações numéricas adequadas para posteriormente serem processadas. Devido à arquitetura do modelo híbrido, abordagens supervisionadas, não supervisionadas e semi-supervisionadas foram usadas, permitindo assim que muitas abordagens fossem experimentadas. O trabalho, também, definiu um método de pré-processamento para limpar trechos de código-fonte para a criação de conjuntos de dados relevantes.

Utilizamos conjuntos de dados do Software Assurance Reference Database (SARD), nos quais obtivemos 3.299 trechos de codigo, tanto vulneráveis e não vulneráveis, onde 1.464 são de SQLI e 1.765 são de XSS. O primeiro passo consiste em identificar corretamente se um trecho de código-fonte é vulnerável e o tipo de vulnerabilidade. Os trechos de código contêm um conjunto distinto de instruções que nem sempre são relevantes para a detecção de vulnerabilidades. Portanto, foi importante definir claramente qual é a melhor representação desse código, por meio de uma linguagem intermédia e também atráves técnicas de NLP, em específico, Doc2Vec.

A partir dos trechos de código, criámos quatro conjuntos de dados, nomeadamente dois binários, cada qual com 19 e 208 características, um de frequência de elementos de código, e o quarto baseado na técnica de incorporacão de parágrafos Doc2Vec.

Foram analisados diversos algoritnmos de ML e pelas métricas de desempenho (precisão, recall e acurácia), tornou-se evidente que certas combinações de modelos ou hiperparâmetros produziram melhores resultados do que outras. A identificação desses padrões e resultados do processo de avaliação levou a ajustes na arquitetura do modelo híbrido e a escolha dos seguintes algoritmos: OCSVM para aprendizagem não supervionada, Random Forest para classicar os trechos de código resultantes como anaomalias do algoritmo OCSVM, e Doc2Vec para processamento de NLP.

Os resultados demonstram uma acurácia geral de 65%, mostrando a validade das abordagens para prever trechos de código vulneráveis a SQLI, XSS ou não vulneráveis. Também demostram métricas de precisão sólidas para identificar vulnerabilidades XSS, enquanto trechos não vulneráveis também apresentaram métricas muito interessantes. Os resultados do modelo Doc2Vec e Regressão Logística, em particular, fornecem uma base sólida para estudos futuros, enquanto o OCSVM é muito limitado pelos dados disponíveis.

A avaliação dos modelos identificou limitações significativas no OCSVM, uma vez que o modelo não foi capaz de identificar corretamente a maioria dos dados vulneráveis. É importante observar que, embora o restante da arquitetura tenha compensado esta limitação, o OCSVM ainda pode ser um modelo relevante a ser explorado em trabalhos futuros devido à sua robustez teórica e uso frequente no espaço de cibersegurança. À medida que os conjuntos de dados se tornarem mais representativos e as características se tornarem mais relevantes, o OCSVM ainda pode ser uma abordagem interessante para identificar comportamentos vulneráveis e não vulneráveis. A Random Forest mostrou resultados sólidos, demonstrando que as árvores de decisão são uma ótima escolha e adaptáveis para resolver muitos problemas em diferentes campos, incluindo a detecção de vulnerabilidades.

Outro ponto otimista foram os resultados positivos do modelo NLP Doc2Vec. Os resultados podem abrir portas a pesquisas futuras nesta direção, como na área de modelos de linguagem grandes (*Large Language Models*) e Inteligência Artificial (IA) generativa. Estes modelos podem fornecer uma solução mais robusta para resolver o problema de detecção automática de vulnerabilidades em código-fonte e prevenção de ciberataques. A acurácia final da arquitetura completa de 65% para todas as classes-alvo mostra que o modelo não sofreu overfitting e, embora possa parecer relativamente baixa, indica que, com uma base de dados limitada disponível, o modelo teve sucesso na previsão da maioria das classes.

Palavras-chave: deteção de vulnerabilidades de web, aprendizagem automática, detecção de anomalias, processamento de linguagem natural, segurança de software

Abstract

Security in web applications is often compromised by poorly written code that is exploited by attackers. Source code vulnerability detection tools have been developed using static analysis and machine learning techniques. The best performing tools seek for very low false negative rates along with acceptable false positives. Static analysis requires manual programming to identify vulnerabilities, depends on human expertise and is usually limited to a specific programming language. On the other hand, classical supervised machine learning approaches previously used may be limited to identify zero-day vulnerabilities or prone to overfit due to limited available datasets.

This dissertation aims to develop a hybrid machine learning (ML) system for vulnerability detection of web applications. The system developed will use a combination of static analysis and Natural Language Processing (NLP) techniques to identify functions related to vulnerabilities that will be used to build representative datasets. The datasets will be used as input for unsupervised machine learning and other behaviour based anomaly detection algorithms in order to signalize as suspicious the code snippets under analysis. For these source code snippets, the system will aim to confirm which are vulnerable and identify the type of vulnerability via supervised machine learning techniques. The dissertation explores a novel approach to vulnerability detection by combining unsupervised anomaly detection models with supervised machine learning and Natural Language Processing techniques. Previous research in vulnerability detection has primarily focused on either unsupervised or supervised methods, neglecting the potential benefits of a hybrid approach. The goal of this research is to investigate the efficacy of hybrid architectures in identifying software vulnerabilities and to determine the optimal machine learning models and datasets for this purpose. The proposed hybrid model consists of different layers. The first uses a One Class Support Vector Machine model (OCSVM) to detect anomalies, the second employs a Random Forest Model to confirm the presence of vulnerabilities on the anomalies. The type of vulnerability is classified by a Logistic Regression Model that relies on the Doc2Vec model for feature extraction.

The research includes experimentation with various machine learning models and datasets, evaluating simple binary features to more complex Doc2Vec embeddings. The thesis demonstrates OCSVM's suitability for semi-unsupervised anomaly detection, yielding promising results across various datasets. Additionally, the study assesses Random

Forests' effectiveness in classifying vulnerable source code snippets based on OCSVM-detected anomalies and validate the use NLP techniques for feature extraction of source-code snippets. Overall, the proposed hybrid model achieved an accuracy of 65%. Although these results seems to be low, this research offers a promising hybrid approach to vulnerability detection, leveraging the strengths of unsupervised and supervised machine learning models. The findings suggest opportunities for further enhancements and optimizations, paving the way for more effective software vulnerability detection systems.

Keywords: web vulnerability detection, machine learning, anomaly detection, natural language processing, software security

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

BinM Binary Main Features Dataset

BinS Binary Single Features Dataset

CBLOF Cluster-Based Local Outlier Factor

CBOW Continuous Bag of Words

CPU Central Processor Unit

D2V Doc2Vec Dataset

DBSCAN Density-based Clustering Algorithm

DT/PT Directory/Path Traversal

FreqM Frequency Main Features Dataset

HYVUDE Hybrid Vulnerability Detection Model

LFI Local File Inclusion

LOF Local Outlier Factor

ML Machine Learning

MLP Multi-Layer Perceptron

MWE Multi-Word Expression Tokenizer

NLP Natural Language Processing

NLTK Natural Language Toolkit

OCSVM One-Class Support Vector Model

OSCI OS Command Injection

PCA Principal Component Analysis

PHP Hypertext Preprocessor

RNN Recurrent Neural Network

RFI Remote File Inclusion

SCD Source Code Disclosure

SQLI SQL Injection

SVM Support Vector Machines

WAP Web Application Protection

XSS Cross-Site Scripting

Chapter 1

Introduction

This document presents the dissertation for a Master's in Data Science in the Faculty of Sciences of the University of Lisbon. This dissertation tackles the domain of Software Security from a Data Science perspective. In other words, this work tackles the problem domain of detecting vulnerabilities in PHP code snippets incorporating data science solutions, which include Natural Language Processing (NLP) and machine learning techniques.

1.1 Motivation

Since the begging of the digital revolution with the invention of the transistor in 1947, the first data transfer device, almost all parts of human activity have been partially, if not fully, moved to the cyber space. As a result, the risk of cyber attacks is a critical issue as its impacts can be felt across all domains of human society. As the COVID-19 pandemic reshaped working culture, organizations throughout the world rushed to have employees working remotely. As a consequence, organizations have continued to become more virtual where web application have been key to ensure the daily operations of companies. The NTT Global Threat Intelligence Report reported that web application attacks continue on the rise, with a reported 32% of overall cyber attacks [11]. These attacks mostly rely on vulnerabilities of the source code of the applications. These vulnerabilities are mostly due to poor programming, where malicious users are able to manipulate and access the data flow. Attackers via an entry point are able to inject malicious inputs that will navigate through a series of instructions of code (source code slices) in an attempt to reach a sensitive asset, known as sensitive sink. To protect web applications, appropriate programming must be done. This includes using specific functions that protect entry points and sensitive sinks, and avoid functions that can compromise the data flow.

Previous work has been done that uses static analysers, Natural Language Processing (NLP), machine learning (ML) or a combination of these methods [1][12][10][13]. The existing work has the objective of detecting vulnerabilities in source code, however each

only uses a single dataset to conduct the experimentation. In addition, when ML is used, the approach is to evaluate the performance of a supervised versus unsupervised models.

1.2 Objectives

The goal of this dissertation is to add to the previous work done by suggesting a new approach that combines supervised and unsupervised models in a hybrid architecture that is tested with different types of datasets. For a tool to be useful, it must have low false positive rates with a fair trade off in regards to false negatives. We aim to contribute to current research work by proposing a method that is inspired by hybrid detection architectures used for network traffic anomaly detection. With the aid of NLP and static analysis techniques, certain key functions will be identified in the source code slices. This will be translated into different representation vectors, which will be used to build various datasets that will be incorporated in a hybrid anomaly detection system.

The proposed hybrid model consists of two main layers. In the first layer, an semi-unsupervised model, specifically the One-Class Support Vector Machine (OCSVM), is trained to detect anomalies within vulnerable patterns. Feature quality improvements using autoencoders and manifold learning techniques are also explored to enhance the performance of the OCSVM model. The data points classified as anomalies are considered suspicious and passed to the second layer. The second layer employs a supervised machine learning model, Random Forest, trained on datasets containing labeled vulnerable and non-vulnerable data. These supervised models aim to confirm non-vulnerable code. Those data points that are considered vulnerable by OCSVM first layer are identified and processed using a unsupervised word embedding technique D2V that creates numeric features that are then used on a supervised Logistic Regression model to identify their vulnerability types, such as SQL Injection (SQLI) or Cross-Site Scripting (XSS).

The thesis presents preliminary results demonstrating the suitability of the OCSVM model in a semi-unsupervised context for anomaly detection. The OCSVM is evaluated on different datasets, that are described in the work, with promising results. The dimensionality reduction techniques, particularly UMAP, show potential in improving the model's performance without significantly increasing computational costs.

Furthermore, the study evaluates the effectiveness of Random Forests in classifying vulnerable source code snippets based on the anomalies detected by the OCSVM. The choice of hyperparameters, such as the number of decision trees and split criteria, is discussed. The final classification results indicate an overall accuracy of 65%, with strong precision metrics for identifying XSS vulnerabilities. The current work will use code slices written in PHP, we hope however that forthcoming work may evolve into using *code neutral features* that may be applicable to other languages. This will be enabled with transferred learning introduced by Medeiros et al [12], a process where key func-

tions for vulnerability detection are identified in different programming languages and thus enable the datasets of PHP source code used to be suitable for other languages.

1.3 Contributions

The main contributions of this research work are the following:

- The creation of new datasets that can be used as input to test ML models. This work differentiates itself from previous research by experimenting with more than one dataset. As it will be discussed in later sections of the paper, the datasets used will be diverse in type as binary, high dimensional and other NLP numeric representations of semantics (Doc2Vec) will make up the datasets.
- A hybrid ML model architecture for the detection of vulnerabilities in source code.
- An unsupervised NLP based approach to converting PHP code into intermediate language representation.
- Experimental evaluations, including hyper-parameter tuning of the ML models used for classification and embedding.

1.4 Structure of the Document

This document will be organized by the following chapters:

- Chapter 2 – Background and Related Work. In this chapter, we tackle the theoretical background on all topics and techniques that will be incorporated into the final architecture followed by a summary of previous related work.
- Chapter 3 – HYVUDE - Hybrid Vulnerabilty Detection. Here, we introduce our solution and guide through the steps necessary and the justifications for them to reach our final classification for vulnerability detection.
- Chapter 4 – Implementation. Here we provide a detailed explanation of the of all the operations, including pre-procession, data cleaning and data flow in the architecture models.
- Chapter 5 – Experiments. The experimentation section is divided into each model individually used in the hybrid architecture along with a final experiment with the full architecture.
- Chapter 6 – Conclusion. This section includes a final observation of the research work carried out along with recommendations for future work.

Chapter 2

Background and Related Work

2.1 Input Validation Vulnerabilities

The most relevant causes of security breaches arise from user input actions. To prevent this, programmers should use appropriate source code to protect the web application from user input violations. Flaws in the source code that provide the opportunity for malicious user input are referred to input validation vulnerabilities [12]. We will briefly describe the different types of input validation vulnerabilities common amongst web application. This section, will outline the vulnerabilities considered, and it will draw from previous tools such as WAP and DEKANT [1][12]. The vulnerabilities considered are SQL injection (SQLI), cross site scripting (XSS), remote file inclusion, directory / path traversal (RFI), local file inclusion (LFI), source code disclosure (SCD), OS command injection (OSCI) and PHP code injection. It is important to note, however, that the current state of our datasets only have SQLI and XSS vulnerabilities labelled, due to this reason the system will be limited to identifying these vulnerabilities by type.

SQL injection (SQLI) occurs when a malicious user uses string building techniques to modify the structure of a SQL query. This attack has the aim of reading from or writing to the database in a unexpected manner. Medeiros et al [1] demonstrated how an SQLI is carried out by the user as metacharacters and normal characters are mixed to modify a query structure, and gain access to confidential information from the database. SQLI can be avoided with preventive programming, as inputs can be sanitized by removing or encoding metacharacters provided by the user or with the use of prepared statements.

```
1: $conn = mysql_connect("localhost", "username", "password");
2: $user = $_POST[ 'user' ];
3: $pass = $_POST[ 'password' ];
4: $query = "SELECT * FROM users WHERE username = '$user'
             AND password = '$pass' ";
5: $result = mysql_query($conn, $query);
```

Figure 2.1: Source Code Snippet in PHP with SQLI vulnerability, adapted from [1].

Figure 1 demonstrates an SQLI vulnerabilty as a malicious user may use metacharacters to manipulate the SQL query in line 4 of the username and password provided in lines 2 and 3. For example, the malicious user may provide a username as *admin' –*, with the later metacharacters manipulation the SQL query script to return information on the admin without the need for a password. This vulnerabilty can be prevented by the appropriate use of sanitization functions (e.g., *mysqli_real_escape_string*) that prevent metacharacters from being used or by only accepting prepared statments [1]. These functions will be discussed further in a later section of the paper (see Section 5.1).

Cross Site Scripting (XSS) occurs as an injection of malicious code into a victim's browser. The browser does not detect the malicious script and executes it. The attacker can then have access to sensitive information and even be able to change content of a HTML page. As shown in Figure 2, an XSS vulnerable source code may only require a line such as *echo $_ GET['username'];* [1]. A malicious entity can convince a user to click on a link that accesses the web application and sends a malicious script that is displayed by the *echo* function and then executed in the users browser.

```
$var = $_GET[ 'username' ] ;
echo $var ;
```

Figure 2.2: Source Code Snippet in PHP with XSS vulnerability, adapted from [1][2].

Remote File Inclusion (RFI) is when a file with malicious code is included in a script. Language including PHP allow for files to be included in scripts, which can be a vulnerability if a file name is taken as user input. When this occurs, the execution of a remote file can be executed on the server, containing malicious code. On the other hand, local file inclusion (LFI) differs from the fact that it inserts a file from the system of the web application and not a remote one. Directory/path traversal (DT/PT) allows for an attacker to access and read files outside the web application directory [12]. Similarly, source code disclosure (SCD) allows the attacker to access source code and files and obtain valuable information for further attacks. OS command injection (OSCI) makes the application execute a command on the shell defined by the attacker. PHP code injections will also be considered. These attacks rely on the PHP *eval* function. This function is used to assess PHP string as PHP code, the attacker targets this function by inserting a malicious script.

2.2 Taint Analysis

Taint analysis is a code static analysis technique that has the goal of tracking code features along the source code and verify if they reach some point of the code. For a vulnerability

to exist, there is a need for a data flow that begins at an entry point and reaches a sensitive sink. A sensitive sink represents the vulnerable point in the code that a malicious user can exploit. In this sense, taint analysis is used to find vulnerabilities, by tracking entry points until reaching some sensitive sink.

Previous work in this topic, which will be further discussed on the related work section, have identified functions that can detect an entry point and a sensitive sink. A tainted function at the sensitive sink, refers to a data flow that can be compromised, and thus vulnerable. On the other hand, sanitization functions can be used to correct or untaint a data flow and thus protect the web application. For example, in the case of an SQLI attack, a sanitization function can be used at the sensitive sink level that deals with user inputs that may be malicious [1]. This means, for instance in Figure 1, the sanitization function *mysqli_real_escape_string* if applied to lines 2 and 3, the user inputs will be protected and thus such vulnerability will be removed.

2.3 Tokenization

Tokenization is the process of Natural Language Processing (NLP) for breaking down a sequence of words, including meta-characters into pieces called tokens [14]. A token will be a piece of a sequence of characters that hold a semantic meaning; usually the tokenization process will remove other characters such as punctuation as these are irrelevant for the next steps (known as stop words). For better performance, it is necessary for pre-processing, such as removing stop words or certain meta-characters in the text to optimize the performance of the Tokenizer. In this work the Natural Language Toolkit library (NLTK) will be used for the tokenization process. It is also important to note that this library contains the option to create new tokens that combine what would be separate tokens. This is done with the Multi-Word Expression Tokenizer (MWE) available in the NLTK library [14]. The below example, Figure 3, further demonstrates the tokenization process of an english sentence. Note that this example demonstrates the tokenization process without any pre-processing such as removal of stop words and other non-alphabetic characters.

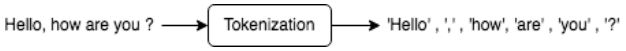

Figure 2.3: Example of Tokenization process of an English sentence.

2.4 Machine Learning

Machine Learning (ML) aims to identify patterns previously learnt from a dataset, and use these patterns to provide predictions on future data or remove uncertainty from a decision-

making process [5]. To achieve this, ML relies on features that are data points used to describe the objects involved in the problem at hand. A task must then be assigned that accurately represents the problem we want to solve [3]. For example, in the case of this work, a suitable task is to identify a source code snippet as one of two classes (vulnerable or non-vulnerable). The algorithmic process that provides an output based on the data points depends on the model used. The models considered in this work will be further discussed below.

2.4.1 Supervised Learning

A supervised approach requires the input data to have a labelled output. In other words, the model learns the relationship between input and output pairings. The labelled output is referred as ground truth and requires a knowledgeable "supervisor" to ensure the correct labelling. Classification maps inputs into outputs, where the number of possible outputs accounts for the number of classes. For example, a source code snippet could either belong to one of two classes (vulnerable or non-vulnerable) or belong to one of many classes for each type of vulnerability (SQLI, XSS, RFI, etc.). However, such classification depends on how the learned dataset was constructed. The following methods will be explored in this paper.

2.4.2 Decision Trees

Decision Tree is a commonly used machine learning algorithm due to it's applicability to solve tasks and structure that is easy to interpret. A decision tree continuously reduces the hypothesis space where the main goal is to learn decision rules based on a splitting criteria. In other words, a tree model is organized by a hierarchy of conditions where each leaf is an outcome. The structure of the model consists of internal nodes (nodes that are not a leaf) that are labelled with a feature. From each internal node a set of literals are derived from it to determine a split. The leaf contains the outcome as a logical expression of all the literals along the path of the tree, from the root to the leaf [3]. If a set of instances in the data are homogeneous then a label (class) can be assigned to these instances, if otherwise a criterion must decide on the best split to define the path of the decision tree.

The criterion is an impurity measurement which defines the best split for the nodes and determines the root of the tree, as the purest first. Impurity can be defined as the empirical probability of a class. For example, let's consider Boolean features $F1$ and $F2$ and that can belong to a positive $C+$ and negative $C-$ class. If all $F1+$ are equal to the positive class $C+$ and $F1- = \emptyset$, then this node split is pure. However, most datasets are not able to provide pure splits and therefore the best split must be decided. The most two most common best split criteria used are the Gini Index and Entropy measurements [3].

When using Gini Index Impurity measurement as a criteria for best split, the impurity of each tree leaf is calculated. This criteria is the expected error if the examples in the leaf are chosen randomly, and can be mathematically expressed as $2p(1-p)$, where the probability for positive (vulnerable in this context) label is p and negative (non-vulnerable) $(1-p)$ [3]. When using a decision tree algorithm with Gini Index as the best split criteria, the lowest gini values will indicate the next best split. On the other hand, Entropy measures a bit value that defines the expected information that is required to classify an instance in a given leaf [3].

Entropy measurements are amongst the main algorithms used for decision trees. Machine learning researcher J.Ross Quinlan introduced decision tree algorithms that became widely adopted in machine learning during the late 1970s and early 1980s with the algorithm ID3 and its successor C4.5 [15]. These algorithms create a decision tree based on entropy measurements. For example, ID3 uses a concept called Information Gain to calculate the best split. Authors Han, Kamber and Pei [16] best describe the process of this algorithm with the following notation.

- Let D is the training data (a set of tuples with a labelled class).
- The labelled class feature has n distinct values, $C_i(i = 1, ...n)$.
- C_i, d represents the set of tuples that are part of class C_i in the data D.
- $|D|$ represents the number of tuples in the data D and $|C_i, d|$ represents number of tuples of tuples in C_i, d.
- N is the node that represents the tuples in data partition D.
- p_i is the non-zero probability that a random tuple from D belongs to class C_i. This probability can be calculated by $\frac{|C_i,d|}{|D|}$.

Using this method, the feature with the highest calculated Information Gain is used as the split for node N. When building a decision tree in this manner, first a average Information Gain is calculated that provides the average information that is needed to classify a random tuple in D.

$$Info(D) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

The following step, takes into consideration all possible values of a feature A that can be seen on the training data. In other words, from the tuples on data D, feature A can have m distinct values $a_1, ..., a_m$ that can be used to split the D into m subsets $D_1, ..., D_m$. For example, subset D_i contains the tuples of D that contain feature value a_i of A. The subsets are used to grow the tree via branches. The information value that is expected to classify

a tuple from D based on the subsets from the values of feature A can be calculated as below.

$$Info_A(D) = \sum_{i=1}^{m} \frac{|D_i|}{|D|} \times Info(D_i)$$

Value $\frac{|D_i|}{|D|}$ acts as a weight for the ith subset of D and the smaller the value of $Info_A(D)$ the higher purity of the partitions. With this approach, the final decision for the feature that will be the best split for the node N is defined by the Information Gain for each feature. The feature that has the highest Information Gain value is chosen as the next best split since this means that the decision tree splits on the feature A that will best classify the data so that the remaining information required to classify the remaining tuples is reduced [16].

$$Gain(A) = Info(D) - Info_A(D)$$

The process described above is used for the ID3 algorithm, however Quilan pointed out that this algorithm has a bias towards choosing features that have a higher number of possible values [15]. This can result on decision trees to opt for the least efficient path to a classification. To overcome this, it's successor, algorithm C4.5 takes into consideration the amount of possible values of each feature to prevent this bias. It does so by introducing a new concept called Gain Ratio [3]. Firstly, a new value that represents the information based on the number of outcomes of feature A is calculated. This value is referred to as Information Split.

$$InfoSplit_A(D) = -\sum_{i=1}^{m} \frac{|D_i|}{|D|} \times log_2(\frac{|D_i|}{|D|})$$

The Gain Ratio for feature A is then calculated. The feature with the greatest Gain Ration is used as the best split.

$$GainRatio(A) = \frac{Gain(A)}{InfoSplit_A(D)}$$

In 1984, Breiman et al [17] introduced the CART algorithm for binary decisions trees using Gini Index as the impurity measurement of the data.

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2$$

A binary decision tree using the Gini Index grows by considering all binary splits for a feature. This means that for every value in feature A, all the subsets are considered. For example, if A has m different values, then the algorithm considers $2^m - 2$ subsets [16].Two subsets are subtracted as they do not represent a split (these are the subsets that

contain all possible feature values and none). As the Entropy approach, the Gini Index is calculated for each feature and for the training data. For each feature it is calculates the weight of the partitions of D. If we consider feature A to have 3 binary partitions of D, the Gini Index of A is calculated as follows.

$$Gini_A(D) = \frac{|D_1|}{|D|}Gini(D_1) + \frac{|D_2|}{|D|}Gini(D_2) + \frac{|D_3|}{|D|}Gini(D_3)$$

The feature that has the lowest Gini value is selected as the splitting feature, as it increases the purity of split [16] [17].

2.4.3 Support Vector Machines

Introduced by Vladimir Vapnick et al [4] and developed at AT&T Bell Laboratories, Support Vector Machines (SVM) are a supervised two-class classification and regression model. SVMs separate the data points linearly by a decision boundary also known as a hyperplane. The support vectors are the data points closest to the decision boundary for each class. If the data points are not linearly separate, an approach called the kernel trick is applied to increase the dimensional space until the classes are separable. [3] [18].

A simple linear classification is based on a decision boundary that can separate two-classes with a linear hyperplane. This hyperplane is a linear line that intercepts the halfway between the center of the data points of the classes. Noting that the input is a vector $\{\mathbf{x}_i, y_i\}$ $i = 1, \ldots, n$ and $y_i \in \{-1, 1\}$ represents the two possible classes, the decision boundary (hyperplane) can be described as the linear equation:

$$\mathbf{x}_i\mathbf{w} + b = 0$$

The objective therefore is to find the optimal hyperplanes where the data can be separated by its according class. To find the optimal hyperplane, firstly to parallel hyperplanes calculated, these intercept the data points that are called the support vectors for both the positive and negative classes as they are closest to the final hyperplane and decision boundary. These hyperplanes can be described as below for the positive class,

$$\mathbf{x}_i\mathbf{w} + b = +1$$

and for the negative class,

$$\mathbf{x}_i\mathbf{w} + b = -1$$

The slope of the regression line increases with higher separation between the classes and decreases if the class distribution is skewed [3]. The *margin* is the distance between the nearest support vector of each class and can be described as $\frac{2}{||w||}$. The main goal is to maximize the margin, or in other words increase $||w||$. In addition, to correctly classify

the data points, for every $i \in (1, n)$, x_i and y_i must be meet the following conditions:

$$\mathbf{x}_i\mathbf{w} + b \geq +1, \quad y_i = +1 \tag{2.1}$$

$$\mathbf{x}_i\mathbf{w} + b \leq -1, \quad y_i = -1 \tag{2.2}$$

$$\equiv \tag{2.3}$$

$$y_i(\mathbf{x}_i\mathbf{w} + b) - 1 \geq 0, \qquad \forall i \tag{2.4}$$

Figure 2.4 provides a better illustration, showing the optimal margin defined as $\frac{2}{||w||}$, along with the decision boundary hyperplane and the support vectors in blue.

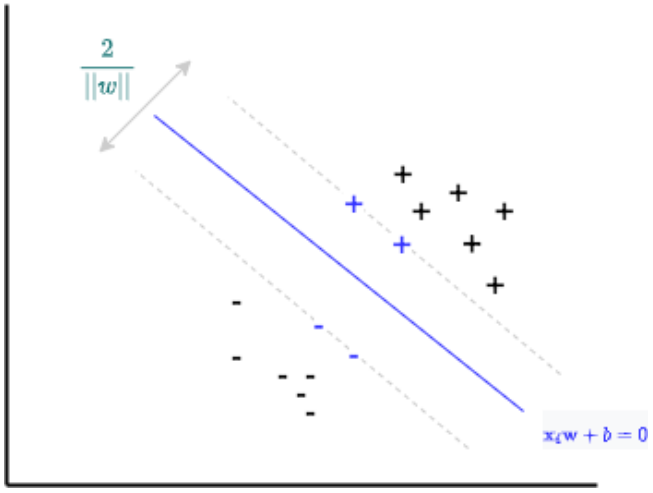

Figure 2.4: Example of linear SVM in a 2-D space, with support vectors in blue, adapted from [3] [4].

In SVMs the mathematical optimization problem of maximizing the margin while correctly classifying the data points is solved using the Lagrange Multiplier concept. Optimization can be defined as finding the best value among a set of alternatives, as in this case maximizing the margin or the equivalent finding of the support vectors. Lagrange multipliers are useful to solve constrained optimization problems where the alternatives have restrictions. In this case being minimizing $\frac{\|w\|^2}{2}$, so that, $y_i(\mathbf{x}_i\mathbf{w} + b) - 1 \geq 0$. This process and solving this mathematical problem is the training process of the SVM [3].

The optimal hyperplane $\mathbf{x}_i\mathbf{w} + b = 0$ separates the data points with a maximal margin, thus maximizing the distance between the training vectors of the two classes as illustrated in figure 2.4. This distance $\rho(w, b)$ can be defined as:

$$\rho(w, b) = \min_{x:y=1} \frac{x \cdot w}{|w|} - \max_{x:y=1} \frac{x \cdot w}{|w|} \tag{2.5}$$

The arguments (w_0, b_0) of the optimal hyperplane maximizes this distance and can be described following (2.4) and (2.5) as:

$$\rho(w_0, b_0) = \frac{2}{|w_0|} = \frac{2}{\sqrt{|w_0| \cdot |w_0|}} \tag{2.6}$$

The optimal hyperplane minimizes $w \cdot w$ under the constrains (2.5), this quadratic optimization equation is solved using a Lagrangian:

$$L(w, b, \Lambda) = \frac{1}{2} w \cdot w - \sum_{i=1}^{l} \alpha_i [y_i (x_i \cdot w + b)] - 1, \tag{2.7}$$

- Where $\Lambda^T = (\alpha_1, ..., \alpha_l)$ is the vector of non-negative Lagrange multipliers corresponding to the constraint at (2.4).
- The solution to this optimization problem is the minimax (where the slopes in all directions are zero) of the Lagrangian in the $2l + 1$ dimensional space of w, Λ and b [4].
- The minimum is taken with respect to $w = w_0$ and $b = b_0$ and maximum to the Lagrange multipliers Λ.

$$\frac{\partial L(w, b, \Lambda)}{\partial w} = (w_0 - \sum_{i=1}^{l} \alpha_i y_i x_i) = 0, \tag{2.8}$$

$$\frac{\partial L(w, b, \Lambda)}{\partial b} = (w_0 - \sum_{\alpha_i} y_i \alpha_i) = 0. \tag{2.9}$$

From (2.8) we can derive the equation below that defines the optimal hyperplane as linear combination of the training vectors:

$$w_0 = \sum_{i=1}^{l} \alpha_i y_i x_i, \tag{2.10}$$

From (2.10) and (2.9) we can express (2.7) as:

$$W(\Lambda) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} w_0 \cdot w_0 \tag{2.11}$$

$$= \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j x_i \cdot x_j \tag{2.12}$$

When dealing with the quadratic dual optimization problem the Karush-Kuhn-Tucker (KKT) theorem states that the Lagrange multipliers α_i^0 and its constraint can be equal [3] at the minimax w_0, b_0, Λ_0 [4].

$$\alpha_i[y_i(x_i \cdot w_0 + b_0) - 1] = 0, \quad i = 1, \ldots, l.$$

This is solved by the fact that when $\alpha_i \neq 0$ this inequality is treated as an equality due to the KKT theorem. Therefore, we can find the support vector vectors x_i for which:

$$y_i(x_i \cdot w_0 + b_0) = 1$$

Soft Margin SVM is an approach used that allows for a solution to be found even if the data is not linearly separable. With this approach, slack variables ϵ_i for each training instance that allows for some to be inside the margin or even incorrectly classfied. [3]. The optimization problem is now:

$$\phi = \frac{1}{2} w \cdot w + C(\sum_{i=1}^{l} \epsilon_i)^k, \quad k > 1 \tag{2.13}$$

Under the constraints;

$$y_i(x_i \cdot w + b) \geq 1 - \epsilon_i, \quad i = 1, \ldots, l \tag{2.14}$$

$$\epsilon_i \geq 0 \quad i = 1, \ldots, l \tag{2.15}$$

In the equation (2.13), C is a parameter that is defined apriori by the user, and acts as a trade-off between margin maximization and slack variable ϵ_i minimization. In other words, the higher the C value the lower the tolerance towards margin error and miss-classifications. This value is refered to as the complexity parameter, as lower C values will decrease the complexity of the SVM, allowing for more erros to maximize the margin [3].

The SVM processes described above deal with linear decision boundaries. However, SVMs are able to go beyond linearity, creating decision boundaries that are not linear through a technique called the kernel trick. As briefly mentioned earlier in this section, when data is not linearly separable in an input space, a kernel can be used to increase the dimensions until the data can be separated linearly in this new dimension feature space, for example:

- The original input space is 2-Dimensional $x = \{x_1, x_2\}$ where the data points are not linearly separable.
- A transformation function changes the input space to a new feature space in 3-D to increase the separability of the data:

$$\phi(x) \rightarrow x_1^2, x_2^2, \sqrt{2x_1x_2} \tag{2.16}$$

The Kernel trick consists of the following computation for a 3-D feature space:

$$k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle \tag{2.17}$$

$$\langle \phi(x_i), \phi(x_j) \rangle = \langle \{x_{i1}^2, x_{i2}^2, \sqrt{2x_{i1}x_{i2}}\}, \{x_{j1}^2, x_{j2}^2, \sqrt{2x_{j1}x_{j2}}\} \rangle \tag{2.18}$$

$$= x_{i1}^2 x_{j1}^2 + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{12}x_{j1}x_{j2} \tag{2.19}$$

Similar calculations are applied to increase the input space for higher dimensional feature spaces. For an infinitive feature space, a Gaussian Kernel is most commonly used, where σ is a parameter that is referred to as bandwidth and scales the width of the kernel [3]:

$$k(x_i, x_j) = \exp(-\frac{||x_i - x_j||^2}{2\sigma^2}) \tag{2.20}$$

2.4.4 Logistic Regression

Logistic regression, is similar to linear regression however it is used for a classification task, where a threshold probability determines whether or not it belongs to a specific class [5]. In other words, it measures the probability of an event (e.g belonging to a certain class) based on a dataset with independent variables [3].

As the output is based on a probability and therefore values of $y \in \{0, 1\}$ logistic regression has two main differences towards linear regression. Firstly it assumes a *Bernoulli* distribution of y instead of *Gaussian* as it most suitable for binary outputs. Taking into consideration the notation introduced in the previous section we can describe the new distribution.

$$p(x|\boldsymbol{x}, \boldsymbol{w}) = Bern(y|\mu(x)) \tag{2.21}$$

- where $\mu(x) = E[y|\boldsymbol{x}] = p(y = 1|\boldsymbol{x})$

Secondly, the linear calculations of the independent variables are passed through a sigmoid function that maps the output between $\{0, 1\}$, creating an S-Shaped curved which is a trademark characteristic of logisic regression. This function is defined as:

$$sigm(\eta) = \frac{1}{1 + \exp(-\eta)} = \frac{e^\eta}{e^\eta + 1} \tag{2.22}$$

Therefore, as the sigmoid function is passed $\mu(x) = sigm(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x})$ it ensures that $0 \leq \mu(x) \leq 1$. We can then define logistic regression as:

$$p(x|\boldsymbol{x}, \boldsymbol{w}) = Bern(y|sigm(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x})) \tag{2.23}$$

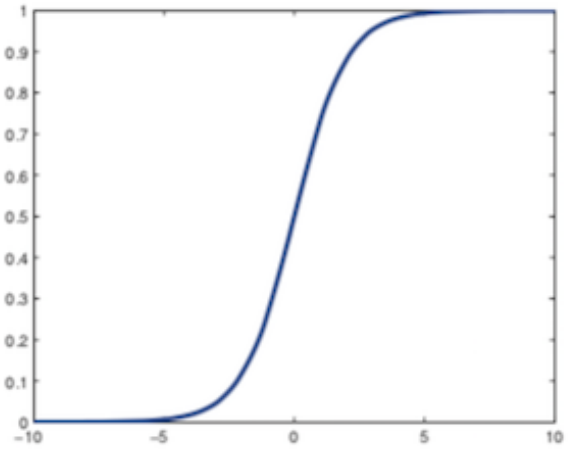

Figure 2.5: Sigmoid Function[5].

Figure 2.5 demonstrates the sigmoid function where $sigm(-\infty) = 0$, $sigm(0) = 0.5$ and $sigm(\infty) = 1$.

2.4.5 Unsupervised Learning

Unlike supervised learning where the models are trained with input data that is labelled with a target output, unsupervised learning only uses input data. In fact, unsupervised learning is similar to human learning, as learning is focused on the input variables and not on the desired output that does not contain much information in itself [5]. For this work, the focus will be on unsupervised knowledge discovery models, mainly clustering algorithms that group similar data together. It is important to note, that the way in which similarity is defined is dependent on the algorithm used. Below we will address models relevant to our work.

K-means is commonly used due to its simplicity and performance. For a set of observations, these will be grouped into k clusters (where k must be smaller than the number of observations) [7]. Random k centroids will be assigned to the data points. A similarity measure will cluster each data point to its nearest centroid based on, most commonly, the Euclidean distance. The centroid of each k cluster is then recalculated and the process repeated until convergence is achieved by the minimization of the sum of the squarer error (SSE). Metrics such as the silhouette coefficient [3] can evaluate the performance of the clustering algorithm and if an appropriate k number was used.

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [19][20] discovers both clusters and noise in a feature space. Points in the same cluster with be found in higher density spaces, and outliers will be found on lower density ones. The algorithm relies on two parameters, Epsilon and MinPts. Epsilon defines a density threshold for a point to be considered as part of cluster while MinPts defines the minimum number of points needed for a grouping to become a cluster. If these parameters are not defined a priori, the algorithm will randomly find a point and determine if it is a point at the border of a cluster, if so it will move on until it finds a point in a central region of a cluster, also referred to as core point. Once all data points are covered, the clusters are defined and outliers (noise) points are grouped also as a separate cluster.

2.4.6 Ensemble Learning

Ensemble learning refers to the combination of ML models to improve performance. Instead of training one learner to solve a task, ensemble methods train multiple learners for the same task [21]. The objective of this approach is to reduce the impact of random fluctuations in single models, by training many models with small differences from the same training data [3]. Next, we will discuss three main approaches that create the necessary model diversity for ensemble learning.

Bagging

This method achieves model diversity by training each model with random samples from the original data. In more detail, a technique called *boostrap sampling* is used where a dataset with x number of training examples is generated from the original dataset with same x number of training examples. However, some of the training examples from the original dataset will appear more than once, while other examples will not appear in the generated training sample [21]. In short, samples are uniformly taken from the original dataset with replacement. Each *boostrap* sample is used to train a sub-model, thus creating diversity on the ensemble. The prediction output of the ensemble model may be the average prediction across all sub-models or by voting where the majority prediction wins [3]. Random Forests, are an ensemble method, where many different decision trees are generated to create a final classification model through a voting process, where the most common prediction becomes the model's final prediction. Similar to *boostrap sampling*, each tree is also built with a random subset of features, increasing the diversity of the model while also reducing the computation time for each tree [3].

Boosting

This approach uses different method for creating diversity in the ensemble model. Similar to bagging, it diversifies the training set, but it does so by attributing weights to miss-classified samples and using it to train the following sub-model [3]. The objective, is to sequentially improve learners, so that the final output is a combination of the improved prediction of all the models. An example of boosting is the *Adaboost* algorithm. This algorithm begins by using a weak learner on the original dataset, and consequentially using copies of this weak learner on the dataset but with adjusted weights on the miss-classified instances. This adjusts the model to focus more on instances which are harder to correctly classify [22].

Voting and Mega-learning

Lastly, voting can be the method of diversifying the ensemble learning by diversifying the algorithmic models, rather than the dataset. This could then lead to a mega-learning approach, where the performance of each model used in the ensemble are used as features to then decide what combination of algorithmic models are mostly suitable for the task [3].

2.4.7 Anomaly Detection

Anomaly Detection is a technique used to identify data patterns that are significantly different from the normal behaviour patterns built in a model [23]. This technique is used extensively in cyber security, in particular when looking for outsider and insider malicious activity in network traffic. Anomaly detection approaches must train a model with a representative dataset of normal patterns of behaviour so that it may accurately identify those that deviate from it. It is important that a low false positive rate is achieved so that the model can be trusted. Clustering is usually used for anomaly detection, where clusters of normal behaviour data are created and outliers can be identified [24]. Nevertheless, there are a few other algorithms used specifically for anomaly detection that we will explore.

Isolation Forests use a different approach to anomaly detection models that focus on profiling normal behaviour pattern to then identify patterns that deviate. Instead, isolation forests focus and are optimized on the anomalies themselves. Anomalies, in most cases, as a deviation from the norm should contain characteristics that are less frequent and therefore are more isolated from other normal instances. Because of this, isolation forests seek to create a tree structure on every single data point, where the most isolated instance are closer to the root of the tree and normal instances are deeper in the tree. Isolation

Forest works by creating an ensemble of trees for a data set where anomalies are identified based on their average path length with the trees [25].

Local Outlier Factor (LOF) is a model that provides an anomaly score to each point on the dataset based on the assumption that anomalies are more distant from their "normal" neighbours. This score, called the Local Outlier Factor is the measurement of the local deviation of the density of a given data point in regards to its neighbouring points. The score therefore depends on how isolated the data point is from its surrounding neighbours [26].

Cluster-Based Local Outlier Factor (CBLOF) applies a similar approach to LOF but based on clustering techniques. Based on the work from Zengyou He et al [27] it assumes that normal data patterns will be located in the bigger clusters, while outliers will be found on smaller clusters. The algorithm first finds the clusters and defines smaller and bigger clusters. The CBLOF score calculates the distance of each point to the closest cluster centroid (for smaller clusters) or the distance of each point to its cluster centroid (for bigger clusters), multiplied by the size of the cluster. A higher score indicates that it is an anomaly [27].

Anomaly Detection - One Class Support Vector Machine

OCSVM (One-Class Support Vector Machine) like the SVM model previously mentioned, uses support vectors to define separation between classes [28]. However, in this case the data given to train the model only belongs to the normal behaviour class. In this case, the support vectors deduce normal behaviour from outliers that represent anomalies.

The model requires that the data is linearly separable. However, this is mostly not the case and therefore a kernel is used (known as the kernel trick) which increases the dimension of the feature space and solves this problem as explained with SVMs. This allows the model to separate data points in a non-linear way. The kernel trick is able to solve the problem of non-linear mapping while persevering the geometric distances and properties of the data points [29]. A transformation function $\phi(.)$ defined by the kernel projects the data points into a higher dimensional feature space. A hyperplane is calculated that separates the data points of the target class from the origin. This hyperplane is the decision boundary that separates normal behaviour from anomalous behaviour which is closer to the origin. The figure below illustrates this further.

In more detail, the OCSVM computes a binary function that represents the input space and density probability, where the function maps the data so that the density probability will be found mostly in non-zero values. Introduced by Scholkopf et al. [28] the algorithm is described in the following manner:

- Let's consider the training data $x_1, \ldots, x_l \in X$.
- $l \in N$ is the number of observations in the set X.

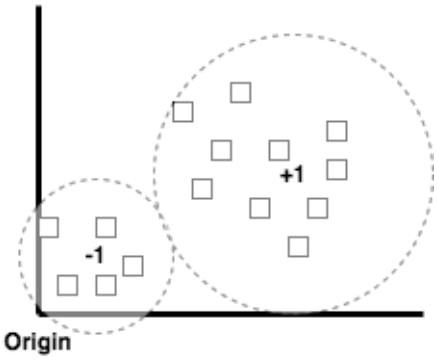

Figure 2.6: OCSVM Example.

- A kernel computes and maps out the data into a dot product space F to increase the dimension of the feature space.

$$k(x, y) = (\phi(x) \cdot \phi(y)) \tag{2.24}$$

The algorithm returns a function that as previously mentioned outputs values $+1$ for the denser region capturing the majority of the data points and -1 for the remaining data points considered outliers. It does so by mapping out the data in the new feature space via a kernel, and maximing the margin between the points closer and further away from the origin as illustrated on figure 2.6. It does so by solving the following quadratic equation:

$$\min_{w \in F, \boldsymbol{\xi} \in R^{\ell}, p \in R} \frac{1}{2}||w||^2 + \frac{1}{v\ell} \sum_i \xi_i - p \tag{2.25}$$

Under the constraints;

$$(w \cdot \phi(x_i)) \geq p - \xi_i, \quad \xi_i \geq 0 \tag{2.26}$$

Where,

- For every new x point, the calculated value $f(x)$ is given by the side of the hyperplane the new x point is in the kernelized feature space.
- The decision function $f(x) = sgn((w \cdot \phi(x_i)) - p)$ is positive for the majority of the data points in the training data, where the margin from the support vector is small, therefore $||w||$ is small.
- $v \in \{0, 1\}$ is a parameter that is chosen apriori and controls the trade-off between the size of $||w||$ and the amount of data that is positive from the decision function.

Based on the dual problem (2.25 and 2.26) the solution of it can be derived including the kernel (2.24), the decision function $f(x)$ can be expressed as a Support Vector expansion where the instances x_i have a non-zero α_i are Support Vectors.

$$f(x) = sign(\sum_i \alpha_i k(x_i, x) - p) \tag{2.27}$$

Another approach to OCSVM was introduced by Tax and Duin et al.[6] where instead of the hyperplane approach that separates points closer and further from the margin, the authors propose a spherical interpretation by constructing a hyper-sphere instead. This approach is referred to as Support Vector Data Description (SVDD). With this approach all data points in the training set are associated to the positive $+1$ class and a hyper-sphere is constructed surrounding these points. If a data point in the test set is placed outside the hyper-sphere, then the data point is considered an outlier. As with the hyperplane approach, a kernel can be used to increase the dimension of the feature space.

The hyper-sphere has a radius based on the center $\boldsymbol{a}$ that is defined as R where $R > 0$. The model minimizes the size of the sphere by minimizing R^2 [6] and forces all the training data points x_i to be inside this sphere. Similar to the SVM approach the error function is defined to minimize R^2 as:

$$F(R, \boldsymbol{a}) = R^2 \tag{2.28}$$

Under the constraints,

$$||x_i - \boldsymbol{a}||^2 < R^2, \quad \forall_i \tag{2.29}$$

As most of the training data points must fall within the sphere, while allowing some outliers to be present in the training set, the distance between a point x_i and the center of the sphere $\boldsymbol{a}$ is not exactly less than R^2. Nevertheless, large distances between x_i and $\boldsymbol{a}$ are penalized via the use of slack variables $\xi_i \geq 0$. This changes the quadratic problem (shown in 2.28 and 2.29) to the below:

$$F(R, \boldsymbol{a}) = R^2 + C \sum_i \xi_i \tag{2.30}$$

Under the constraints that most data points are within the sphere,

$$||x_i - \boldsymbol{a}||^2 \leq R^2 + \xi_i, \quad \xi_i \geq 0 \quad \forall_i \tag{2.31}$$

Parameter C defines the trade-off between the volume of the sphere and the amount of errors it tolerates. This quadratic problem can then solved as described in the SVM section via the introduction of Lagrange multipliers where the support vectors are defined as well as the hyper-sphere. Figure 2.7 aims to illustrate this approach, where the training data is within the sphere, the data points on the border of the sphere are the support vectors and point x_i is an outlier where $\xi_i > 0$.

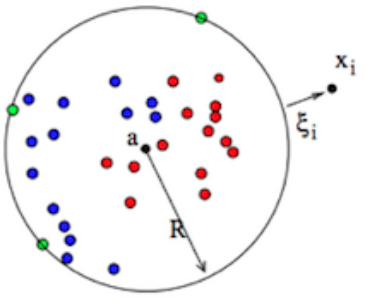

Figure 2.7: OCSVM SVDD [6].

Hybrid Detection

Misuse detection techniques, use supervised ML approaches previously mentioned, that identify malicious patterns of data. While this approach can have a high detection rate it may have problems identifying zero-day-attacks as the models are only trained with known attacks. Anomaly detection techniques on the other hand may be able to identify new attacks, but may have a high false positive rate if these new attacks show similar patterns to normal behaviour. Hybrid detection systems combine the strengths of using both anomaly and misuse detection techniques. Building a good interaction between these two techniques is the key challenge to ensure that the model performs well [23].

2.4.8 Natural Language Processing Deep Learning Models

In this section, we will discuss NLP models that are used for word embedding using Neural Networks. A Neural Network is inspired by the human brain, it's architecture mimics the current understanding of how a human brain learns. It based on neurons that are connected to each other and all participate in the learning progress. In particular, the focus of this section will be on the Word2Vec model. However, this approach incorporate both a Feedfoward Neural Network also referred to as Multi-Layer Percepton (MLP) and a Recurrent Neural Network (RNN)[8]. We will briefly provide some background to these Neural Networks before describing Word2Vec approaches.

Feedfoward Neural Network

A Feedfoward Neural Network is based of a simple Neural Network architecture where the neurons are connected in a foward manner only. Feedfoward networks are the simplest neural network used that are very efficient to learn and solve less complex patterns [30].

Each neuron is a simple linear regression that can be influenced by an activation function called a perceptron. This perceptron, similar to the process in a human neuron can fire if a certain stimulus occurs, in this case the stimulus is a threshold value defined for the outcome of the linear regression in the neuron which will cause the perceptron to activate and predict the output [7].

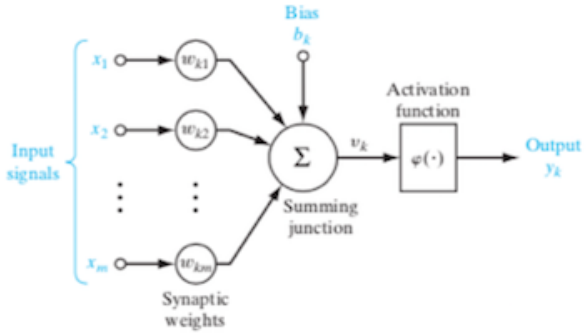

Figure 2.8: The Neuron Model [7]

.

If we take into consideration neuron k illustrated on figure 2.8, the input signals x_j is multiplied by the synaptic weight w_kj. The summing junction is a linear combiner of all input signals and their respective synapse weights. If a certain threshold is surpassed the activation function limits the amplitude of the output of the neuron to a finitive value, usually as a closed unit interval between [0,1] or [-1,1]. There is also a Bias b_k associated to the summing junction that is applied and increases or decreases the influence or net input of the activation function on the neurons final output [30].

A feedfoward Neural Network is a type of architecture where the neurons are layered in the simplest form. That is there is an input layer for source nodes that represent the input variables and then an output layer of neurons where the process explained above and computation occurs. In this structure the output neurons do not feed-backwards to the input signals and therefore are known as a feedfoward [7].

Recurrent Neural Network (RNNs)

A Recurrent Neural Network architecture aims to delve deeper into the human brain and how it learns by introducing feedback mechanism that allow communication within the layer of neurons and their respective inputs. These feedback mechanism improve the

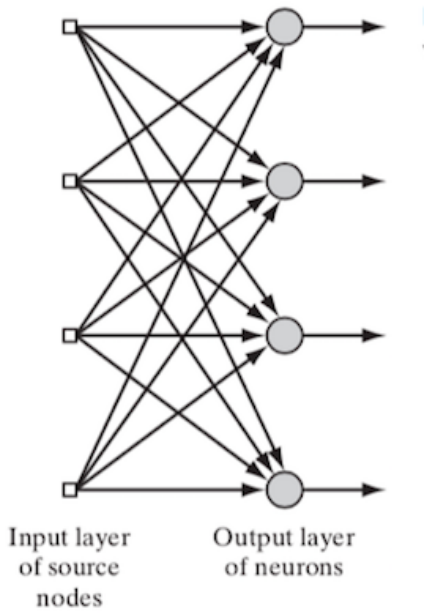

Figure 2.9: Feedfoward Structure wth a single layer of Neurons [7]

.

performance of the model, as we can see in figure 2.10 each input is branched into a time delay with $z^{-}1$ creating a new input for the other neurons.

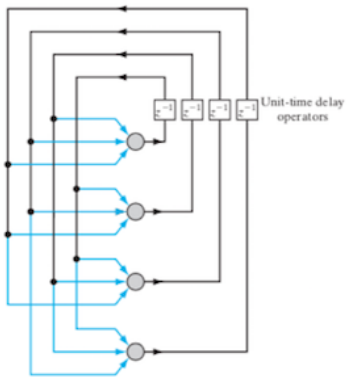

Figure 2.10: RNN Simple Architecture [7]

.

It is important to note most Neural Networks include also hidden layers, these are one or more layers of neurons that are not seen as from the input or output but contribute to the improvement of the computations and performance of the Neural Network.

Word and Paragraph Embedding

Word2Vec is a word embedding technique commonly used in NLP developed and presented in 2013 by Google researchers Mikolov *et al* [8]. This approach represents sim-

ilarity between words mathematically. The algorithm of Word2Vec does so, by being trained with a text corpus that obtains a numeric representation of the words in relation to each other. The numeric representation of each word is calculated based on a spatial coordinate of a word in relationship to the other words trained in the corpus. This approach therefore maintains the semantic relationship and context in its numeric representation of the word. There are two main approaches to the Word2Vec algorithm, Continuous Bag of Words (CBOW) and the Skip-Gram Model. We will also introduce an extension of these models called Doc2Vec.

Continuous Bag of Words (CBOW), Skip-gram Model and Doc2Vec

Continuous Bag of Words (CBOW), takes on a feedfoward neural network language model that attempts to predict a target word based on its context, that is on the words that are surrounding it. On the other hand the Skip-gram Model uses a RNN approach to instead of predicting a word based on it's surrounding ones, it does so by maximizing the classification of a word based on another in the sentence [8].

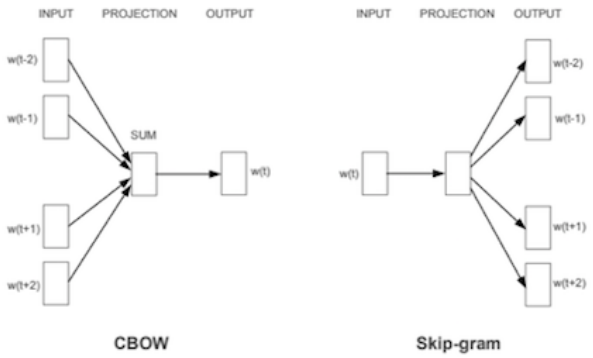

Figure 2.11: CBOW and Skip-Gram Word2Vec Models [8].

.

Doc2Vec, is based on the Word2Vec, but also creates a unique numeric representation for each paragraph. The output arrays for each paragraph can then be used as features for ML models.[31][32]. This process is based on the previous approaches mentioned however a unique paragraph ID is also part of the computation process for predicting the next word, where the paragraph ID can be interpreted as another word in the prediction process [33]. Figure 2.11 futher illustrates this method.

2.4.9 Dimensionality Reduction Techniques

Principal Component Analysis (PCA)

PCA is used to reduce the dimensionally of an input feature, so that it retains the most information possible. This will be of particular interest in this project when dealing with

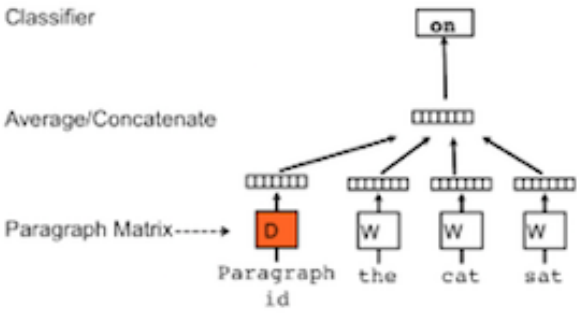

Figure 2.12: Doc2Vec Model [8].

.

sparse binary feature vectors. The PCA algorithm works by taking the multidimensional dataset, transposing it to a new matrix where each data point is subtracted by the mean of all rows. A covariance matrix is then calculated, from this covariance matrix the eigenvalues are calculated and its values ranked in descending order. The eigenvalue of a vector, divided by the sum of all eigenvalues gives a percentage of importance for each vector. PCA allows therefore to discard those vectors with lower importance and reduce the dimensionality of the feature space [34][23].

Auto-encondend Embedding

An autoenconder consists of a neural network of two parts. The first is an enconder which maps the input given, x, into a new feature vector with a lower dimension. The second part is a decoder which reverses the operation and maps the new feature vector back into it's original input feature space. Autoenconders learn by having a loss function that penalizes the enconded feature vector from being dissimlar to x [35]. A loss function commonly used is the Mean Squared Error (MSE) a value that is closer to 0 as the error is minimized. It is important to note that the autoenconder does not preserve the geometric distances between the data points, therefore we used manifold learning in addition to the autoenconder to preserve the distance characteristics in the data that will be key for geometric distance based models such as the OCSVM.

UMAP

UMAP (Uniform Manifold Approximation and Projection) is a manifold learning method that performs well on larger datasets due to it's scalability in comparison to PCA. UMAP has been shown to preserve both the local geometric structure of data points as well as the global geometric structure of the dataset [36]. We therefore transformed the feature space into a smaller dimensional feature space where the data points distance in the original space have been preserved. The parameters of UMAP can effect the trade-off between local and global geometric structure. This can be adjusted by the number of nearest

neighbours parameter; a lower value preserves more the local structure rather than the global.

2.5 Related Work

This paper aims to propose a tool that can complement previous advances in source code vulnerability detection. Static analysers are often used to ensure security at the application level, Huang et al. [37] show that security issues arise from data flows that have data integrity violations. To prevent these violations, it is necessary that mechanisms are used to ensure data integrity. The authors suggest a tool that can detect the information flow semantics and correct code that is causing a breach in data integrity without any input from the programmer. Static checking is used to find bugs in code, this method usually implies high false positive rates as it is not concerned with proving or checking the bugs found. Kronjee et al. [38] focused on the flow of data for vulnerability detection. To extract features, the authors used phply, a parser that converts code into Abstract Syntax Trees (ASTs). The ASTs were then used to obtain Control Flow Graph (CFG) that represent the flow of information of the program including all paths that can be traversed during it´s execution. Finally, the authors analysed the CFGs using techniques to identify changing definitions on the program, constants used and tainted (vulnerable) data flows. The features were obtained via the analysis and then used on machine learning algorithms (decision tree, random forest, logistic regression, naive Bayes, and tree augmented naive Bayes) to detect vulnerabilities. The research concluded that using ASTs and CFGs can be an effective way of detecting vulnerabilities.

The WAP tool [1] was developed to automatically detect input validation vulnerabilities, correct them and inform the program of the correction. The programmer in the loop approach, unlike previous work mentioned, contributes to the safety of web applications as programmers may learn and avoid future mistakes. To achieve this, static and taint analysis were used along with supervised machine learning to detect and correct vulnerabilities. As used by Kronjee et al. [38], taint analysis needs to manually done via human expertise to determine source code functions that are vulnerable. The WAP tool also looks at ASTs and CFGs to achieve this. Then, ML classifiers are used to reduce the false positive rates of the static analysis and the appropriate fixes are made to the code keeping the programmer on the loop. The feature selection process for the classifiers included taint analysis done by studying the vulnerabilities and attempting attacks to con-firm them. Our current approach will use the features of this tool, with some additions on research derived from WAP. Another tool named MERLIN, Figueiredo et al. [10], allows for vulnerabilities to be found in any programming language by combining CFG with a language neutral intermediate code. Slices of the intermediate code are then classified as vulnerable or non-vulnerable via machine learning algorithms. The approach we propose

on this paper, aims to use the features previously defined with a focus on unsupervised and anomaly detection machine learning algorithms.

DEKANT [12] is a tool derived from WAP that uses NLP for a new approach. As NLP derives meaning based on the surrounding text to the one being processed, this tool draws on this concept by using Hidden Markov Models (HMM) to determine whether a code slice is vulnerable or not. The source code is translated into an intermediate slice language (ISL) that is used by the model to predict the hidden state as vulnerable or not based on previous ones.

Other methods have been explored, for example Vuldeepecker [13] aims to eliminate the need to manually define features and therefore remove the necessity of human expertize. This tool uses an intermediate language (IL) that translates code into vectors while maintaining the semantic meaning of the code excerpt. It is important that semantic meaning of the code is kept so that it may detect vulnerabilities and also maintain a high granularity so that the location of the vulnerability may be identified. The IL is used as input for the ML model, which in this case is a deep learning model, in particular a BLSTM neural network. Vuldepeeker achieved very low false positive rates **1.9%**, however, when with a clear trade off for false negative rates, as testing with new datasets show rates as high as **90%**. Other research work by Fidalgo et al [9] also explores the use of ML deep learning for vulnerability detection in PHP code. In this case, deep learning model Long Short-Term Memory (LSTM) is used to detect SQLI vulnerabilities, also using an IL that assists the translation of source code into vector representations.

Chapter 3

HYVUDE - Hybrid Vulnerabilty Detection

In this chapter, we will discuss how we plan to tackle the problem of automatically finding vulnerabilities in PHP source code through a hybrid machine learning (ML) architecture and Natural Language Processing (NLP) techniques. We will begin by defining the problem of classifying vulnerable source and describe the issues that we intend to overcome with our proposed solution. In addition, we will describe the necessary steps required to represent suitable features with numeric values that can be used as input for our ML models. This will include, feature selection, pre-processing, tokenization and paragraph embedding. Alongside with the model description, we will detail how we construct the datasets from the source dataset, which will feed our ML models. Afterwards, we will introduce our hybrid architecture model to detect input validation vulnerabilities in web applications as well as the process used to decide the ML algorithms that will integrate the model.

The main problem that this thesis tackles is of using a hybrid machine learning to automatically detect vulnerabilities in source code. However, this main question is branched into research sub-questions that have guided this work and have the aim of providing a new contribution to previous work done in the field. In this section we will define these research sub-questions. Throughout this chapter we will provide an overview of the sample data available and our approach, the chapter will resolve with our new datasets built based of the sample data as well as the architecture of *HYVUDE* model.

The problem of automatically detecting vulnerabilities in source code can be further developed into the following sub-questions:

- What are the best features to represent source code snippets and can we derive datasets from these features with different levels of granularity?

- What is a suitable pre-processing approach that can be applied to general real-world source code snippets?

- Can NLP models provide unsupervised feature selection accurately?
- What is a suitable ML anomaly detection approach?
- Can a hybrid ML architecture provide an innovative solution?
- How can this research open doors for future work?

As we keep these guiding sub-questions in mind. We will first explain and provide examples of the problem of detecting vulnerabilities in source code. We will then, describe our solution process by going through the different stages until our final model proposition is presented.

3.1 Sample Data

For our work we will use datasets from the Software Assurance Reference Database (SARD) [2]. In total we will use three different datasets from this database that contain PHP source code snippets of both vulnerable and non-vulnerable code. Our first dataset has 1,363 instances of SQLI, where 861 are vulnerable and 502 are non-vulnerable. However, the following two datasets are split by type of vulnerability. One of the datasets contains 1,765 instances, where 965 have XSS vulnerabilities and 800 do not. We also have a small dataset of 101 instances where 16 have SQLI vulnerabilities and 85 do not. Overall we have 3,299 slices, where 1,464 are SQLI and 1,765 are XSS. This main dataset we will be refered to as *source dataset*, in which different datasets will derive from it.

The main task of our model is to correctly identify whether a source code snippet PHP application is vulnerable, and if so the type of vulnerability. The current sample data that we have available contains code snippets with various lines, each with a distinct set of instructions that are not all relevant for vulnerability detection. The first process is then to clearly define what is the best representation of this code, through an intermediate language. Let's consider the following code snippets as examples in Figures 3.1 and 3.2, both figures are very similar to each other, however is vulnerable to SQLI and the latter is not. The original source code snippet (Figure 3.1) is from the SARD database that will be used throughout this work. Similar to the example demonstrated on Chapter 2, this code snippet instructions pass through an entry point and reach as sensitive sink that can be exploited by a malicious user. In more detail, the entry point *$_POST['UserData']* can be exploited by entering a malicious input of meta-charters that can make the sensitive sink SQL query read or write in an unexpected manner. In addition, the function 'unserialize' is classified as a security risk as this function can result in code being executed as the function permits object instantiation and autoloading. A malicious user can abuse this function to inject malicious code. On the other hand, Figure 3.2 shows the same code snippet without the vulnerability. Here, the security risk function *unserialize* is removed

on line 2 and a sanitization function *mysql_escape_string* is added on line one to protect metacharacters from being maliciously used on the sensitive sink SQL query.

```
1   $string = $_POST['UserData'] ;
2   $tainted = unserialize($string);
3
4   //no_sanitizing
5
6   $query = "SELECT * FROM COURSE c
7                   WHERE c.id IN
8                       (SELECT idcourse FROM REGISTRATION
9                           WHERE idstudent=' $tainted ')";
10
11  //flaw
12  $conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
13  mysql_select_db('dbname') ;
14  echo "query : ". $query ."<br /><br />" ;
15
16  $res = mysql_query($query); //execution
17
18  while($data =mysql_fetch_array($res)){
19  print_r($data) ;
20  echo "<br />" ;
21  }
22  mysql_close($conn);
23
24  ?>
```

Figure 3.1: Example of a SQLI vulnerable code snippet taken for SARD database[2].

.

Taking into consideration the source code snippet discussed, the proposed solution must incorporate an intermediate language that can identify key functions that accurately represent each source code snippet. Previous work in the field [1][9] has provide us to have access to a complementary dataset of 65,552 instances, with 9005 marked as vulnerable and 56,547 non vulnerable. These vulnerabilities are not specified by type. Each instance consists of 19 features plus one for class, which all of them are binary. These features will be further discussed, as they are a framework to the intermediate language that we will adopt and develop to define the feature selection process.

3.2 Feature Selection

The features used were based on previous work in the field, mainly the work carried out by Medeiros et al [1] and Figueiredo et al [10] on the development of the WAP and MERLIN tools. Each feature accounted for a set of PHP code elements which are functions that are considered key elements that either make a code snippet vulnerable or non-vulnerable. Therefore, we defined these features as being *main features*. The main features range from functions that represented sensitive sinks or sanitization functions to the presence of SQL aggregation or even encryption functions. For example, as previously mentioned, SQLI attacks use metacharacters to manipulate SQL queries. If a code snippet contains a

```php
$string = mysql_real_escape_string($_POST['UserData']);

$query = "SELECT * FROM COURSE c
                WHERE c.id IN
                    (SELECT idcourse FROM REGISTRATION
                        WHERE idstudent=' $string ')";

$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

while($data =mysql_fetch_array($res)){
print_r($data) ;
echo "<br />" ;
}
mysql_close($conn);

?>
```

Figure 3.2: Adapted example of the same code snippet taken for SARD database without the vulnerability[2].

.

sanitization function that prevents inputs with metacharacters, this is a key factor in determining whether a snippet is vulnerable or not. Table 3.1 specifies in more detail each main feature defined, along with a brief description and a PHP code example. For instance, the main feature *Entry Point* (e.g *$_GET, $_POST*) will represent all code elements associated with the reception of user inputs through the web application surface. In total, we defined 19 main features, which aggregate 208 code elements. The 208 code elements for each main feature can be seen in more detail in Table 3.2.

Table 3.1: Main Features adapted from [9] and [10]

Main Feature	Description	PHP Example	Total
Entry Point	Where inputs enter the application data flow	$_GET	10
If Statement	Allows for conditional execution	if, else	2
Sensitive Sink	For a vulnerability to exist it must reach a sensitive sink	mysql_query	79
Sanitization Function	Clean inputs that may be malicious	mysql_escape_string	19
Extract Substring	Reads only a part of a string	substr	3
String Concatenation	Used to combine character strings into one	concat	1
Replace String	Replace strings, can be used to prevent malicious inputs	str_replace	15
Remove Whitespace	Removes whitespace or other characters from strings	trim	3
Type Checking	Checks the data type of a variable	gettype	32
Is Set	Check if the variable has been previously assigned with a value and is not NULL	isset	3
Pattern Control	Used to match expressions given a pattern	preg_match	5
Whitelist	Allows only for certain actions via the use of filters	filter_var	4
Error	Uses exception models to catch errors in the code	throw	2
Encryption	Is used to ensure privacy of data	crypt	5
Add Char	Used to prepare string inputs if user makes a mistake or is malicious	addslashes	1
SQL Query - AGG Function	Used for operations with aggregation functions on a SQL database	AVG	14
SQL Query - FROM Function	Used to determined whether the table on a SQL statement is a variable	FROM	1
SQL Query - Numeric Entry	Data matching numeric fields on a SQL statement, vulnerable if accepts meta-characters	REGEXP	2
SQL Query - Complex	SQL statements containing various tables and other queries	INNER JOIN	8

3.3 Pre-processing

In this stage, the aim is to prepare the data in order to translate the source code snippets into numeric vectors through two approaches, tokenization and Doc2Vec, both after a

Table 3.2: Full List of Single Features

Entry Point	db2_exec	imagecreatefromstring	eval	db2_escape_string	**Remove Whitespace**	preg_match	crypt
$_get	pg_query	imagecreatefromwbmp	setcookie	pg_escape_string	trim	preg_match_all	password_hash
$_post	pg_send_query	imagecreatefromxbm	setdrawcookie	pg_escape_bytea	ltrim	ereg	hash_equals
$_cookie	sqlite_query	imagecreatefromxpm	session_id	htmlentities	rtrim	eregi	password_verify
$_request	sqlite_exec	require	find	htmlspecialchars	**Type Checking**	strnatcmp	sodium_crypto_pwhash_str
http_get_vars	sqlite_array_query	require_once	findOne	strip_tags	gettype	strcmp	**SQL AGG**
http_post_vars	sqlite_single_query	include	findAndModify	urlencode	get_debug_type	strncmp	APPROX_COUNT_DISTINCT
http_cookie_vars	sqlite_unbuffered_query	include_once	insert	pg_escape_bytec	settype	strncasecmp	AVG
http_request_vars	ldap_add	readfile	remove	sqlite_escape_string	get_class	strcasecmp	CHECKSUM_AGG
$_files	ldap_delete	passthru	save	strip_tags	is_array	**Is Set Source**	COUNT
$_servers	ldap_list	system	execute	escapeshellcmd	is_bool	isset	COUNT_BIG
If Statement	ldap_read	shell_exec	header	escapeshellarg	is_callable	empty	GROUPING
else	ldap_search	exec	mail	**Replace String**	is_float	is_null	GROUPING_ID
if	xpath_eval	pcntl_exec	**Extract Substring**	str_replace	is_int	**Pattern Control**	MAX
Sensitive Sink	xptr_eval	popen	substr	str_ireplace	is_null	preg_match	MIN
mysql_query	xpath_eval_expression	echo	str_pad	substr_replace	is_numeric	preg_quote	STDEV
mysql_unbuffered_query	fopen	print	substr_replace	preg_replace	is_object	preg_match_all	STDEVP
mysql_db_query	file_get_contents	printf	**String Concatenation**	strtr	is_resource	preg_replace	STRING_AGG
mysqli_query	file	sprintf	concat	preg_filter	is_scalar	preg_split	SUM
mysqli_real_query	copy	vprintf	**Sanitization Function**	ereg_replace	is_string	**Whitelist**	VAR
mysqli_master_query	unlink	die	mysql_escape_string	eregi_replace	function_exists	filter_var	**SQL Numeric**
mysqli_multi_query	move_uploaded_file	error	mysql_real_escape_string	str_shuffle	method_exists	filter_input	REGEXP
mysqli_stmt_execute	imagecreatefromgd2	exit	mysqli_escape_string	chunk_split	ctype_alpha	filter_var_array	RLIKE
mysqli_execute	imagecreatefromgd2part	file_put_contents	mysqli_real_escape_string	str_split	ctype_alnum	filter_input_array	**SQL Complex**
mysqli->query	imagecreatefromgd	file_get_contents	mysqli_stmt_bind_param	preg_split	is_double	**Error**	INNER JOIN
mysqli->multi_query	imagecreatefromgif	vfprintf	mysqli->escape_string	explode	is_integer	throw	LEFT JOIN
mysqli->real_query	imagecreatefromjpeg	fprintf	mysqli->real_escape_string	split	intval	catch	RIGHT JOIN
mysqli_stmt->execute	imagecreatefrompng	scanf	mysqli_stmt->bind_param	spliti	boolval	**Encryption**	FULL JOIN

cleaning stage. These approaches will be used for the construction of datasets and the detection of vulnerabilities.

3.3.1 Cleaning

The code snippet is a piece of PHP code that is cleaned and pre-processed as text so the relevant words are maintained. Let's take for example Figure 3.3 as a code snippet taken for the SARD database.

```php
<?php
/*
Unsafe sample
input : get the $_GET['userData'] in an array
Uses an email_filter via filter_var function
construction : interpretation with simple quote
*/

$array = array();
$array[] = 'safe' ;
$array[] = $_GET['userData'] ;
$array[] = 'safe' ;
$tainted = $array[1] ;

$sanitized = filter_var($tainted, FILTER_SANITIZE_EMAIL);
if (filter_var($sanitized, FILTER_VALIDATE_EMAIL))
  $tainted = $sanitized ;
else
  $tainted = "" ;

$query = "SELECT lastname, firstname FROM drivers, vehicles WHERE drivers.id = vehicles.ownerid AND vehicles.tag=' $tainted '";

//flaw
$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password'); // Connection to the database (address, user, password)
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

while($data =mysql_fetch_array($res)){
print_r($data) ;
echo "<br />" ;
}
mysql_close($conn);

?>
```

Figure 3.3: Example of full SARD PHP code snippet [2]

The cleaning phase takes into consideration PHP programming basic syntax. This in-

cludes identifying comments and differentiating it from source code. For example, comments in PHP are between meta characters for both multi-line and single line comments. For single line comments it starts with // and multi-line comments are between */ and /*. Therefore, the cleaning phase includes removing any text that is a comment. This is done with a python script that removes multi-line text between */ and /* and single lines after the meta characters //.

3.3.2 Tokenization

The tokenization of the text is then done with the addition of usual NLP text cleaning techniques that transform all text into lower case, remove non-alphabetical characters (e.g. commas, brackets) and English stop words (e.g. the, is, and).

The final datasets used will consist of numeric vectors. Each position in the vector represents a feature. These features, depending on the type of vector to be constructed, can represent a single feature or a main feature, which is a categorical representation of a set of functions. These main features can be binary or as the frequency each main feature appears in the snippet. Hence, we have three type of vectors; two that are binary, for single and main features, and one with the frequency a main feature appears in a give code snippet. In addition, the source code snippets can also be represented through a fourth process, by numeric vectors derived from Doc2Vec, will be explained in more detail later.

As discussed in the previous section, a main feature represents a set of a certain type of functions related to the same purpose (e.g., sanitization function) and that were manually selected based on previous research done in the field. Firstly, features contain the code elements associated with them as text fragments. On the other hand, the source code provided by the SARD datasets is text (i.e, a set of line of code viewed as statements). Therefore, NLP techniques are used to obtain tokens of the code snippets and code elements meaning that both the source code and the code elements are tokenized. This allowed us to create a process that would compare the tokens in each main feature and code element to the tokens in each line of the code and thus obtain an intermediate representation of the code snippet, and from it return a vector based on the binary or frequency representation.

Nonetheless, there are challenges in the tokenization process as PHP language, while it is English based, it does not operate with the regular grammar and syntax rules of a spoken language. To cope with this, we resorted from the Multi Word Expression Tokenizer (MWE) NLTK library to create 128 multi-word tokens, combining separate words into a single token when these appear consecutively and are related to the same code element. It is also important to note that the expressions we wanted to combine into a multi-word token were added manually to the MWE to ensure the correct tokenization of the functions. For example, the code element *mysqli->mysql_query* when translated will be represented

by the token *mysqlimysqlquery* that we created. In other words, we had to ensure the tokenizer combined the 3 tokens *"mysqli" "mysql" "query"* into a single multi-word token. To achieve this processing, the first pre-processing step removes the characters common to PHP source code and replaces them with an empty space. It was necessary to indicate the removal of characters commonly used in PHP. Secondly, tokens are created where another pre-processing step removes any remaining non-alphabetic characters and stop words that may appear in the source code (e.g, $!->$). The resulting tokens we referred to as *first order tokens* which will be inputs to another tokenization step. Finally, the second tokenization step incorporates the MWE tokens added, so that it may recognize consecutive tokens that make up a function as a final token. Figure 3.4 below demonstrates this process.

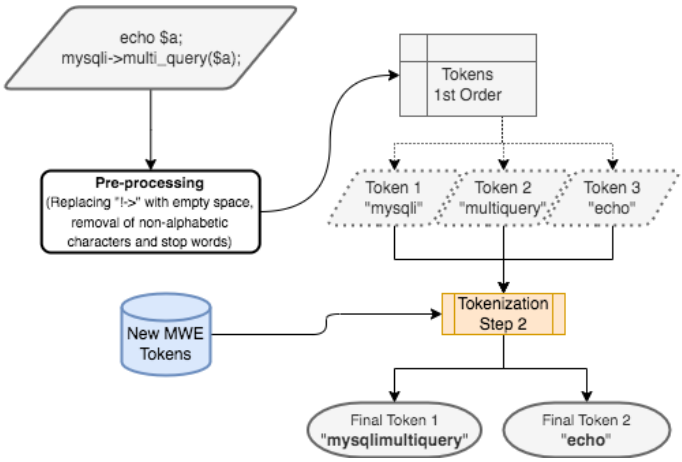

Figure 3.4: Extraction of the MWE tokens from the source code.

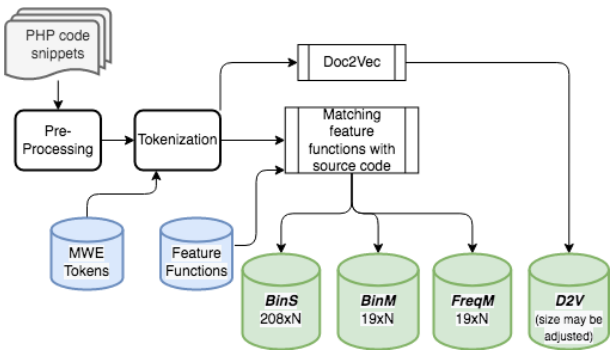

Figure 3.5: Process for the datasets construction.

Binary Main Features (BinM) **19xN**

Entry Point	Sensitive Sink	Sanitization Function	...	Vulnerability
1	0	1		0

Frequency Main Features (FreqM) **19xN**

Entry Point	Sensitive Sink	Sanitization Function	...	Vulnerability
2	3	1		1

Binary Single Features (BinS) **208xN**

Get	mysqlquery	mysqlescapestring	...	Vulnerability
1	1	1		0

Figure 3.6: BinS, BinM and FreqM Datasets

3.4 Paragraph Embedding - Doc2Vec

The second approach to create a numeric array from the source code text is via the use of Doc2Vec. As described in the previous chapter, this technique provides a numeric vector representation for each document, which in this case a document represents a code snippet. The Doc2Vec model was trained based on the tokenized snippet as previous steps and provided a numeric vector output for each code snippet on the dataset to be analysed by the vulnerability detection model. The 2 models CBOW and Distributed Memory model approach will be used and from these a final model will be selected based on the results of the ML model it will be used as input for. The first approach is the Continuous Bag of Words (CBOW) model from Word2Vec where for a given trained vocabulary, every word is mapped onto a unique vector, that is represented in a matrix column. This column has a index that indicates the position of the word in the vocabulary trained. The task of the model is to predict a word given another [8]. This is done by the sum of the unique vectors that are used as features in a neural network to predict the following word. In probablistic terms, the objetive of the model is given the training words $w_1, w_2, \ldots, w_T$ is to maximize the log probability;

$$\frac{1}{T}\sum_{T-k}^{t=k} \log p(w_t|w_{t-k}, \ldots, w_{t+k}) \tag{3.1}$$

After the model is trained, the words that have similar meaning (that is similar probability given same set of words) are mapped out in a similar position in the vector. It is important to note that the model improves it's word representation based on the quality of the vocabulary it is trained with. We have used our *source dataset* that is tokenized to train this model.

The second approach is the Distributed Memory Model that includes the Paragraph ID. Similar to CBOW model but instead the paragraph ID is also used to predict the following word. This paragraph token can be considered as another word that is used for the prediction computation[33]. This token can be also considered as a memory of what

the context of the paragraph is. In the case of this research work, if the document represent a code snippet and the context of this snippet will be remember with this paragraph token when predicting the words within it. It is important to note that each paragraph ID is taken into consideration when predicting the words of that paragraph only, but all the words in the vocabulary are taken into consideration regardless of the paragraph the prediction is being made in. Both approaches create an output that is a numeric vector that represents the content of each code snippet. These numeric vectors will be labelled for a classification task.

The importance of having this approach in addition to the current datasets built is that in this manner the features are not selected manually. This means, that we can use an unsupervised ML model with Doc2Vec to select the features (the numeric output of the code snippets). This removes the need for human know-how for the feature selection process and can be an interesting approach for both this thesis and future work.

3.5 Datasets

Four datasets were created, derived from the source dataset from SARD. These datasets include two datasets that are binary (one with the 19 main features and another with the 208 single features). One that represents the frequency of code elements in each main feature and one dataset that is based on the paragraph embedding technique Doc2Vec. As explained in the previous sections, every dataset firstly goes through a pre-processing and tokenization stage and then three of them enter in a matching token-code element process to finalize their construction while the fourth datset goes through the Doc2Vec process. For clarity, we will name the datasets as the following; *BinM* (binary main features dataset), *BinS* (binary single features dataset), *FreqM* (frequency main features dataset) and *D2V* (Doc2Vec dataset). Figure 3.5 illustrates the process to build the four datasets.

3.6 Hybrid Vulnerability Detection Model - HYVUDE

Previous work in vulnerability detection has not experimented with a hybrid approach in which unsupervised anomaly detection models used in inspection of network traffic will be used for detection of software vulnerabilities and combined with unsupervised learning. The objective is to experiment different ML models and datasets in an hybrid architecture and conduct experiments in order to achieve the best model for vulnerability detection.

The objective is to train an unsupervised model to i.e., first detect anomalies in non-vulnerable patterns. Those data points that are classified as anomalies, or outliers, will then be used as input for the supervised ML models. The supervised models will be

trained with the datasets that have been created and therefore will have ground truth to vulnerable and non-vulnerable data. As the examples used to train the models have SQLI and XSS vulnerabilities, the model should be able to detect these types of vulnerabilities.

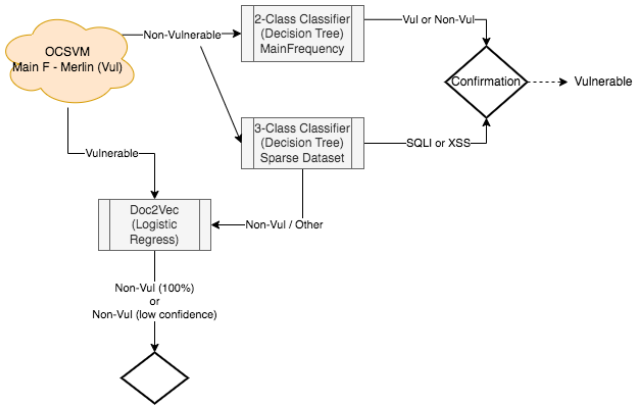

Figure 3.7: Hybrid Vulnerability Detection Model Architecture.

As illustrated in Figure 3.7, after the necessary pre-processing and tokenization process for the construction of our datasets, at a first instance we have experimented with the various datasets for best results of the unsupervised ML anomaly detection phase of our model. The aim is to first identify possible vulnerable data which will be referred to as suspicious data. We will then increase granularity by using a a combination of the remaining datasets on supervised models that will seek to confirm the vulnerability and identify the type. In this stage of the model architecture, we will experiment with both supervised individual and ensemble learning ML models for best results (e.g., Decision Tree and Random Forests). This first layer supervised ML model will have as input the suspicious data from the previous step and classify it as either vulnerable or non-vulnerable. The data classified as vulnerable will be considered as potential vulnerabilities, otherwise it will be considered as non-potential vulnerabilities. In both cases, it will be used on the second and final layer of supervised learning. This layer aims to maximize granularity by using the datasets that contain the most semantic information on the source code snippets. Therefore both *D2V* and *BinS* datasets will be used to train this supervised ML model and provide a final classification as either non-vulnerable or vulnerable including the vulnerability class identification. The aim of the architecture is to leverage different branches of current ML practices and provide a coherent solution that optimizes NLP for unsupervised feature selection, geometric-based (unsupervised or semi-unsupervised) anomaly detection models and supervised classification models.

3.6.1 One-Class Support Vector Machine (OCSVM)

Our first model, is based on a anomaly detection approach commonly used for traffic analysis in the cyber-security space. In this section, we will provide a more detailed

explanation of how the OCSVM model will be used in the context of detecting vulnerabilities in source code from our built datasets. In theory, a OCSVM is a unsupervised method in the context of outlier detection. That is, the trained data contains outliers that are defined by their geometric distance from the majority. In this approach, there is not a need for the data to be known as outliers apriori, and this method is most commonly used in the context of data cleaning. Nevertheless, in the context of our project, we are using the OCSVM for anomaly detection. This is a different problem that the OCSVM can solve, and while there is not a clear label for anomalies and normal data points, the training data must contain only data points that are considered normal. For clarity, we will describe this approach as semi-unsupervised, since if we look from a classification lens we know one of the two classes apriori for the training data only.

The *Scikit Learn* library will be used for the OCSVM model. The recommended kernel used is the non-linear *RBF* kernel [39], and the algorithm uses the approach introduced by Tax and Duin et al. [6] by fitting the training data of normal behaviour in the hypersphere. The original idea was to increase the granularity of the data throughout the architecture, meaning that the datasets would progress from simplest data *BinM* to the most complex *D2V*. Firstly, it was necessary to ensure that the OCSVM obtained satisfactory results in the testing phase with the datasets available. As previously mentioned, due to the fact that the OCSVM model operates in a semi-unsupervised fashion by being trained with labelled data apriori, we decided to test the model by training it with either vulnerable or non-vulnerable data from each dataset. Then, we tested it with either vulnerable or non-vulnerable data. For example, if the OCSVM was trained to recognize vulnerable data as normal behaviour, we then tested this OCSVM with both vulnerable and non-vulnerable data. It was then possible to classify a true positive if a vulnerable test data was classified as normal behaviour and a true negative if non-vulnerable data was classified as an anomaly. All combinations of test and train were performed for the datasets *BinM, FreqM, BinS* as well as the complementary dataset derived of the MERLIN tool [9] that we will refer to as *MERLIN* as well as the Doc2Vec arrays. Please note that these results are part of the model and dataset combination selection process, they are not experimental results that will be introduced later in this work with more appropriate ML model performance metrics.

To improve the quality of the features that best represent the vulnerable and non-vulnerable data, we experimented some autoenconding and manifold training techniques. As OCSVM are distance based, it was suitable to experiment with feature quality improvements used in unsupervised clustering algorithms. McConville *et al* [35] developed and coined the term *"Not too Deep (N2D) Clustering"*, a method that uses manifold learning on top of an autoenconded representation of the features where a shallow clustering algorithm is applied. For our project, we experimented with the same approach, except that instead of a shallow clustering algorithm such as K-means, we used a OCSVM.

The research carried out by McConville *et al* [35] concluded that combining local manifold learning to an autoencoded embedding, particularly UMAP that preserves the global structure, is most suitable for clusterability (Figure 3.8 demonstrates the order of steps that the data passes through before reaching the OCSVM model). Note that the data has been separated such that training is done only with vulnerable data and testing with non-vulnerable. We have also tested in the opposite order and will present some preliminary results in this chapter, to justify our architecture choices for the complete model.

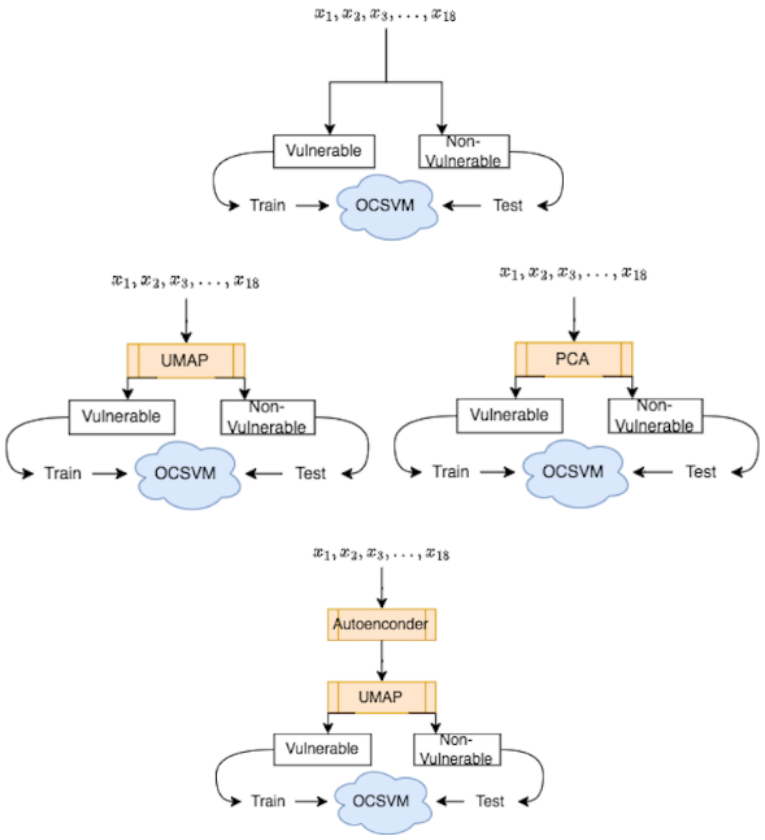

Figure 3.8: OCSVM Autoenconding and Manifold.

For all dimensionality reduction techniques we opted to reduce the data to 2 Dimensions as it is the standard practice for these techniques as well as it suits best the OCSVM for computation of the hypersphere. The autoenconder that was combined with UMAP was built with the *Keras* library. This autoenconder reduced the feature space to 5-D, UMAP reduced this feature space to 2-D. This autoenconder consisted of the standard *Keras* recommendation [40]:

- The input, which is the dataset used, which could have 19 or 208 integers.
- A fully connected neural network encoder with the activation function *"Relu"*.
- A fully connected neural network decoder with the activation function *"sigmoid"*.

- An optimizer that compiles the model using the *Adam* algorithm, that uses the gradient descent method as well the binary cross entropy loss function that penalizes wrong predictions of the model.

- A train and test split of the input data to train the model, and the training phase of 50 epochs of the autoencoder.

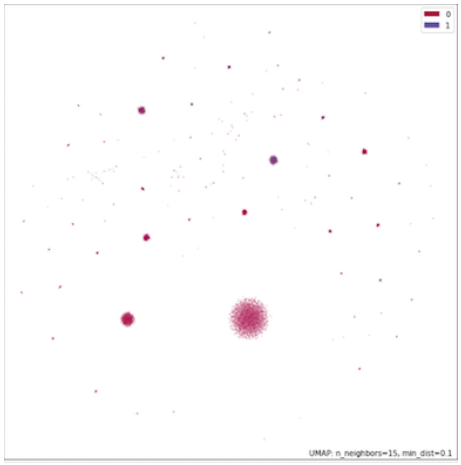

Figure 3.9: UMAP 2-D Feature Space.

Table 3.3: OCSVM without any dimensionality reduction

Normal	Anomalous	False Positives	False Negatives
MERLIN (Non-Vul)	MERLIN (Vul)	60.1%	10%
MERLIN (Vul)	**MERLIN (Non-Vul)**	**10%**	**23%**
MainF Bin (Non-Vul)	MainF Bin (Vul)	98%	21%
MainF Bin (Vul)	MainF Freq (Non-Vul)	94%	20.1%
MainF Freq (Non-Vul)	MainF Bin (Vul)	92%	12.9%
MainF Freq (Vul)	MainF Freq (Non-Vul)	90%	8.7%
Sparse Bin (Non-Vul)	Sparse Bin (Vul)	76.5%	85.6%
Sparse Bin (Vul)	Sparse Bin (Non-Vul)	77%	17.6%
Doc2Vec (Vul)	Doc2Vec (Non-Vul)	90%	8.7%
Doc2Vec (Non-Vul)	Doc2Vec (Vul)	96%	8%
Merlin (Vul)	MainF Bin (Non-Vul)	63%	10.2%
MainF Bin (Non-Vul)	**Merlin (Vul)**	**12%**	**17.8%**
Merlin (Non-Vul)	MainF Bin (Vul)	46.9%	10.5%
MainF Bin (Vul)	Merlin (Non-Vul)	43.3%	13.5%

The preliminary results presented in this chapter in the Tables 3.3 and 3.4 regarding dimension reduction techniques are to highlight our decision to opt for the UMAP approach without the autoenconding for our final model. This is because, while the autoenconder improved results for the *Sparse Bin* dataset that has 208 integers as input, it does not improve subtentially the other results where the dimensionality is low (19). In addition, the autoenconder uses a neural network with 50 epochs in training that increase computational costs substantially.

Table 3.4: OCSVM Experimental results using PCA and UMAP with and without Autoenconder for MERLIN data

Technique	Normal	Anomalous	False Positives	False Negatives
PCA	MERLIN (Vul)	MERLIN (Non-Vul)	12%	93%
PCA	MainF Bin (Non-Vul)	MERLIN (Vul)	15%	84%
UMAP	MERLIN (Vul)	MERLIN (Non-Vul)	13%	9.7%
UMAP	MainF Bin (Non-Vul)	MERLIN (Vul)	8%	10.8%
Autoenconder + UMAP	MERLIN (Vul)	MERLIN (Non-Vul)	23%	19.4%
Autoenconder + UMAP	MainF Bin (Non-Vul)	MERLIN (Vul)	31%	21.2%
Autoenconder + UMAP	Sparse Bin (Non-Vul)	Sparse Bin (Vul)	54%	79.7%
Autoenconder + UMAP	Sparse Bin (Vul)	Sparse Bin (Non-Vul)	62%	11.2%

3.6.2 Decision Tree and Random Forest

The previous OCSVM will be used as first layer to detect anomalies that will be interpreted as vulnerable or non-vulnerable. The objective, is for the data points considered normal behaviour to progress into the the logistic regression layer and those identified as anomalies, thus non-vulnerable to a random forest multi-class classifier, which as previously explained in Chapter 2 is an ensemble of decision trees. This supervised approach will aim to identify the type of vulnerability, in this case either SQLI, XSS or non-vulnerable. The random forest combines decision trees and uses averaging for the final predictions. The following parameters will be taken into consideration (further experimentation and results will presented in Chapter 5) [39]:

- Number of decision trees will be set to the default 100.
- The best split criteria can be based on the *"Gini Index"* or *Entropy*.
- Maximum tree depth, this parameter if it is not defined apriori will result on the decision trees creating splits until there is a pure node.

For example, if all instances of the feature *Sensitive Sink* are equal to one for vulnerable labeled data and zero for non-vulnerable, then the decision tree would consist of a root and two pure leafs that could determine the class of the code snippet (see Figure 3.10). Since, this is not the case, and the presence of sensitive sink is also possible in non-vulnerable data, a decision tree uses the best split criteria and grows until a pure node is found at it's maximum depth.

The *HYVUDE* architecture uses the *FreqM* (frequency main features dataset) and *BinS* (binary single features dataset) for 2 random forest models. If both models predict the same vulnerability, then the final classification is indeed that type of vulnerability. However, if both models do not agree on the vulnerability then the prediction is vulnerable with high confidence if both point to a different vulnerability, or low confidence if at one of the two predicts the data as non-vulnerable.

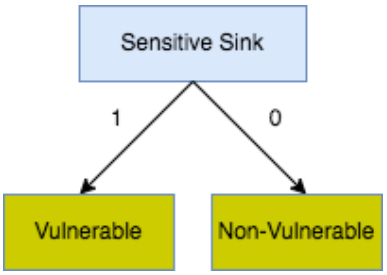

Figure 3.10: Pure Leaf - Decision Tree.

3.6.3 Logistic Regression using the Doc2Vec Dataset

The datasets generated from the CBWO and Distributed models will output an a numeric array representing the code snippet vocabulary. These arrays will be used for a multi-classification task using logistic regression. In the multi-classification problem, the parameters are set as follows [39]:

- Multi-class parameter is set to *ovr* instead of the one versus all method used for binary classification.
- *C* parameter defines the generalization strengh where lower values increase the generalization.
- *Solver* parameter defines the algorithm used for optimization, in the case of multi-classification it can either be *'newton-cg', 'sag', 'saga'* and *'lbfgs'*.

This model has the highest level of granularity as it uses features determined by the unsupervised Doc2Vec model that take into consideration the context and overall vocabulary of all code snippets.

Chapter 4

Implementation

In this chapter we will provide a detailed explanation of the implementation steps of the model. We will begin by specifying the pre-processing and intermediate language creation stage that creates our datasets. We will then describe how the ML models were implemented in the hybrid architecture and how model provides a final classification. The resulting tool was implemented in Python, resorting diverse modules from it for representation of data structures, NLP, and machine learning.

4.1 Source Code Pre-processing

In *Python* we used the libraries *Pandas*, *Numpy* and *NLTK* for the pre-processing of the PHP source code. Source code snippets were extracted from *SARD* and stored in a file (one file for each source code snippet). A *python* script was used to process the PHP source code and tokenize each word for our dataset construction. As explained in Chapter 3, a code snippet may contain comments that are between characters $<$ and $>$. The text between these characters will be removed on the pre-processing stage. In addition, characters such as *$* will also be removed. The remaining text is tokenized and the datasets are created via static analysis that determine the presence of key functions on the dataset. Note that the functions that represent the features and are being matched with the tokenized *PHP* source code are stored in separate files that are called for the matching dataset creation phase. These functions are also tokenized and then used to match with the tokens generated from the *SARD* code files.

Algorithms 1 and 2 describe this process, the main features where stored in *csv* files and the source code snippets from SARD on a separate folder containing the respective *php* files. In addition, the Multi-Word Expression (MWE) Tokens that combine a different tokens into a single one were declared and used as input when tokenizing both the main feature and source code snippet files.

To build the datasets, both tokenized Main Feature Functions and PHP Source Code are matched and appended to a dataframe where each column has described in Chapter 3

Input: MWE Tokens;
PHP code snippets;
Main Feature Functions CSV Files;
Output: Pre-processed and Tokenized PHP Code Snippets
foreach *Main Feature Function as CSV File* **do**
  Tokenize ← nltk.word_tokenize
  Convert Tokens into lowercase
  Remove punctuation and non-alphabetic characters
  Filter out stop words
  Save final tokens in variable with Main Feature Name
end
foreach *PHP Code Snippet as Code* **do**
  Tokenize ← nltk.word_tokenize
  Convert Tokens into lowercase
  Remove punctuation and non-alphabetic characters
  Filter out stop words
end
return Res [] List of Lists with pre-processed and tokenized PHP Code

Algorithm 1: Pre-Processing and Tokenization

accounts for a Main Feature. Therefore, we can describe the implementation process as below.

After the pre-processing and dataset creation stage, the datasets labelled firstly as vulnerable or non-vulnerable for a binary classification task, but then also for a multi-classification task with the type of vulnerability.

4.2 Full Architecture Construction and Model Evaluation

4.2.1 OCSVM Implementation

As discussed on Chapter 3, the architecture of our model begins with a OCSVM that will be trained with vulnerable data from the *Merlin* binary dataset. The OCSVM algorithm will predict anomalous data points from the *BinM* dataset. However, it is important to note that the data that will be used in the prediction phase represents new data (that is a code snippet) that was not seen by the model and the user would like to classify. Also, unlike the preliminary experiment in Chapter 3, the predicted data will contain both vulnerable and non-vulnerable data to evaluate the OCSVM ability to correctly identify both normal behaviour (vulnerable data) and anomalous behaviour (non-vulnerable). In addition, the data points will have a dimension reduction via the use of *UMAP* that preserves the both the local and global geometry of data points. Algorithm 3, illustrates the steps for the

Input: Pre-Processed Main Feature Functions;
Pre-Processed PHP code;
Output: *BinM, BinS, FreqM* Datasets
if *Tokenized PHP code snippet contains* Tokenized Main Feature function x **then**
  feature = 1 // e.g. contains an entry point
end
else
  feature = 0
end
return Binary Main Feature Dataset
if *Tokenized PHP code snippet contains* Tokenized Main Feature function x **then**
  feature = enumerate // e.g. number of times it contains an entry point
end
else
  feature = enumerate
end
return Frequency Main Feature Dataset
if *Tokenized PHP code snippet contains* Tokenized Sparse Feature function x **then**
  feature = 1 // e.g. number of times it contains any type of entry point function such as "get"
end
else
  feature = 0
end
return Sparse Binary Dataset

Algorithm 2: Dataset Construction

OCSVM part of the architecture.

4.2.2 Random Forest Implementation

As the OCSVM has been trained to recognized Vulnerable data - we will have higher confidence in the predicted data that is recognized as Normal (vulnerable) behaviour. The data points that are predicted as Non-Vulnerable will move on to the random forest stage of the hybrid architecture. This section consists of two different random forest models, each using a different dataset for training. As we increase granularity of the information that is described in each dataset, we will use the *MainF* (frequency of each Main Feature) dataset for a binary classification task solved by a random forest to predict the class as Vulnerable or Non-Vulnerable. This model will be used in combination of another random forest that will use the more detailed dataset *Sparse Bin* that has the presence of all 208 features described as binary values for a multi-classification task that can predict 3 distinct outcomes; Non-Vulnerable, SQLI or XSS. If the first random forest predicts non-

vulnerable and the second a type of vulnerability, then this data will be further classified on the following model. However, if the first random forest classifies the data as vulnerable and the second random forest predicts a vulnerability type, then the architecture will interpret this as the final prediction. Algorithms 4 describes this classification phase, with the training of random forests. As input we will use in the binary classification task the *MainF Frequency* dataset and in the multi-classification for the vulnerability type the *Sparse Bin*. The random forest classifier will take a subset of the instances of the data used, also referred to as bagging as introduced in Chapter 2. For each bootstrap sample a decision tree will be learned, the algorithm will compute the decision gain based on a smaller subsample of the features which will be less computationally expensive and faster then doing so for all features. In addition, this ensemble learning technique uses mega learning as it defines the final prediction based on the combined outcomes of all decision trees.

Input: *Training and Test Dataset* defined as per $Train$ and $Test$ Split (testsize $= 0.2$), where the training data $D = (x_1, y_1), \ldots, (x_n, y_n)$ with the features F and the number of decision trees in the random forest T.
Output: *Predicted Class* based on random decision trees majority voting.
Random Forest(D, F)
$R \leftarrow \emptyset$
foreach $i \in 1, \ldots, T$ **do**
  $D^i \leftarrow$ *the* ith bootstrap sample from the training Data D
  $r_i \leftarrow$ *RandomTreeLearn*(S^i, F)
  $R \leftarrow R \cup \{r_i\}$
end
Predict the outcome of every random decision tree on the test data
Define the class that was predicted the most times as the final predicted class

Algorithm 3: Random Forest Process for both Binary Classification using *MainF Frequency* and Multi-Classification using *Sparse* Datasets

4.2.3 Doc2Vec and Logistic Regression Implementation

The implementation of the Doc2Vec model that will be used for a logistic regression classifier will firstly use the final tokens that is returned on Algorithm 1. These tokens will be used to train as vocabulary and test the Doc2Vec model. The numeric representation of each PHP code snippet will then be used as input for the logistic regression for the multi-classification of the vulnerability type. It is also important to note, that there will be a threeshold value that will determine the level of confidence of the logistic regression prediction. The model will output a probability that a snippet belongs to a certain class, and therefore the threshold value of this probability will define the final classification. Algorithm 4 describes the Doc2Vec model training where each PHP code snippet is to-

kenized as per Algoritm 1 and then labeled with the type of vulnerability. This is used to train the model with the current standard parameters. These parameters such as for example, *vectorsize=300* will be experimented for best results in the following chapter. In this case, we are training the model to output a numeric representation of each code snippet of 300 vectors. Once the model learns the vocabulary it is given, it provides a numeric representation of the spatial positioning of each token in regards to the context of the overall vocabulary. This numeric representation makes up the *D2V* dataset that will be used to train and test the Logistic Regression Multi-Classifier for the vulnerability type defined on Algorithm 5. In this model, a hyperparameter tuning process will test out a combination of different parameters for best results. However, we have fixed the parameter number of jobs equal to 1 as of standard practice for CPU computation efficiency and regularization parameter is set to 10^5 which instructs the model to provide higher importance to the training data rather than to complexity to fit the model in spite of the training data given.

Input : A list of Labeled Source Code snippets (tagged documents)
Output: A trained Doc2Vec model that provides y_{train}, X_{train}, and y_{test}, X_{test} for the Logistic Regression Model

D2V Model ← Doc2Vec(dm=1 or 0, vector size=75, 150, or 300)

Module `Vector for learning` (*model, labeled source code snippets*)
  return targets, regressors

y_{train}, X_{train} ← `Vector for learning` (*D2V Model, train tagged documents*)
y_{test}, X_{test} ← `Vector for learning` (*D2V Model, test tagged documents*)

for *epoch in range(10, 30, or 60)* **do**
  `Train` ← (train tagged documents)
end

Algorithm 4: Doc2Vec Model Training

Input: Training data X_{train}, training labels y_{train}, testing data X_{test}
Output: Predicted labels y_{pred}
Instantiate logistic regression model $logreg$ with hyperparameters
  number of jobs $n_{\text{jobs}} = 1$, regularization parameter $C = 10^5$
  $logreg \leftarrow$ LogisticRegression(n_{jobs}, C)
  Fit $logreg$ on training data and labels
  $logreg.fit(X_{\text{train}}, y_{\text{train}})$
Predict labels for testing data using the trained model
  $y_{\text{pred}} \leftarrow logreg.predict(X_{\text{test}})$
return y_{pred} `// Final Prediction`

Algorithm 5: Logistic Regression Classifier

The logistic model will provide the final classification and act as a decision maker if the previous models give conflicting outcomes. As this approach uses an unsupervised feature extraction method with the numeric arrays via the vocabulary training with the Doc2Vec model, it is higher in granularity in comparison to others and therefore given more weight the final prediction of the overall architecture. To summarize, if the OCSVM model detects the data points as Vulnerable, then the logistic regression model is used to specify the vulnerability type. However, if the OCSVM model classifies the data as Non-Vulnerable, then as specified on the previous chapter, the final classification will be defined as the table 4.1 below.

Table 4.1: Final Predictions given OCSVM Non-Vul

MainF Random Forest	**Sparse Random Forest**	**D2V Logistic Regression**	**Final Prediction**
Vul	XSS	-	Vul XSS
Vul	SQLI	-	Vul SQLI
Vul	Non-Vul	Non-Vul	Non-Vul
Vul	Non-Vul	SQLI	Vul
Vul	Non-Vul	XSS	Vul
Non-Vul	XSS	XSS	XSS
Non-Vul	SQLI	SQLI	SQLI
Non-Vul	SQLI	XSS	Vul
Non-Vul	XSS	SQLI	Vul
Non-Vul	SQLI	Non-Vul	Non-Vul
Non-Vul	XSS	Non-Vul	Non-Vul
Non-Vul	Non-Vul	Final Prediction	Final Prediction

Chapter 5

Experiments

In this chapter, we will present and describe the experiments conducted on the proposed hybrid vulnerability detection model to determine a final configuration for both each individual ML model and the complete architecture of the HYVUDE tool. Firstly, we will tackle each individual model and experiment with the hyper-parameters of each model as well as the datasets used. We will then present an overall experimentation process of the full architecture with the current datasets introduced. This chapter aims to provide the appropriate results to answer the proposed research questions of that we defined on Chapter 3:

- What are the best features to represent source code snippets and can we derive datasets from these features with different levels of granularity?
- What is a suitable pre-processing approach that can be applied to general real-world source code snippets?
- Can NLP models provide unsupervised feature selection accurately?
- What is a suitable ML anomaly detection approach?
- Can a hybrid ML architecture provide an innovative solution?
- How can this research open doors for future work?

5.1 Evaluation Metrics

The metrics we used to evaluate HYVUDE, and its individual modules, are based on a confusion matrix which identifies the number of correctly identified vulnerabilities (True Positives) and non-vulnerabilities (True Negatives) as well as incorrectly identified vulnerabilities (False Positives) and non-vulnerabilities (False Negatives). These same metrics can also be used in the context of a multi-classification task.

1. Precision

   The ratio of correctly identified vulnerable code snippets among all vulnerabilities that were identified by the model, given by:

   $$Precision = TruePositives/(TruePositives + FalsePositives)$$

2. Recall

   The ratio of correctly identified vulnerable code snippets among all vulnerabilities known apriori, given by:

   $$Recall = TruePositives/(TruePositives + FalseNegatives)$$

3. F-Score

   Measurement of the model's accuracy, based on the harmonic mean of precision and recall. The values are between 0 and 1, where 1 is means the model has a perfect precision and recall. This value is given by:

   $$F - score = 2 * Precision * Recall/(Precision * Recall)$$

   In addition, for multi-classification of the type of vulnerability we will also take into consideration the macro-averaged F1 score, which is computed using an weighted mean of all the F1 scores per class - this method is most suitable since it is useful for unbalanced datasets where all classes are equally important.

4. Accuracy

   The ratio of the total number of correct predictions of the total predictions made by the model, given by:

   $$Accuracy = (TP + TN)/(TP + TN + FP + FN)$$

From our architecture, what we want to achieve is a fair trade-off between true positives and false negatives. In the context of vulnerabilities, a false negative is more concerning than a false positive, therefore, this is the main focus. Nevertheless, if it is at the expense of a high false positive rate, then the tool becomes unusable and unreliable. Therefore, we expect a high precision and recall value (which leads to high F-score) as well as a an acceptable high accuracy score.

5.2 Evaluation of the OCSVM Model

The experiments of the OCSVM are key to the architecture as they are the fundamental in understanding the validity of a hybrid architecture for vulnerability detection. The goal is

to understand that with the current datasets available this approach can bring added value and/or provide an interesting approach to further study. We will begin by training the OCSVM with only vulnerable data points with different hyper-parameters and testing to understand it's performance.

The hyper-parameters we will experiment are:

1. *Kernel* - As explained on Chapter 2, kernels will increase the input space for higher dimensional feature spaces. The kernels experimented with will be *'linear', 'poly', 'rbf' and 'sigmoid'*.

2. Parameter *nu* - Similar to *C* value introduced by Tax and Duin et al.[6], this value is between $[0, 1]$ and represents the trade-off between generalization and overfitting of the model. In other words, the higher the *nu* value a higher percentage of the training data points can be wrongly miss-classified as an anomaly in the training phase, a lower value will mean that all data points are within the decision boundary and therefore classified as normal behaviour that can cause the model to overfit.

3. All other hyper-parameters will be set to the default values.

Table 5.1 presents the results for the different parameters that were experimented. In this case, the OCSVM was trained only with vulnerable data points from the *MainF* binary dataset. The results pointed out that the model could identify non-vulnerable points to high precision. In particular, *nu* values of 0.1 provided the best results across all Kernels, with the *sigmoid* kernel performing the worst. These experiments suggest that the OCSVM can identify non-vulnerable data points however, false negatives are very high as vulnerable data points have very low precision, recall and F-1 Score values. We can then take into consideration in our architecture that if the OCSVM is trained with vulnerable data as normal behaviour - the model with the Merlin dataset will classify most data as anomalous and thus non-vulnerable. We will continue our experimentation phase of the OCSVM with the 1) training the OCSVM with non-vulnerable data points 2) training and testing the model with *MainF Bin* and 3) training the model with the a mixed dataset and testing it with new instances from both the *Merlin* and *MainF Bin* datasets. Table 5.2 shows the results of these 3 experiments.

Table 5.1: OCSVM tested with the MainF Binary Dataset

Kernel = rbf	**Parameter *nu = 0.1***				**Parameter *nu = 0.01***			
		Precision	Recall	F-1 Score		Precision	Recall	F-1 Score
	Non-Vulnerable (**Anomaly**)	0.89	0.75	0.81	Non-Vulnerable	0.90	0.65	0.75
	Vulnerable (Normal)	0.21	0.42	0.28	Vulnerable	0.20	0.56	0.30
	Accuracy			0.70	Accuracy			0.64
	Parameter *nu = 0.5*				**Parameter *nu = 0.9***			
		Precision	Recall	F-1 Score		Precision	Recall	F-1 Score
	Non-Vulnerable	0.86	0.77	0.80	Non-Vulnerable	0.86	1.00	0.93
	Vulnerable	0.19	0.32	0.24	Vulnerable	0.00	0.00	0.00
	Accuracy			0.71	Accuracy			0.81
Kernel = linear	**Parameter *nu = 0.1***				**Parameter *nu = 0.01***			
		Precision	Recall	F-1 Score		Precision	Recall	F-1 Score
	Non-Vulnerable	0.88	0.80	0.84	Non-Vulnerable	0.88	0.84	0.86
	Vulnerable	0.21	0.32	0.25	Vulnerable	0.22	0.29	0.25
	Accuracy			0.74	Accuracy			0.76
	Parameter *nu = 0.5*				**Parameter *nu = 0.9***			
		Precision	Recall	F-1 Score		Precision	Recall	F-1 Score
	Non-Vulnerable	0.87	0.82	0.84	Non-Vulnerable	0.87	0.96	0.91
	Vulnerable	0.18	0.26	0.21	Vulnerable	0.23	0.07	0.11
	Accuracy			0.74	Accuracy			0.84
Kernel = poly	**Parameter *nu = 0.1***				**Parameter *nu = 0.01***			
		Precision	Recall	F-1 Score		Precision	Recall	F-1 Score
	Non-Vulnerable	0.88	0.80	0.84	Non-Vulnerable	0.87	0.85	0.86
	Vulnerable	0.21	0.32	0.25	Vulnerable	0.20	0.24	0.21
	Accuracy			0.74	Accuracy			0.76
	Parameter *nu = 0.5*				**Parameter *nu = 0.9***			
		Precision	Recall	F-1 Score		Precision	Recall	F-1 Score
	Non-Vulnerable	0.87	0.84	0.86	Non-Vulnerable	0.86	0.99	0.92
	Vulnerable	0.19	0.24	0.25	Vulnerable	0.18	0.01	0.03
	Accuracy			0.76	Accuracy			0.86
Kernel = sigmoid	**Parameter nu = 0.1**				**Parameter nu = 0.01**			
		Precision	Recall	F-1 Score		Precision	Recall	F-1 Score
	Non-Vulnerable	0.89	0.79	0.84	Non-Vulnerable	0.87	0.90	0.88
	Vulnerable	0.22	0.37	0.27	Vulnerable	0.21	0.17	0.19
	Accuracy			0.64	Accuracy			0.80
	Parameter nu = 0.5				**Parameter nu = 0.9**			
		Precision	Recall	F-1 Score		Precision	Recall	F-1 Score
	Non-Vulnerable	0.87	0.84	0.86	Non-Vulnerable	0.87	0.96	0.91
	Vulnerable	0.19	0.23	0.21	Vulnerable	0.23	0.07	0.11
	Accuracy			0.76	Accuracy			0.84

Table 5.2: OCSVM - Further Experimentats

Parameter nu = 0.1	***Merlin Binary Dataset***	Precision	Recall	F-1 Score	***MainF Binary***	Precision	Recall	F-1 Score
Kernel = poly	Non-Vulnerable (**Normal**)	0.92	0.50	0.65	Non-Vulnerable (Anomaly)	0.43	0.46	0.44
	Vulnerable (Anomaly)	0.19	0.74	0.30	Vulnerable (**Normal**)	0.57	0.54	0.56
	Accuracy			0.53	Accuracy			0.51
	Trained MainF Bin, Merlin Bin ***Tested Merlin Binary***				***Trained MainF Bin, Merlin Bin*** ***Tested MainF Bin***			
	Non-Vulnerable (Anomaly)	0.86	0.97	0.92	Non-Vulnerable (Anomaly)	0.43	1.00	0.60
	Vulnerable (**Normal**)	0.17	0.03	0.05	Vulnerable (**Normal**)	0.00	0.00	0.00
	Accuracy			0.85	Accuracy			0.43

In terms of hyperparameters, our experiments suggest that with the *nu* value of 0.1 the *poly* kernel presents the best results. The OCSVM has high precision values detecting non-vulnerable anomalous behaviour particularly when trained with Merlin dataset, however a mixed dataset provides also good results for anomaly detection. Nevertheless, we must understand that this model will always be limited to the dataset we have available. Like most problems from a data science perspective we rely on the quality of our dataset as it must be representative of the real-world. Nevetherless, these results provide a basis where future study can look at using a OCSVM for anomaly detection on better representative datasets. Due to our model's weakness in correctly classifying normal behaviour (that is vulnerable) we will rely on the other models in our architecture, specially due to the fact the model has a high false negative ratio which is problematic for vulnerability detection. However, due to the model's success in identifying anomalous vulnerable behaviour, this approach is valid and can be improved in future research as more and more data is added and a higher representation of vulnerable and non-vulnerable instances are used to trained the model.

5.3 Evaluation of the Random Forest Classifier

The next step of our HYVUDE architecture is the Random Forest Classifier. The original conception of the architecture has two random forest models. The first, is a binary classifier that predicts a data point to be Vulnerable or Non-Vulnerable. The second random forest model is used to classify the type of vulnerability as one of 3 classes; XSS, SQLI or Non-Vulnerable. The concept of our architecture is based on the increase in granularity of the datasets. Therefore, in this phase as described on Chapter 3, we intend to use the *MainF Frequency* dataset for the binary classification and the *Sparse* dataset for the multi-classifier. In this experimentation phase, we test the binary and classification models with both datasets and in addition experiment with hyper-parameter tuning to improve the models.

Table 5.3: Random Forest Binary Classification without Hyper-parameter Tuning

Dataset	Precision	Recall	Accuracy
MainF Frequency	0.68	0.69	0.70
Sparse	0.79	0.75	0.79

For random forest, we used the *scikit learn* library's *Randomized Search* for training the model and with 10-Cross Fold Validation. This approach splits the dataset into a training and test set but the training set is further split K times, where K is a fold, or subset of the training data. The model will fit the data K times (we will use 10) where the data will be trained with $K-1$ folds and validated on fold K. This is done for all K times and the performance metrics of the model will be averaged to determine the final metrics of the random forest. For hyper-parameter tuning this is repeated for all possible combinations combinations of hyper-parameters and defines the best combination of parameters based on the best performing metrics. When dealing with Random Forest's the hyper-parameters we will be dealing with are the following:

1. *Number of estimators* - This defines the numbers of trees in the Random Forest.

2. *Maximum Depth* - This determines how far the tree progresses, if this parameter value is set to *None*, the the tree will split until all leaves in the tree are pure or there is only one sample left on an internal node.

3. *Best Split Criteria* - as described in Chapter 2, this is the impurity measurement used to define a best split, we will experiment with *Gini* and *Entropy*.

4. All other hyper-parameters will be set to the default values.

We will begin by presenting the results of the binary classification Random Forest with and without the hyper-paremter tuning for the *MainF Frequency* and *Sparse* datasets.

Table 5.3 shows the results for the binary random forest classifier without any hyper-parameter tuning. In addition, the confusion matrices on Figure 5.1 as well the Receiving Operating Characterist (ROC) Curve on Figure 5.2 provide further visualization of these initial results. The ROC Curve demonstrates the relationship between True Positive and False Positive rates between the two datasets used. It can be visualized that the *Sparse* dataset performed better than *MainF Frequency*, as the latter shows unsatisfactory results with only a accuracy of 0.70 in comparission to 0.79 for the *Sparse* dataset.

As previously mentioned, using *scikit learn* library's *Randomized Search* with Cross Fold Validation set to 10 we obtained results showed on Figure 5.5.

Both datasets had a slight improvement in their metrics with hyper-parameter tuning, it is important to note that *Sparse* dataset performs better across all metrics than *MainF Frequency*, we will therefore consider adjusting our architecture accordingly taking into

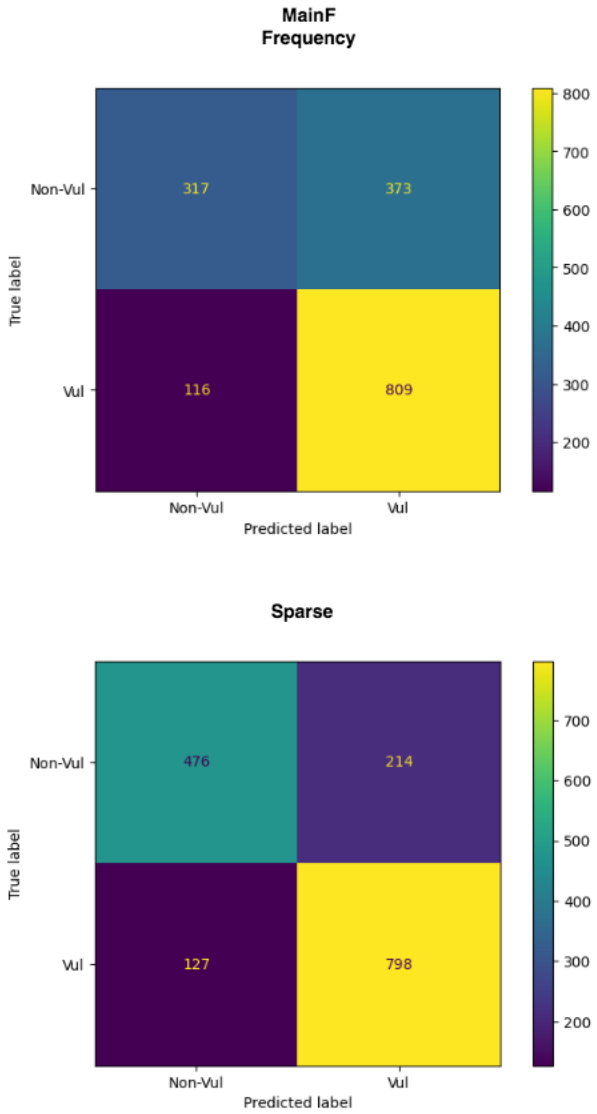

Figure 5.1: Binary Random Forest Classifier without hyper-parameter tuning Confusion Matrix for *MainF Frequency* and *Sparse* Datasets.

consideration these results. The same process, has been done for the Multi-Classifier Random Forest. As hyper-parameter tuning improves results and *Sparse* presents the best results, we have use it for our 3 Class (SQLI, XSS, Non-Vulnerable) Random Forest model.

The results obtained in this Random Forest experimentation section indicate that this model with the *Sparse* dataset available is most suitable for binary classification and is not useful for multi-class classification tasks for the type of vulnerability, as indicated the metrics recorded. The Random Forest model is not able to identify Non-Vulnerable data points when tasked with a multi-class prediction, however it is successful in doing it for

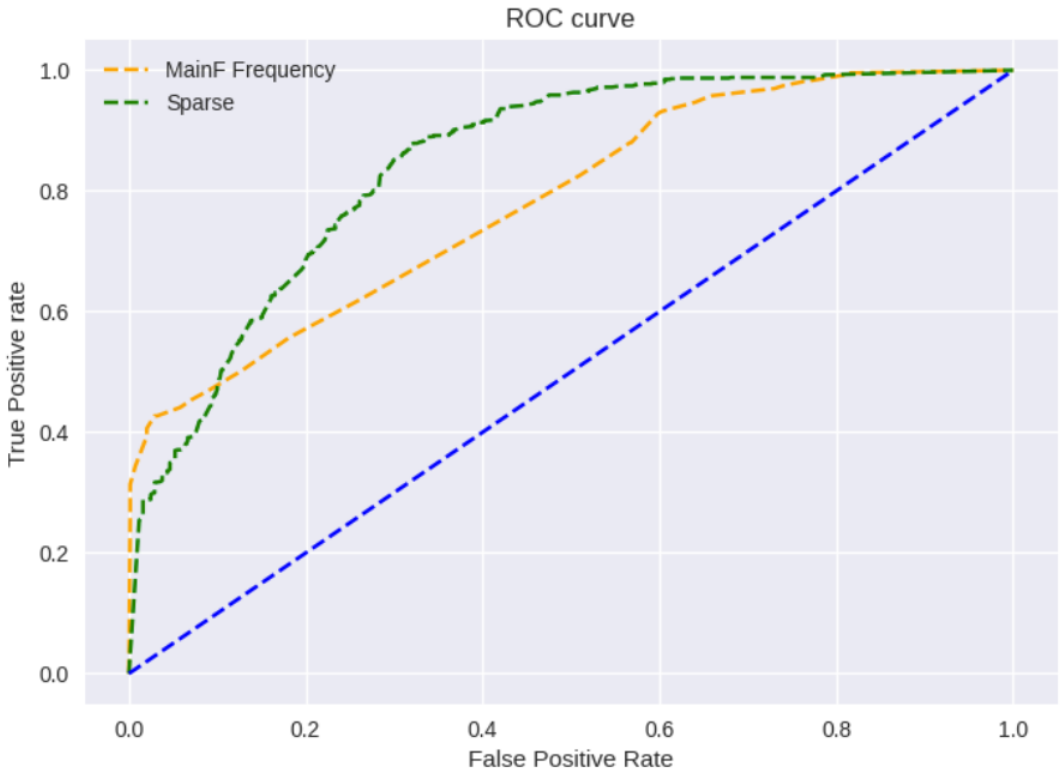

Figure 5.2: ROC Curve

the binary classification task, Table 5.4 presents results for the performance of the Random Forest binary classification with hyper-parameter tuning. The accuracy of 0.80 for binary classification using hyper-parameter tuning and of 0.67 (see Table 5.5) for multi-class classification show that the final architecture must rely on the Doc2Vec Logistic Regression model for classifying the type of vulnerability.

Table 5.4: Sparse Dataset Multi-Class Random Forest

Sparse Dataset max_depth = 9 n_estimators = 433 criteria = *Gini*

Vul Type	Precision	Recall	F-1 Score	Accuracy
Non-Vul	0.65	0.20	0.31	
SQLI	0.73	0.90	0.81	0.67
XSS	0.62	0.92	0.74	

Table 5.5: Random Forest Binary Classification with Hyper-parameter Tuning

	max_depth	n_estimators	criteria	Precision	Recall	Accuracy
MainF Frequency	10	315	*Gini*	0.69	0.70	0.71
Sparse	7	493	*Gini*	0.78	0.89	0.80

5.4 Evaluation of the Doc2Vec Model with Logistic Regression

In this final section of our architecture, we will test our Doc2Vec Model with the code snippets from the multi-class *SARD Dataset* that are labelled with the type of vulnerability. The snippets, are preprocessed as and tokenized as explained in Section 3.3 and the final tokens for each code snippet represent a document that is labelled with the type of vulnerability. In this section we will be experiment with the Doc2Vec Model parameters that include:

- *Training Algorithm* - *DM* is either $0, 1$ which refers to the model using the Distributed Memory algorithm if 1 and if 0 the *DPOW* (Distributed Bag of Words) algorithm
- *Vector Size* - This refers to the dimension of the output numeric vector of the D2V Model. We will experiment with 75, 150 and 300.
- *Epochs* - Number of iterations over the text corpus used. We will experiment with 10, 30 and 60.

We trained the Doc2Vec Model on the code snippet features with the different hyper-parameter combinations (see Table 5.6) and then used a simple linear regression model to predict the class of the vulnerability. We have observed that the *DPOW* model is more effective and in particular the smaller the size of the vector and iterations (epochs) over the the training text corpus, the better the performance. For the full architecture, we will therefore train the Do2Vec model with a vector size of 75 and iterate 10 times.

5.5 Results of full Architecture

Based on the results obtained in the previous sections, it has been observed that the performance of the hybrid model would be limited specially in the multi-class classification random forest model for the vulnerability type using the *Sparse* dataset. Therefore, we have removed this part from the final architecture. In addition, for the Random-Forest binary classification we will use the *Sparse* dataset instead as it yielded best results, these changes to the final architecture can be visualized on Figure 5.3. The hybrid model,

Table 5.6: Results for Different Hyperparameter Combinations of the Doc2Vec Model for Logistic Regression

DM	Vector Size	Epoch	Precision	Recall	Accuracy
1	75	10	0.65	0.67	0.64
	75	30	0.59	0.60	0.61
	75	60	0.72	0.71	0.73
	150	10	0.62	0.61	0.59
	150	30	0.77	0.77	0.75
	150	60	0.35	0.33	0.37
	300	10	0.62	0.63	0.62
	300	30	0.60	0.60	0.59
	300	60	0.66	0.67	0.68
0	75	10	0.88	0.88	0.87
	75	30	0.80	0.80	0.79
	75	60	0.50	0.44	0.47
	150	10	0.86	0.86	0.85
	150	30	0.58	0.56	0.55
	150	60	0.52	0.45	0.49
	300	10	0.88	0.88	0.87
	300	30	0.58	0.55	0.56
	300	60	0.46	0.42	0.45

which combines the strengths of multiple individual models, has shown promising results but also some areas for potential enhancement to justify a hybrid approach to vulnerability detection.

By analyzing the performance metrics (precision, recall, and accuracy) it became evident that certain combinations of models or hyperparameters yielded better results than others. The identification of these patterns and insights from the evaluation process have prompted adjustments to the hybrid model's architecture. The final architecture (see figure 5.7) will have the following hyper-paramenters for each model.

- OCSVM with *nu* = 0.1 and *poly* kernel, trained with *MainF Binary* Vulnerable data.

- Random Forest Binary Classifier with *max depth* = 7, *n estimators* = 493 and *criteria*= Gini

- Doc2Vec Model *vector size* = 75 trained with *epochs*= 10

	Precision	Recall	F-1 Score
Non-Vul	0.57	0.73	0.64
SQLI	0.67	0.50	0.57
XSS	0.80	0.69	0.74
Accuracy			**0.65**

Table 5.7: Final Full Architecture Classification Report

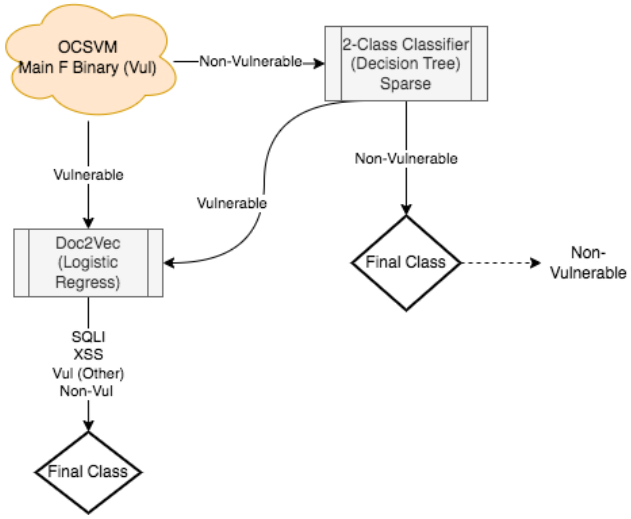

Figure 5.3: Final Hybrid Vulnerability Detection Model Architecture.

The final classification results seen on Table 5.7 demonstrate that the model achieves an overall accuracy of 65%, showing the approaches validity to predicting code snippets that are vulnerable to SQLI, XSS or not. The project relied on a limited set of data, yet it yielded a strong precision metrics for identifying XSS vulnerabilities while non-vulnerable snippets had also very interesting metrics. The results of the Doc2Vec and Logistic Regression Model in particular provide a good position for further study, while the OCSVM is very limited to the data available.

Chapter 6

Conclusion

The work carried out in this research project aims to provide a benchmark for the use of new approaches to vulnerability detection, in particular incorporating a type of intermediate language that is based on key functions that are defined apriori (e.g. *MainF Bin, Sparse Bin* as features and also using NLP models for word embedding that obtain features without prior human knowledge to define them). Due to the hybrid model architecture, both supervised, unsupervised and semi-supervised approaches were used, therefore this allowed for many approaches to be experimented with, both individually and simultaneously. The work carried out successfully produced a pre-processing method for cleaning source code snippets for the creation of relevant datasets. The evaluation of the models identified strong limitations on the OCSVM, as the model was not able to correctly identify most of the vulnerable data. It is important to note, that while the rest of the architecture ammended this limitation, the OCSVM can still be a relevant model to explore in future work due to its theoretical robustness, and frequent use in the cyber-security space. As the datasets become more representative and features become more relevant the OCSVM can still be an interesting approach to identify vulnerable from non-vulnerable behaviour. The random forest showed solid results, demonstrating that decision-trees are a great choice and adaptable to solve many problems in different fields including vulnerability detection.

Another optimistic point was the positive results of the NLP model such as Doc2Vec. This will point future research in this direction, where more up to date models that have been developed recently in the area of large language models and generative AI can provide a more robust solution to solving the problem of automotically detecting vulnerabilities in source code and preventing cyber-attacks. The final full architecture accuracy of 65% for all the target classes, shows that the model did not overfit and while it may seem relatively low it does indicate that with the limited database available the model was successfully in predicting most of the classes.

6.1 Future Work

The work carried out can be a basis for a lot of future research, below we will leave some possible directions that can hopefully build and provide further contributions to this topic:

- Focus on NLP models, in particular new advances on Large Language Models (LLM's), study and continue to develop feature extraction and vulnerability detection. Through the end phase of this thesis there has been big releases in the field of generative AI, a few open source models are available and can also provide an interesting solution to analyse source code and provide recommendations to increase safety that are context specific.
- Continue to build a database for code snippets, current work can be improved with the use of data augmentation techniques.
- Explore how the model can learn about new zero days vulnerabilities and incorporate this knowledge into the model.
- Providing model explainability where engineers can understand why the model has identified a vulnerability.

Bibliography

[1] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2016.

[2] P. E. Black, "A software assurance reference dataset: Thousands of programs with known bugs," *Journal of Research of the National Institute of Standards and Technology*, vol. 123, 2018. [Online]. Available: https://samate.nist.gov/SARD

[3] P. Flach, *Machine learning: the art and science of algorithms that make sense of data*. Cambridge University Press, 2012.

[4] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[5] K. P. Murphy, "Adaptive computation and machine learning series," in *Machine Learning: A Probabilistic Perspective*. University of British Columbia, 2012.

[6] D. M. Tax and R. P. Duin, "Support vector data description," *Machine learning*, vol. 54, pp. 45–66, 2004.

[7] S. O. Haykin, "Haykin, neural networks and learning machines," 2009. [Online]. Available: http://www.mif.vu.lt/~valdas/DNT~

[8] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[9] A. Fidalgo, I. Medeiros, P. Antunes, and N. Neves, "Towards a deep learning model for vulnerability detection on web application variants," in *In Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020, pp. 1–12.

[10] A. Figueiredo, T. Lide, D. Matos, and M. Correia, "Merlin: Multi-language web vulnerability detection," in *In Proceedings of the 2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, 2020, pp. 112–117.

[11] NTT, "2022 global threat intelligence report," Report, pp. 1–30, 2022. [Online]. Available: https://www.security.ntt/pdf/2022-global-threat-intelligence-report-v8.pdf

[12] N. N. Ibéria Medeiros and M. Correia, "Dekant: A static analysis tool that learns to detect web application vulnerabilities," *Proceedings of the 25th International Symposium on Software Testing and Analysis*, p. 1–12, 2016.

[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *In Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018, pp. 215–228.

[14] E. L. Bird, Steven and E. Klein, *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

[15] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, pp. 81–106, 1986.

[16] J. Han, M. Kamber, and J. Pei. (2012) Data mining concepts and techniques, third edition. Waltham, Mass.

[17] L. Breiman, "Classification and regression trees," *Machine learning*, vol. 5, no. 2, pp. 185–227, 1984.

[18] S.-W. Du, M.-C. Zhang, P. Chen, H.-F. Sun, W.-J. Chen, and Y.-H. Shao, "A multiclass nonparallel parametric-margin support vector machine," *Information*, vol. 12, no. 12, p. 515, Dec 2021. [Online]. Available: http://dx.doi.org/10.3390/info12120515

[19] A. Ram, S. Jalal, A. S. Jalal, and M. Kumar, "A density-based algorithm for discovering density varied clusters in large spatial databases," in *In Proceedings of the International Journal of Computer Applications*, vol. 3, no. 6, 2010, pp. 70–78.

[20] M. Daszykowski, B. Walczak, and D. Massart, "Looking for natural patterns in data: Part 1. density-based approach," *Chemometrics and Intelligent Laboratory Systems*, vol. 56, pp. 83–92, 05 2001.

[21] Z.-H. Zhou, *Ensemble methods: foundations and algorithms*. Taylor Francis, 2012.

[22] R. E. Schapire, "Explaining adaboost," in *Empirical inference*. Springer, 2013, pp. 37–52.

[23] S. Dua and X. Du, *Data mining and machine learning in cybersecurity*. CRC Press, 2016.

[24] G. Fernandes, J. J. P. C. Rodrigues, L. F. Carvalho, J. F. Al-Muhtadi, and M. L. Proença, "A comprehensive survey on network anomaly detection," *Telecommunication Systems*, vol. 70, no. 3, pp. 447–489, 2018.

[25] F. T. Liu, K. M. Ting, and Z. H. Zhou, "Isolation forest," pp. 413–422, 2008.

[26] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Identifying density-based local outliers," *ACM SIGMOD Record*, vol. 29, no. 2, pp. 93–104, 2000.

[27] Z. He, X. Xu, and S. Deng, "Discovering cluster-based local outliers," *Pattern Recognition Letters*, vol. 24, no. 9-10, pp. 1641–1650, 2003.

[28] B. Schölkopf, R. Williamson, A. Smola, J. Shawe-Taylor, and J. Piatt, "Support vector method for novelty detection," in *Advances in Neural Information Processing Systems*, 2001, pp. 582–588.

[29] H. J. Shin, D.-H. Eom, and S.-S. Kim, "One-class support vector machines—an application in machine fault detection and classification," *Computers Industrial Engineering*, vol. 48, no. 2, pp. 395–408, 2005.

[30] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1999.

[31] A. Drozd, A. Gladkova, and S. Matsuoka, "Word embeddings, analogies, and machine learning: Beyond king-man+ woman= queen," in *Proceedings of coling 2016, the 26th international conference on computational linguistics: Technical papers*, 2016, pp. 3519–3530.

[32] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[33] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2014.

[34] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, pp. 2015–2022, 2016.

[35] R. McConville, R. Santos-Rodriguez, R. Piechocki, and I. Craddock, "N2d: (not too) deep clustering via clustering the local manifold of an autoencoded embedding," *25th International Conference on Pattern Recognition*, Aug. 2019, 2020 25th International Conference on Pattern Recognition. ; Conference date: 15-01-2021.

[36] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection," *Journal of Open Source Software*, vol. 3, no. 29, p. 861, 2018.

[37] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," *Proceedings of the 13th conference on World Wide Web - WWW 04*, pp. 245–256, 2004.

[38] J. Kronjee, A. Hommersom, and H. Vranken, "Discovering software vulnerabilities using data-flow analysis and machine learning," in *In Proceedings of the 13th International Conference on Availability, Reliability and Security*, 2018, pp. 159–166.

[39] Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: http://jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf

[40] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous systems," *arXiv preprint arXiv:1603.04467*, 2016.