

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Machine learning approaches for API abuse detection

João Sá



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Dr. José Manuel De Magalhães Cruz

Second Supervisor: Eng. Sílvio Macedo

Third Supervisor: Eng. Xavier Silva

July 27, 2022

Machine learning approaches for API abuse detection

João Sá

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Rui Camacho

Referee: Prof. José Manuel De Magalhães Cruz

Referee: Prof. Rita Paula Almeida Ribeiro

July 27, 2022

Abstract

In today's world, the internet plays a crucial role in personal and professional daily life. People and businesses increasingly rely on the web for numerous functions. **APIs**, short for Application Programming Interfaces, define the rules and specifications on which the internet relies for information transfer. Due to their use-cases, APIs create an ever-changing attack surface specific to a business setting, requiring fast-adapting methods for detection of abusive behaviour.

Machine learning algorithms build models capable of capturing the training data underlying patterns allowing them to make informed predictions on unseen data; furthermore, as the training data changes, the models adapt. This dissertation explores supervised machine learning approaches for API abuse detection, consisting of the misuse of API functions.

The dataset used in this dissertation is extracted from real-world API traffic log data, facilitated by the proponent organization XAREVISION S.A., and includes business-specific API abusive patterns. The abusive patterns, initially only two data points, go through a data generation process designed to introduce data variability while ensuring its integrity, quality, and veracity. The dataset then undergoes a feature extraction process. The API traffic data is separated into time periods and transcribed into a tabular and an image format, with different feature spaces representing the API traffic during the respective period.

The state-of-the-art study supported using a Multilayer-Perceptron model for the factual tabular dataset format and a Convolutional Neural Network for the more interpretable, novel image format. The two models are part of this dissertation-developed API abuse detection system prototype decision core. The development of the system included the creation of several development artifacts.

The proposed machine learning approaches were evaluated against a with-held validation split of the training dataset via the appropriate evaluation metrics and obtained promising marks. Furthermore, the system was also the subject of a performance evaluation ensuring the system could function in a real-world scenario.

While the results validate the potential of ML-based API abuse detection systems, the design and implementation of this dissertation solution allowed an understanding of the several reasons contributing to the non-adoption of these systems: - API vulnerabilities are dynamic and specific to a business setting. - API endpoints can have variable parts that require collapsing. - Introducing an API vulnerability to the training dataset involves extracting many data points from it. These factors contribute to the primary and complex limitation of the developed body of work, the API traffic training dataset preparation, and maintainability. Despite this limitation, several exciting avenues of further work can be pursued, developing a solution that mitigates these issues and hopefully contributes to adopting such systems.

Keywords: API abuse, Machine Learning, Neural Network, Multilayer-Perceptron, Convolutional Neural Network.

Resumo

No mundo de hoje, a 'internet' desempenha um papel crucial no quotidiano pessoal e profissional. Pessoas e empresas dependem cada vez mais da 'web' para inúmeras funções. As **APIs**, acrónimo de Application Programming Interfaces, definem as regras e especificações das quais a 'internet' depende para a transferência de informação. Devido aos seus casos de uso, as APIs criam uma superfície de ataque específica para cada negócio e em constante mudança, requerendo métodos capazes de adaptação rápida para deteção de comportamento abusivo.

Os algoritmos de aprendizagem de máquina criam modelos capazes de capturar os padrões subjacentes dos dados de treino, permitindo a criação de previsões informadas em dados nunca antes vistos; além disso, à medida que os dados de treino mudam, os modelos adaptam-se. Neste sentido, esta dissertação explora abordagens de aprendizagem de máquina supervisionadas para deteção de comportamento abusivo em APIs.

O conjunto de dados usado nesta dissertação foi extraído de logs de tráfego de API do mundo real, facilitados pela organização proponente XAREVISION S.A., e inclui padrões abusivos de API específicos ao negócio. Os padrões abusivos, inicialmente apenas dois pontos de dados, passam por um processo de geração de dados desenhado para introduzir variabilidade nos dados, garantindo a integridade, qualidade e veracidade dos mesmos. O conjunto de dados passa então por um processo de extração de 'features'. Os dados de tráfego da API são separados em períodos de tempo e transcritos em formato tabular e de imagem, com diferentes espaços de 'features' representando o tráfego da API durante o respetivo período.

O estudo do estado da arte apoiou o uso de um modelo Perceptron Multicamadas para os factuais dados tabulares e uma Rede Neural Convolucional para o mais interpretável e inovador formato de imagens. Os dois modelos fazem parte do núcleo de decisão do sistema, o desenvolvimento do qual incluiu a criação de diversos artefactos de desenvolvimento.

As abordagens de aprendizagem de máquina propostas foram avaliadas em relação a uma divisão de validação por meio das métricas de avaliação apropriadas e obtiveram notas promissoras. Além disso, o sistema também foi objeto de uma avaliação de desempenho, garantindo que o sistema funciona em cenário de produção.

Enquanto os resultados validam o potencial dos sistemas de deteção de abuso de API baseados em ML, o design e implementação desta solução durante a dissertação permitiu compreender as várias razões que contribuem para a não adoção destes sistemas: - As vulnerabilidades de APIs são dinâmicas e específicas para uma configuração de negócios. - Os endpoints da APIs podem ter partes variáveis que necessitam de ser colapsadas. - A introdução de uma vulnerabilidade de API ao conjunto de dados de treino requer a extração de muitos pontos de dados dela. Estes fatores contribuem para a complexa limitação primária do corpo de trabalho desenvolvido, a preparação do conjunto de dados de treino e a sua manutenção. Apesar dessa limitação, vários caminhos interessantes de trabalho adicional podem ser explorados, desenvolvendo uma solução que mitiga

esses problemas e possa contribuir para a adoção de tais sistemas.

Agradecimentos

Agora no final do meu percurso académico marcado pelo término da dissertação no Mestrado de Engenharia Informática e Computação, quero agradecer primeiramente à minha família por me proporcionarem as melhores condições possíveis para aprender, estudar, e ser feliz. Em especial ao meu Pai por me inspirar a sempre fazer grandes coisas e sem o qual eu não teria metade do meu sucesso.

Um obrigado ao Engenheiro Sílvio Macedo que, infelizmente, não pôde observar o resultado do meu trabalho que nasceu de uma visão sua. Embarquei neste desafio que me lançou devido ao seu entusiasmo e conhecimento. Com o decorrer deste trabalho aprendi que para além de líder exigente, era uma pessoa preocupada com o bem-estar de todos, fascinada pela tecnologia e pela evolução da humanidade.

Um agradecimento também aos meus orientadores, Professor Doutor José Manuel De Magalhães Cruz e Engenheiro Xavier Silva por me guiarem, corrigirem os meus erros e me ajudarem a aprender durante o desenvolvimento deste projeto.

Esta dissertação foi desenvolvida na empresa XAREVISION S.A. e como tal tenho que agradecer a todos os meus colegas de trabalho pela sua disponibilidade no tema da minha dissertação.

Finalmente aos meus amigos, que sempre me ajudaram neste e noutros momentos da minha vida, um muito obrigado por tudo, que continuemos a fazer o mundo nosso.

Dedico este trabalho à minha querida falecida Mãe que certamente está algures feliz do que me tornei.

João Sá

"If I had seen further than others, it's because I stood upon the shoulders of giants."

Isaac Newton

Contents

1	Introduction	1
1.1	Context	2
1.2	Motivation	3
1.3	Goals	3
1.4	Document Structure	4
2	State of the Art	5
2.1	Machine Learning Background	5
2.1.1	Supervised Learning	5
2.1.2	Unsupervised Learning	9
2.1.3	Reinforcement Learning	11
2.1.4	Machine Learning Challenges	12
2.1.5	Learning Methodologies	12
2.1.6	Evaluation Metrics	13
2.1.7	Cross-Validation	15
2.1.8	Hyperparameter Tuning	15
2.1.9	Technologies for Machine Learning	16
2.2	Related Work	19
2.2.1	API Abuse Detection using Machine Learning	19
2.2.2	Network Traffic Anomaly Detection using Machine Learning	21
2.2.3	Malware Detection using Machine Learning	23
2.2.4	Literature Review Summary and Comparison	24
3	Problem and Proposed Solution	26
3.1	Problem	26
3.1.1	API Abuse	27
3.1.2	Problem Statement	28
3.2	Requirements Analysis	28
3.2.1	Domain Concepts	28
3.2.2	Functional Requirements	29
3.2.3	Non-Functional Requirements	30
3.3	Proposed Solution	31
3.3.1	Proposed System Architecture	31
3.3.2	Proposed System Flows	34
3.3.3	Proposed Formats Feature Spaces	39
3.3.4	Adopted Technologies	41

4	Implemented Solution	42
4.1	Architectural Components	43
4.1.1	Data Component	43
4.1.2	Analysis Component	48
4.2	Dataset-related Processes	50
4.2.1	Handling Numerical Features	50
4.2.2	Handling Categorical Features	51
4.2.3	Image Pre-processing	51
4.2.4	Training Validation Split	51
4.3	System Phases	52
4.4	Software Tests	53
4.4.1	Integration Tests	54
4.4.2	System Tests	55
5	Evaluation and Results	56
5.1	Test Hypothesis	56
5.2	Experimentation and Evaluation	57
5.2.1	Evaluation Methodology	57
5.2.2	Evaluation Metrics	57
5.2.3	Experiments Conditions	58
5.3	Dataset study	58
5.3.1	Overview	58
5.3.2	Exploratory Data Analysis	58
5.4	Network Topology and Hyperparameter Tuning	60
5.4.1	MLP Experiment	60
5.4.2	CNN Experiment	63
5.5	System Performance Evaluation	65
5.6	Results Discussion	65
6	Conclusions and Future Work	67
6.1	Achieved Requirements	68
6.2	Limitations and Further Work	69
	References	71
7	Appendix A	78
7.1	Exploratory Data Analysis	78
7.1.1	Total Dataset Analysis	78
7.1.2	Data Analysis of the Regular Label Split	80
7.1.3	Data Analysis of the API Abuse by volume split	82
7.1.4	Data Analysis of the API abuse by sequence split	83
7.2	MLP Bayesian Optimization Process Trials	85
7.3	CNN History in regards to the ILSVRC	87

List of Figures

2.1	Artificial neural network architecture overview.	6
2.2	Overview of a fully connected layer.	8
2.3	A visual representation of a convolutional layer and its kernel.	8
2.4	Overview of a SVM.	9
2.5	Autoencoder architecture overview.	11
2.6	Reinforcement learning, agents and environment visualization.	11
2.7	Receiver operating characteristics curve examples.	15
3.1	Example of API abuse by volume.	27
3.2	Example of API abuse by sequence.	28
3.3	Domain model for an API traffic abuse detection system.	29
3.4	Deployment diagram of the proposed system execution architecture	32
3.5	Component diagram of the proposed system.	33
3.6	Sequence diagram of the pre-condition logging flow.	34
3.7	Sequence diagram of the ingestion flow.	35
3.8	Sequence diagram of the training flow.	36
3.9	Sequence diagram of the update flow.	37
3.10	Sequence diagram of the prediction flow.	38
3.11	Sequence diagram of the analysis flow.	39
3.12	Image feature space example representation.	41
4.1	Beta distribution probability density and cumulative distribution functions.	45
4.2	Image representation of the original API abuse by volume pattern.	46
4.3	Image representation of the original API abuse by sequence pattern.	46
4.4	Class diagram of the feature extraction module.	47
4.5	Dataset obfuscated image format representation.	48
4.6	Class diagram of the traffic analysis module.	49
4.7	State machine diagram of the system's training phase.	52
4.8	State machine diagram of system's serving phase.	53
4.9	Integration testing run output.	54
4.10	System testing run output.	55
5.1	Column chart for feature nr_total_previous_time_periods analysis.	59
5.2	Column chart for feature nr_total_api_requests analysis.	60

List of Tables

2.1	Machine learning technologies comparison.	19
2.2	Related work summary.	25
3.1	Tabular feature space example representation.	40
4.1	Original log data format.	44
4.2	Intermediate CSV data format.	44
4.3	Final CSV data format.	44
4.4	Dataset obfuscated tabular format representation.	48
5.1	Hyperparameters search spaces.	62
5.2	Bayesian optimization process best performing hyperparameter configurations. .	62
5.3	Best performing hyperparameter configurations evaluation metrics.	62
5.4	Experiment architectures and evaluation metrics.	64
5.5	System's periodic traffic analysis procedure execution time.	65
5.6	MLP and CNN models in the prototype system.	65
7.1	Descriptive statistics about the total experiment training dataset.	78
7.2	Descriptive statistics about the regular label split.	80
7.3	Descriptive statistics about the API abuse by volume split.	82
7.4	Descriptive statistics about the API abuse by sequence split.	83
7.5	Bayesian optimization process trials.	86

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
AUC	Area Under the ROC Curve
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CS	Computer Science
CSV	Comma Separated Value
DB	Database
DDoS	Distributed Denial of Service
DL	Deep Learning
DNN	Deep Neural Network
DoS	Denial of Service
ELM	Extreme Learning Machine
GPU	Graphics Processing Unit
HAT	Hoeffding Adaptive Trees
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection Systems
ILSVRC	ImageNet Large-Scale Visual Recognition Challenge
IP	Internet Protocol
IPS	Intrusion Prevention systems
IT	Information Technology
KDD	Knowledge Discovery and Data Mining tools
KPI	Key Point Indicators
MAE	Mean Absolute Error
ML	Machine Learning
MLP	Multilayer-Perceptron
MNIST	Modified National Institute of Standards and Technology
MSE	Mean Squared Error
NAS	Neural Architecture Search
NN	Neural Network
OS	Operating System
OWASP	Open Web Application Security Project
PII	Personable Identifiable Information
R2L	Remote to Local
RF	Random Forests
RFC	Request for Comment

RL	Reinforcement Learning
RMSE	Root Mean Squared Error
ROC	Receiver Operating Characteristics
SVM	Support Vector Machine
TCP	Transmission Control Protocol
U2R	User to Root
UC	Use Case
UDP	User Datagram Protocol
UID	Unique Internal Identifier
UML	Unified Modeling Language
k-NN	k-Nearest Neighbour

Chapter 1

Introduction

“I am convinced that there are only two types of companies: those that have been hacked and those that will be. And even they are converging into one category: companies that have been hacked and will be hacked again” - Robert S. Mueller, III ex-Director of the Federal Bureau of Investigation [1]

Technology is causing a digital business transformation phenomenon by breaking down barriers between people, businesses, and things, building new models, processes, and systems that result in competitive advantage [2]. According to a Cisco report, [3], this digital disruption can potentially reshape markets faster than perhaps any force in history.

Application Programming Interfaces (APIs) are a way for two processes to communicate with one another over a medium [4]; they define sets of rules and specifications for software programs to interact with [5]. APIs allow companies to expose their services to the outside world; because of this, they have increasingly become a target for attackers [6].

According to a report by McAfee and the Center for Strategic and International Studies [7], cybercrime is now a nearly trillion-dollar industry, forcing companies to invest in security solutions. Industry analysts Gartner [8] forecasted worldwide Information Technology (IT) security spending to exceed \$150 billion by the end of 2021, a 6.4% increase from the previous year, and this annual growth is expected to endure at least through 2025 [9].

The cyber-security field is yet another where Machine learning (ML) algorithms are gaining notoriety. These algorithms learn from the underlying patterns in training data to make predictions or decisions on yet-unseen data [10].

This chapter discusses the context and motivation of the dissertation, its goals, and the document's organization.

1.1 Context

APIs are at the heart of digital business transformation because they allow companies to expose their services to the outside world. At the same time, APIs vulnerabilities may expose application logic, Personable Identifiable Information (PII), and provide an entry point for attackers. The Open Web Application Security Project (OWASP) has created a Top 10 detailing unique vulnerabilities and security risks of APIs and strategies or solutions to understand and mitigate them called API Security. The most recent is the 2019 API Security Top 10 [6].

Furthermore, APIs can generate massive API traffic, which may be challenging to analyze via traditional methods such as Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) to detect any potential abnormal behavior.

Existing IDS and IPS solutions generally apply rule-based and signature-based techniques to analyze traffic at the packet level. In contrast, API traffic abnormalities may only be apparent and detectable with a more extensive overview of the business logic, and a higher-level analysis [11].

Naturally, the difficulty of preventing all the known attack vectors is very severe and only increases when factoring in previously unknown attack vectors. Detecting the attacks as fast as possible is sometimes the only course of action.

Machine learning is the scientific study of algorithms and statistical models that computer systems use to perform a specific task without being explicitly programmed [12]. ML algorithms have been applied successfully in diverse fields, ranging from pattern recognition, spacecraft engineering, finance, and computational biology [13].

A supervised machine learning algorithm is a computational process that uses input data to achieve the desired task without being programmed; these algorithms automatically alter or adapt their architecture through repetition to become better and better at achieving the desired task. During training, the adaptation process happens; the algorithm receives input data samples and their respective outcomes. It then optimally configures itself to produce the desired outcome when presented with the respective inputs, even from new, previously unseen data [13].

Analyzing API traffic data via supervised machine learning algorithms is an exciting avenue of research. If the application of these algorithms to the API Abuse field demonstrates high detection marks and efficient data pre-processing and analysis steps, allowing for the detection of attacks within a reasonable time. They could enable the mainstream adoption of API Abuse detection systems.

Furthermore, supervised machine learning algorithms lend themselves to analyzing tabular and image data. Convolutional Neural Networks (CNN), an algorithm specializing in image recognition tasks, is currently the state of the art in the latter camp. The possibility of analyzing API traffic via image data by translating API traffic to an interpretable image is a novel idea, and its performance versus the traditional tabular approach is yet another interesting explored question.

1.2 Motivation

APIs work as a connector between systems and applications, and their adoption is increasingly becoming widespread; however, they process a wide variety of sensitive PII such as user credentials and payment information, making them of significant interest to attackers.

A report by Radware [14] reports that between 2020 and 2021, approximately 55% of the surveyed organizations experienced a Denial of Service (DoS) attack against their APIs, 49% experienced some form of injection attack, and 42% some element or attribute manipulation, noting that these are monthly statistics. Naturally, ensuring API security is of utmost importance for these organizations, which rely on APIs to offer their services.

The proposal and the development of a prototype system capable of detecting API abuse in a business setting and the investigation of machine learning approaches, which are capable of generalizing on the API abuse problem, may be relevant in developing the field. There are, however, significant challenges alongside such a system.

The first challenge is the dataset; while there are readily available datasets in the similar network traffic anomaly detection field, according to [15] these datasets do not satisfy the requirements of API traffic. The business setting under which this dissertation materialized possesses a microservice architecture that allowed for the capture of an API traffic dataset from microservice log records, further motivating the development of such a system.

To maximize the applicability of such a system, the ML approaches should be relevant in the maximum possible number of API architectures. This dissertation applies a two-pronged approach by pre-processing the dataset into two formats, a tabular and an image format, each having its own feature space. The novel image format, originally applied to malware detection by [16], offers an easily interpretable data format; this interpretability has the intrinsic value of allowing better visualization of API traffic set by a system administrator. Naturally, the tabular data format offers a more factual representation of the data at the cost of interpretability.

This two-pronged approach requires two different ML models, one capable of recognizing underlying patterns in tabular data and the other capable of detecting abnormalities in the API traffic images. The evaluation of these methods allow the reader to understand their potential. Furthermore, the classifications in the proposed system are combined to form a more refined decision about an API traffic set.

1.3 Goals

This dissertation seeks to develop and evaluate an ML-based system for API traffic abuse detection, capable of integrating with existing business microservice architecture. The system's detection rate and execution efficiency evaluate its existence in a business setting. The detection rate allows to realize the potential of such methods, and the time efficiency allows the system administrator to take adequate measures against such attacks.

The dissertation goals are further enumerated:

1. Evaluate, using the proper evaluation metrics, the hypothesis that ML algorithms can successfully apply to the API abuse detection problem.
2. Develop a system capable of working with existing microservice architectures without interfering with its normal functioning, provided the API traffic data records include the necessary information for feature extraction.
3. In the developed system, the API traffic analysis should run efficiently to detect anomalies in an opportune time range to allow the system administrator to prevent continuous API abuse.

1.4 Document Structure

This dissertation comprises six chapters: Introduction, State of the Art, Problem and Proposed Solution, Implemented Solution, Experimentation and Results and Conclusions and Future Work.

The Introductory chapter briefly contextualizes the dissertation, its motivation, and its goals.

The State of the Art chapter contains two main sections: the Machine Learning Background section, which summarizes ML background knowledge concepts as per the literature, and the Related work section, which reviews papers that apply ML to the same or similar problems.

The Problem and Proposed Solution chapter comprise the Problem section and the Proposed Solution section, with an intermediate Requirement Analysis section. The Problem section discusses the API abuse problem and presents the problem statement. A Requirement Analysis of the API abuse system is detailed before the Proposed Solution section. The Proposed Solution section shows the architecture for the proposed API abuse detection system, the system's flows, and the ML proposed approaches to the API abuse problem.

The Implemented Solution chapter describes the dissertations implemented solution consisting of an API abuse detection prototype system with an ML-based approach, including its components, modules, and processes that come together as Use Case realizations. It also describes the specific dataset-related processes that the ML-based solution requires and the developed software tests to ensure system correctness.

The Evaluation and Results chapter systematically describes the experimentation and evaluation methods conducted to evaluate and fine-tune the proposed ML-based approaches and the developed system performance, in addition to presenting and discussing the results.

The Conclusions and Future work chapter provide an overview of the dissertations' traversed path, followed by the analysis of the dissertation's achieved requirements. A section containing the dissertation limitations and potential further work finalizes the document.

Chapter 2

State of the Art

This dissertation employs machine learning in the API abuse problem. Section 2.1 provides machine learning background knowledge for this dissertation. The topics and technologies detailed are expected to be relevant for the developed work. Section 2.2 analyzes current machine learning solutions for the topics of malware detection, API abuse, and network anomaly traffic detection.

2.1 Machine Learning Background

Machine learning is a field of Artificial Intelligence (AI) and Computer Science (CS) [17], where learning algorithms are trained and used to make predictions from provided data, to learn and execute specific tasks without specific programming [12].

Building a high-quality machine learning model is iterative and time-consuming, involving the experimentation of different algorithms and techniques and effectively tuning their hyperparameters [18].

According to the literature, the three major categories of Machine Learning are: Supervised Learning, Unsupervised Learning, and Reinforcement learning [17].

2.1.1 Supervised Learning

Supervised Learning refers to the problem where the data is on the form:

$$\begin{aligned} \{x_i, y_i\}_{i=1}^n \\ n \in N \end{aligned} \tag{2.1}$$

where x_i denotes inputs and y_i denotes outputs.

This data form is called labeled data, where each data point has an input x_i and an output y_i , which explicitly explains "what we see in the data" [19]. Supervised learning problems divide into

two types: classification and regression [19]. Whether a problem is about regression or classification depends only on its output. In regression, the output is quantitative, and in classification, the output is qualitative. The input can be either quantitative or qualitative in both cases [19].

The sub-sub sections below present the most relevant supervised learning algorithms to our dissertation by the literature.

2.1.1.1 Artificial Neural Network

Artificial Neural Network (ANN), more commonly referred to as Neural Network (NN), is a supervised learning technique.

According to [17] NNs have three distinct components: the input layer, the output layer, and the hidden layer or layers. The first layer is responsible for processing the input and feeding the network with initial data. The output layer is in charge of producing the final result. The hidden layer(s) correspond to the layers between the input and output layers, where most communication occurs.

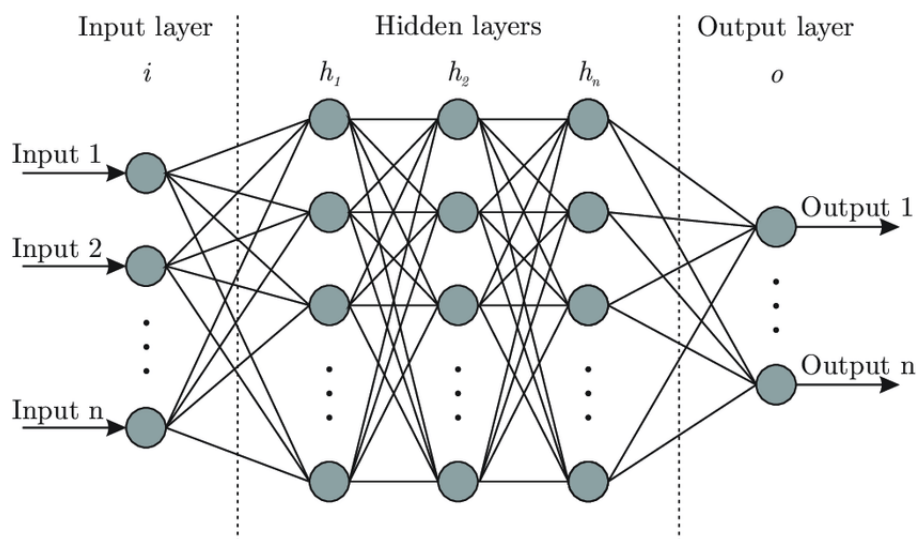


Figure 2.1: Artificial neural network architecture overview [20].

In a neural network, neurons take as input a weighted sum of the incoming outputs produced by the neurons connected to their incoming edges [17]. During the training phase, the synaptic weights rearrange to attain the maximum reduction in the difference between the labeled and the prediction value; this process is called backpropagation and is typically done via stochastic gradient descent [21].

It is then possible for a neural network to modify its topology via the adaptation of synaptic weights during the training process [21]. The same is true for the ML developer; using the acquired expertise, the developer can change the composition of the neural network layers, including removing and adding new ones.

NNs containing no hidden layer, also called Perceptrons, are only capable of solving linearly separable problems [22].

NNs containing one or more hidden layers, also called Deep Neural Networks (DNNs), are capable of solving non-linearly separable problems [22].

Three types of ANNs exist according to the connection patterns between the layers [22]. The fully connected type is where every network neuron has a connection to every neuron of the previous and subsequent layers. The partially connected type, where neurons connect to only some neurons of the previous and next layer. The locally connected type, where the neurons connected to a neuron are from a defined region (e.g., two parallel layers in a DNN merging).

Furthermore, ANNs can be of two types depending on the internal flow of information [22]: The feedforward type, where the information flows layer to layer in sequential order, and the feedback type, where feedback connections allow for a neuron to receive the output of another neuron in the same layer, or in a posterior layer.

The Multilayer Perceptron (MLP) is a feedforward, fully connected DNN [22]. An MLP traditionally deals with tabular data because image data may be highly complex to compute using only fully connected feedforward layers, depending on the image dimensionality, features and complexity, and the number of classes. The image dimensionality influences the input number of connections. The image features and complexity influence the number of hidden layers and neurons required to solve the problem. The number of classes influences the output number of connections. The number of neuron connections present in an MLP for which the input of the image type is in the form of an equation in 2.2 In represents the image dimensionality. Hn and hln represent the number of neurons in the hidden layer and the number of hidden layers, which depend on the image features and complexity (more complex images require more processing in the form of more neurons and more hidden layers). On represents the number of output classes.

$$(In * Hn_1) + (Hn_i * Hn_{i+1})^{hln} + (Hn_{hln} * On), i = [1, hln], i \in N \quad (2.2)$$

To solve the image data complexity problem, a new subtype of DNN has emerged - convolutional neural networks. CNNs incorporate into their architecture image-specific layers, and connection types [23].

Another subtype of DNNs is the Extreme Learning Machines (ELMs). They are feedforward DNNs, where the number of hidden layers, the number of neurons in each layer, and the hyperparameters of each layer are automatically assigned. Furthermore, ELMs do not use gradient descent-based backpropagation to work. Instead, they use Moore-Penrose generalized inverse to determine the weights of the neurons [24][25].

The layers of a NN can be of several types. This dissertation reviews three types: Fully Connected layers, Convolutional layers, and Pooling layers.

Fully Connected layer In fully connected layers, every node is directly connected to every node in both the previous and the next layer [26]. Figure 2.2 represents an example of a Fully Connected Layer.

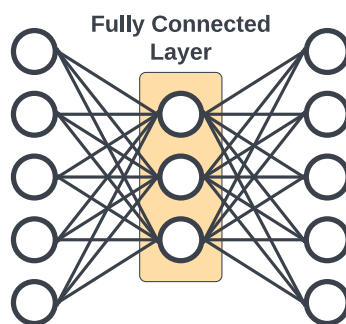


Figure 2.2: Overview of a fully connected layer.

Convolutional layer Convolutional layers apply a convolution operation to the input before passing the output result to the next layer. The layer's parameters focus on the use of learnable kernels. Every kernel will have a corresponding activation map to be stacked against the passed image as often as required along its dimensions [27], Figure 2.3 shows a visual representation of a convolutional layer and its kernel.

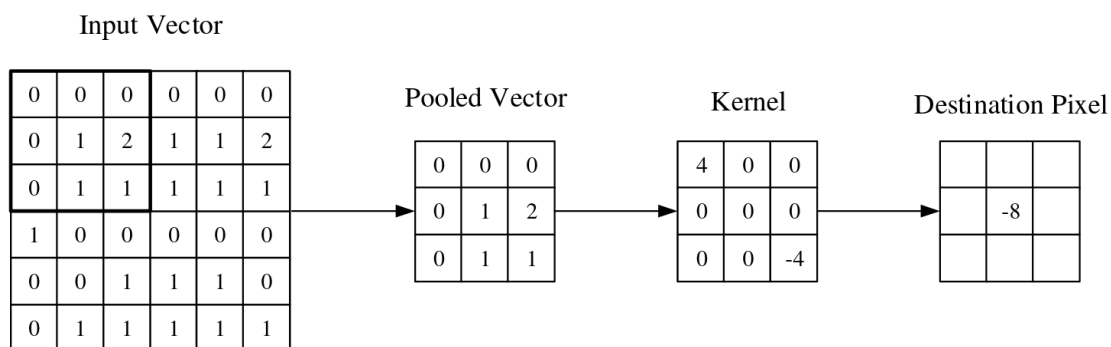


Figure 2.3: A visual representation of a convolutional layer and its kernel [27].

Convolutional layers introduce three hyperparameters:

- The **depth** or the number of filters hyperparameter, as the number of filters increases, so does the output's depth, increasing the model's pattern recognition and complexity via the number of neurons [27].
- **Stride** is the distance or the number of pixels that the kernel moves over the input matrix [27].
- **Zero-padding** is the behavior taken when the filters do not fit the input image [27].

Pooling layer Pooling layers analogously to convolutional layers reduce the image representation's dimensionality and, by doing so, the model's computational complexity. The pooling

operation applies the filter across the input matrix, but unlike in a convolutional layer, the kernel does not hold any weights; instead, it applies an aggregation function to the input values [27].

2.1.1.2 Support Vector Machine

A Support Vector Machine (SVM) is a supervised ML algorithm.

It works by creating a decision boundary known as the hyper-plane between points of the classes. The hyper-plane is as far as possible from the closest data points from each class. The closest data points from each of the classes are called support vectors. The separation distance is called margin [28]. The hyper-plane is N-dimensional, with N being the number of features of each class. The margin maximization process ensures that future data points will be classified correctly [29].

In figure 2.4 we see a components overview of an SVM with the described concepts of Support Vectors, Hyper-plane and Margin. The illustrated figure entails a problem with the classes having two features and thus being able to be drawn in a 2-D space.

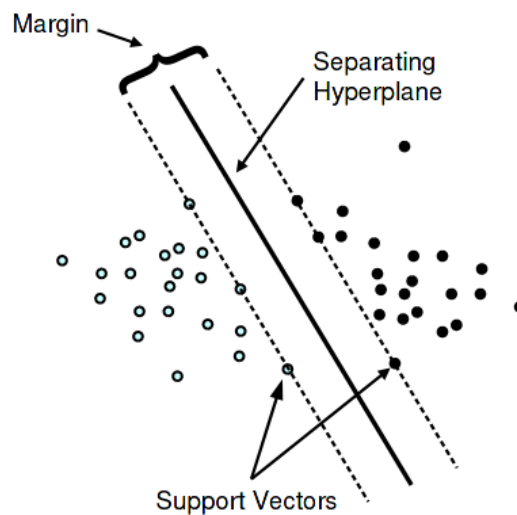


Figure 2.4: Overview of a SVM [30].

2.1.2 Unsupervised Learning

One of the main problems of supervised learning is that it requires labeled data. Finding a suitable dataset can sometimes be challenging because labeling data is often expensive and challenging.

This situation has motivated the development of unsupervised learning methods which only require the input data to be in the form:

$$\begin{aligned} \{x_i\}_{i=1}^n \\ n \in N \end{aligned} \tag{2.3}$$

that is so-called unlabeled data [19].

Unsupervised learning is a machine learning camp where algorithms find previously unknown patterns in unlabeled data [22].

The following sub-sub sections discuss this dissertation's most relevant unsupervised learning algorithms.

2.1.2.1 Clustering

Clustering is a technique to transform unlabeled data into clusters based on points of similarity, allowing a more extensive understanding of a subject [31]. The resulting clustered information may be of use to label the data.

The literature [31], determines the following types of clustering:

Hierarchical Clustering A Hierarchical Clustering Algorithm creates, from a proximity matrix, a sequence of nested partitions [22].

Regarding Hierarchical Clustering, there are two possible approaches: Agglomerative Hierarchical Clustering, which begins with n clusters of one object and groups clusters successively until there are n objects in k clusters, and Divisive Hierarchical Clustering, which begins with a single cluster composed of all the objects and successfully divides them into smaller sets [31].

Partitional Clustering Partitioning algorithms rely on specifying an initial number of groups and iteratively reallocating objects among groups to convergence. This algorithm typically determines all clusters at once [31].

Density Clustering Density-based clustering algorithms can discover arbitrary-shaped clusters. [22] In this approach, a cluster is a region in which the density of data objects exceeds a threshold [31].

Grid-based Clustering The grid-based clustering approach employs a grid data structure. It defines the space as a finite number of cells that form a grid structure on which all the operations for clustering apply [31].

Model-based Clustering Model-Based Clustering methods attempt to optimize the fit between the given data and some mathematical model. Such methods assume that a mixture of underlying probability distributions generates the data. Model-Based Clustering methods follow two major approaches: Statistical Approach and the Neural network approach [31].

2.1.2.2 Auto-Encoders

Autoencoders are yet another type of NN where the number of neurons in the input and output layers are the same, connected via a bottleneck. This bottleneck creates an encoding and decoding process.

During the training process, the autoencoder can learn more efficient representations of the data set [32].

Figure 2.5 shows a visual representation of the autoencoder architecture.

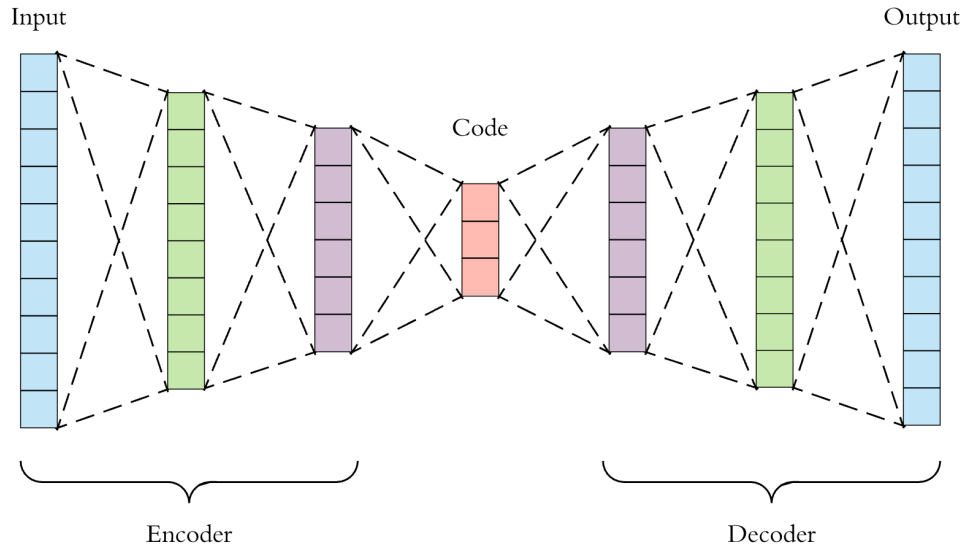


Figure 2.5: Autoencoder architecture overview [33].

2.1.3 Reinforcement Learning

Reinforcement learning (RL) refers to the problem where the goal is to maximize some performance criterium in a system.

The RL systems model two types of "actors": the environment and the agents. The environment refers to a set of states, and the agent(s) refers to an entity capable of making decisions according to the environment state [34].

Figure 2.6 illustrates the two RL interveners. The agents are responsible for taking action. The environment is responsible for updating the agents with a reward signal and an update on the environment state; this is a continuous process where actions get chosen according to the feedback provided by the environment.

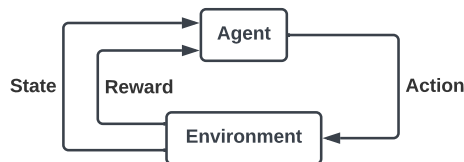


Figure 2.6: Reinforcement learning, agents and environment visualization after [35].

These systems aim to learn an action strategy, which consists of discovering a sequence of actions that maximize the cumulative reward provided by the environment [36].

Its Policy algorithm determines the actions an agent chooses. The policy algorithm can be simpler or more complex, depending on the situation. Typical ways of exploring the policy space include neural networks, stochastic algorithms, and genetic algorithms [36].

2.1.4 Machine Learning Challenges

Machine Learning systems may face different challenges during training. For example, Data quantity or quality, limitations regarding the approach, or training time. According to the literature, the subsections below present some of the most frequent challenges machine learning systems face.

2.1.4.1 The Data Quality Problem

Lack of data quality manifests in several forms, including data that is missing, incomplete, inconsistent, inaccurate, duplicate, and dated [37] that may lead to faulty predictions. Therefore, we need to ensure that data preprocessing which includes removing outliers, filtering missing values, and removing unwanted features, is done with the utmost care.

2.1.4.2 The Over-fitting and Under-fitting Problem

Over-fitting is whenever the model fits too closely to the training by fitting the noise in the data and memorizing various peculiarities of the training data rather than finding a general predictive rule [38].

Under-fitting is the opposite of over-fitting and occurs when the model is incapable of capturing the variability in the data and thus also not being able to generalize, resulting in low accuracy predictions [39]

Over-fitting solutions include introducing regularization methods, cross-validation, and reducing the model complexity. On the opposite side, under-fitting solutions include removing excess regularization methods and increasing the model complexity.

2.1.5 Learning Methodologies

The learning criteria can also classify machine learning systems, whether or not the system can learn incrementally from a stream of incoming data [36]. This section discusses offline and online learning techniques.

2.1.5.1 Offline Learning

Offline learning (also known as **batch learning**) is where the system trains using all the available data. This process requires considerable time and extensive computational resources. The system is trained, launched into production, and runs without learning anymore [36].

To "update" an already trained offline learning model with new training points, it is necessary to train a new version of the system from scratch with the new dataset and then replace the old

model with the newly trained model. Because this process can take a long time to execute, it is usually done weekly on commercial systems [40].

2.1.5.2 Online Learning

In online learning, the system trains incrementally, consuming data either individually or in small groups called batches [40]. The system can learn about new data on the fly as it arrives [36].

Online learning systems are usually applied when needed to adapt to change rapidly or autonomously. If the computational resources are limited, once the system has learned about new data instances, they may be discarded (at the cost of not being able to roll back to a previous state and "replay" the data) [36]. However, since these systems constantly consume new data, their performance may suffer from incorrect data [40].

2.1.6 Evaluation Metrics

Machine learning-based systems can be evaluated according to different mathematical metrics to evaluate system quality. As stated previously, Machine Learning problems fit into two classes depending on their output: regression (output is quantitative) and classification (output is qualitative), each having its accepted evaluation metrics by the literature.

2.1.6.1 Evaluating Regression

When evaluating regression algorithms, errors can be calculated based on the distance between predicted and known values from the training set [22]. According to [22] the most common metrics used when evaluating regression algorithms are the Mean Absolute Error (MAE), the Mean Squared Error (MSE), and the Root Mean Squared Error (RMSE). These metrics are always positive, and lower values correspond to models with better generalizing abilities.

Mean Absolute Error The arithmetic average of the absolute errors uses the same scale as the measured data [41].

$$MAE = \frac{1}{N} \sum_{i=1}^n |x_i - y_i| \quad (2.4)$$

Mean Squared Error The average of the errors' squares incorporates the estimator's variance and its bias (how far off the average estimated value is from the actual value) [42].

$$MSE = \frac{1}{N} \sum_{i=1}^n (x_i - y_i)^2 \quad (2.5)$$

Root Mean Squared Error The square root of the average of squared errors, the effect of each error on RMSE is proportional to the size of the squared error [41].

$$RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^n (x_i - y_i)^2} \quad (2.6)$$

2.1.6.2 Evaluating Classification

When evaluating classification algorithms, metrics are based on the correctness and incorrectness of the prediction when compared to the remaining training set. Four indicators evaluate the performance of these algorithms [22]:

- **True Positives (TP)** - Number of examples where the model correctly predicts the positive class.
- **True Negatives (TN)** - Number of examples where the model correctly predicts the negative class.
- **False Positives (FP)** - Number of examples where the model wrongly predicts the positive class.
- **False Negatives (FN)** - Number of examples where the model wrongly predicts the negative class.

The sections below present some of the most relevant measures of classification algorithms combining the four evaluation indicators according to the literature [17].

Accuracy The accuracy metric indicates the proportion of the number of correct trials to the total number of trials [43].

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} = \frac{TP + TN}{TP + TN + FP + FN} \quad (2.7)$$

Precision The precision metric determines what proportion of identifications was correct [44].

$$Precision = \frac{TP}{TP + FP} \quad (2.8)$$

Recall The recall metric determines what proportion of actual positives was correctly identified [44].

$$Recall = \frac{TP}{TP + FN} \quad (2.9)$$

F1-Score The F1-Score is the harmonic mean of the precision and Recall giving equal importance to both [44].

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \quad (2.10)$$

Receiver operating characteristics Curve The Receiver operating characteristics curve, more commonly known as the ROC curve, comes from communications theory [19].

Classification algorithms determine whether the predicted value belongs to a class based on the defined probability threshold represented by t .

A ROC curve is created by drawing the true positive rate against the false-positive rate for all values of $t \in [0, 1]$. The curve typically looks as shown in Figure 2.7. A ROC curve for a perfect classifier (always predicting the correct value with total certainty) touches the upper left corner, whereas a classifier, which only assigns random guesses, gives a straight diagonal line. A compact summary of the ROC curve is the AUC, area under the ROC curve. From Figure 2.7, a perfect classifier has an AUC of 1, whereas a classifier that only assigns random guesses has an AUC of 0.5 [19].

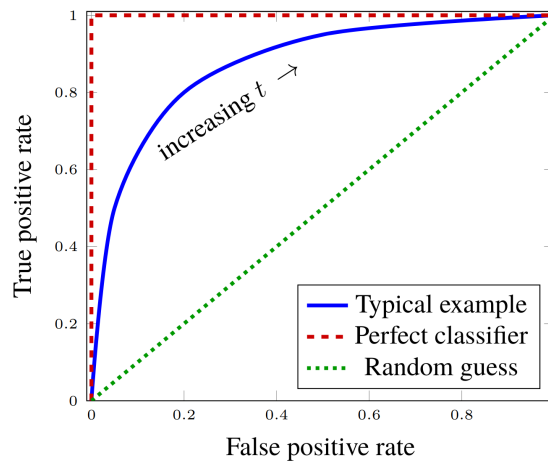


Figure 2.7: Receiver operating characteristics curve examples [19].

2.1.7 Cross-Validation

Cross-Validation is a statistical method of comparing and evaluating learning algorithms by dividing data into segments where the segments must cross over in successive rounds so that each data point has a chance of being validated. Each data point will be used for training and validation in different rounds [45].

K-Fold Cross Validation is the most common type of cross-validation used in machine learning. First, the data is partitioned into k equally (or nearly equally) sized segments or folds. Subsequently, k iterations of training and validation run such that within each iteration, a different fold of the data is held-out for validation, and the remaining $k-1$ folds serve to train the model [45].

2.1.8 Hyperparameter Tuning

Hyperparameters are the tunable parameters in Machine learning and optimization systems. They are in the form of network architecture (what layers), depth of the network (number of layers),

optimization algorithm, and others. The goal of the ML specialist is to tune them to minimize the error, in other words, to have the predictions as close as possible to actual values [46].

This dissertation discusses relevant approaches for hyperparameter tuning.

2.1.8.1 Manual Search

Manual search is the traditional approach to hyperparameter tuning. The selected values for the HyperParameters of a model are selected based on experience and intuition. The model is trained and evaluated on these parameters. This process repeats with another set of values until the achievement of satisfactory results [47].

2.1.8.2 Grid Search

In grid search, there are possible values or ranges for the hyperparameters, and experiments are run iteratively through all the possibilities, so in this method, every possible combination is tested [46]. This process can be very time-consuming and does not scale very well. However, it is guaranteed to find the optimal set of hyperparameters from the set [46].

2.1.8.3 Random Search

Similar to grid search, in random search, there are possible values or ranges for different hyperparameters. However, instead of running through all the possibilities, in Random Search, we try a random subset of possible combinations. This random subset may not include optimal. It is, however, much more efficient and will more often than not return as good results as grid search [48].

2.1.8.4 Bayesian Optimization Search

Defining the accuracy/error ML model as $f(x)$, where X is possible values or ranges for the different hyperparameters, then in bayesian optimization, the interest is in finding the minimum of a function $f(x)$ on some bounded set X . Under the hood, constructs a probabilistic model for $f(x)$ and then exploits this model to decide where in X to evaluate the function next while integrating uncertainty. The basic philosophy is to use the information available from previous evaluations of $f(x)$ [49].

2.1.9 Technologies for Machine Learning

Several technologies or frameworks exist to help develop machine learning solutions, reducing the effort to develop and deploy machine learning applications. Depending on the framework, they typically provide implementations for most state-of-the-art algorithms.

For this dissertation, we review some of the most commonly used ones.

2.1.9.1 TensorFlow

TensorFlow¹ is an open-source software library for numerical computation using data flow graphs created and maintained by Google.

TensorFlow is suitable for large-scale distributed training and inference. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them [50].

TensorFlow backend is in C++², but its APIs are available for Python³ and C++, with Java⁴, Go⁵, R⁶, and Haskell⁷ developments. Tensorflow can run models on both the Central Processing Unit (CPU), and Graphics Processing Unit (GPU), mobile devices, and large-scale distributed systems of hundreds of nodes [50]

2.1.9.2 Keras

Keras⁸ is an open-source machine learning library available for Python. It is developed and maintained by Francois Chollet and works as a wrapper around lower-level technologies, such as TensorFlow, or the no longer maintained CNTK⁹ and Theano¹⁰ [51].

In Keras, models are a sequence or a graph of stand-alone, fully configurable modules (e.g., cost functions, optimizers, activation functions) that can be plugged together with as few restrictions as possible. Keras can run models on both the CPU and the GPU[51].

2.1.9.3 SciKit-Learn

SciKit-Learn¹¹ is an open-source library available for Python. It started as a Google Summer of Code project by David Cournapeau and is under active development. SciKit-Learn uses NumPy¹² and SciPy¹³ libraries as its core and provides multiple algorithms as a higher-level tool. This technology is well maintained, frequently updated, and supports many subsets of problems [52].

Besides providing many learning algorithms has a large number of convenient functions for pre-processing data tasks (e.g., normalization) [53]. The Sci-Kit library does not support GPU computing [51].

¹<https://tensorflow.org/>

²<https://cplusplus.com/>

³<https://python.org/>

⁴<https://docs.oracle.com/en/java/>

⁵<https://go.dev/>

⁶<https://www.r-project.org/>

⁷<https://haskell.org/>

⁸<https://keras.io/>

⁹<https://docs.microsoft.com/en-us/cognitive-toolkit/>

¹⁰<https://pypi.org/project/Theano/>

¹¹<https://scikit-learn.org>

¹²<https://numpy.org/>

¹³<https://scipy.org/>

2.1.9.4 PyTorch

PyTorch¹⁴ is an open-source machine learning library available for Python, developed by Facebook's AI research group in 2016. PyTorch supports tensor computation with strong GPU acceleration and DNNs built on a tape-based autograd system. It has become famous for allowing the building of complex architectures quickly. Typically, changing the way a network behaves means starting from scratch. PyTorch uses a reverse-mode auto-differentiation technique, which allows changing the way a network behaves with small effort (i.e., dynamic computational graph or DCG) [54].

PyTorch backend is in C++, Python, and CUDA¹⁵, and, like TensorFlow, has APIs interfaces for both Python and C++. Pytorch can run models on the CPU, GPU, and mobile devices [54].

2.1.9.5 Comparing Technologies for Machine Learning

In Table 2.1 the enumerated frameworks are compared against each other.

Among the languages supported by these technologies, Python is the more prevalent; all of these technologies are capable of being scaled into production. However, not all of them support GPU and edge computations. SciKit-Learn only supports CPU computations, and Keras only supports edge devices depending on the technology it uses under the hood (e.g., when relying on Tensorflow, it can support mobile devices).

There are apparent differences in the development of these technologies: SciKit-Learn and Keras provide high-level abstraction libraries, TensorFlow is a low-level library, and PyTorch sits in the middle.

All the enumerated technologies include good official documentation and support from the community, maintaining a constantly updated repository at this writing. SciKit-Learn is the least prevalent technology in the industry.

¹⁴<https://pytorch.org/>

¹⁵<https://docs.nvidia.com/cuda/>

Table 2.1: Machine learning technologies comparison.

	TensorFlow	Keras	SciKit-Learn	Pytorch
Programming Languages Support	C++, Python	Python, R	Python	Python
Scalability	Yes	Yes	Yes	Yes
Open-Source	Yes	Yes	Yes	Yes
Edge Device Support	Yes	Yes, depending	No	Yes
Level of Technology	Low-Level	High-Level	High-Level	"Mid"-Level
Good documentation	Yes	Yes	Yes	Yes
Good Community Support	Yes	Yes	Yes	Yes
Mostly used in	Academia and Industry	Academia and Industry	Academia	Academia and Industry

2.2 Related Work

Companies and businesses worldwide are undergoing a process of digital transformation. Many businesses make their services available to customers via APIs, growing their abuse potential.

This dissertation defines API abuse as the misuse of API functions to carry out malicious activities (e.g., using the API to gain unauthorized access or sending many requests to the server's APIs, trying to drain its resources).

This dissertation is to employ an ML-based solution to detect API abuse. Due to a lack of work on this specific topic, similar topics as malware detection and network anomaly traffic detection are also considered for the literature review. The criteria for selecting these papers was their usage of ML in these related topics.

This section reviews the state-of-the-art in API abuse and the related problem of network anomaly traffic detection and malware detection with applied machine learning by systematically describing the essential publications.

2.2.1 API Abuse Detection using Machine Learning

According to [55] API abuse has two identifiable types, API abuse by volume and API abuse by sequence, further explained below.

API abuse by volume refers to volumetric anomalies. Each path or endpoint gets assigned a threshold of how often it can get accessed by a specific Internet Protocol (IP) address, user agent, or other session identifiers. If this threshold gets surpassed, then it is considered abnormal use.

API abuse by sequence refers to the sequential anomaly detection problem. Consuming the API paths or endpoints with specific data and in a particular sequence, an attacker might trigger unforeseen conditions to put the service or system in a vulnerable state.

The authors of [55] propose solutions for API abuse of these two types. In the case of API abuse by volume, they propose an unsupervised learning algorithmic solution. For API abuse by sequence is proposed the usage of Markov Chains [56] to build a transition matrix consisting of the typically chosen sequence of endpoints accessed by a user. However, there are no given results for any of these methods.

In order to be able to implement solutions to these two types of identifiable types of API abuse, [55] aims to use a process called path normalization. In this process, pre-processing the endpoints name diminishes the possible number of states or endpoints.

Described below is an example of path normalization. For the endpoint `/auth/*/enter/`, there are as many possible states as accepted by the wildcard, such as:

1. `api.example.com/auth/421/enter/`
2. `api.example.com/auth/321/enter/`

Here "421" and "321" are identifiers, and both endpoints serve the same purpose and by applying path normalization, they could be collapsed into the following equal paths:

1. `api.example.com/auth/*/enter/`
2. `api.example.com/auth/*/enter/`

The authors of [15] propose an ML-based solution employing a SVM to serve as a binary classifier, separating the regular from the abnormal API traffic.

In this case, the article considers six API traffic features: the client IP address, the Hypertext Transfer Protocol (HTTP) request type, the HTTP code, the number of consecutive requests, the bandwidth of the request, and the connection-token.

Due to an ML-based supervised solution, one of the requirements is a labeled dataset. The article considered the usage of the 1999 Knowledge Discovery and Data Mining Tools Cup (KDD'99 Cup) dataset, which includes four types of intrusions simulated in a military network environment: Denial of Service(DoS), Probing, User to Root (U2R), and Remote to Local (R2L). However, the authors concluded that this and other public datasets do not satisfy the specifications of API traffic data. Furthermore, most of the public datasets are outdated. Instead, they decided to generate their dataset as, at the time, no labeled public dataset of API traffic was readily available due to intellectual property or other privacy-related issues.

Their experiments compared the proposed solution to IDS solutions like Snort and Suricata. Their findings show that the SVM system could substantially detect previously unseen attacks better than rule-based API traffic solutions. Also, the SVM algorithm required less time for training, testing, and classification. Because of this, the authors claim that the proposed anomaly detection method can be used in a plug-and-play manner and fits into the existing micro-service architecture with few adjustments to provide accurate results in a fast and reliable way.

2.2.2 Network Traffic Anomaly Detection using Machine Learning

Intrusion detection and intrusion prevention systems are devices or software applications that monitor networks or systems for malicious activity or policy violations.

Existing IDS and IPS solutions may be adapted to classify access to APIs. These systems typically work at the packet level and are not designed to monitor abusive patterns emerging at the API traffic level.

Regarding abstraction, API abuse detection sits a step above traditional network traffic anomaly detection, a higher-level analysis in comparison. API traffic is specific with unique features and behaviors that differ from other network traffic at the packet level, and consequentially, the input features for ML solutions applied to this topic are typically different.

A network traffic anomaly refers to the traffic traces being different from those considered normal behaviors. These traces may differ in many aspects, such as misuse of network equipment, network operations anomaly, flash crowd, network intrusion, and so on [57].

The authors of [58] propose an SVM solution for the network anomaly traffic detection problem.

Because of the nature of an ML-based supervised solution, a labeled dataset is necessary. The article uses the previously discussed KDD'99 Cup dataset. It extracts six features from each request, used to construct the feature vector for the experiments: Source IP, Destination IP, Source Port, Destination Port, Packet Size, and Packet Type.

The four types of attack behavior in the KDD'99 Cup dataset formed a single category for the experiments. Thus the problem is modeled as a binary classification problem, with the two possible outputs being that the traffic is either abnormal or normal.

The experiments demonstrate the proposed system's effectiveness with up to 90% accuracy. The proposed system demonstrates superior results compared with other proposed methods through the ROC curve.

When dealing with network anomaly detection systems, the sampling rate of the network may reach very high values creating a large amount of data, thus requiring the detection model to train with amounts of data that may fail regular NNs. Because of this, the authors of [59] propose an ML-based supervised solution using an ELM approach.

Using ELMs requires a labeled dataset, such as the NSL-KDD used by the authors for their experiments. The NSL-KDD dataset is a revised version of the KDD'99 Cup dataset created by the University of New Brunswick. No feature selection or data reduction is applied.

This article considers both a binary classification problem and a multi-class classification problem. In the same way, as in the previously reviewed article, the four types of attacks present in the dataset are grouped into the abnormal category for the binary classification problem. In the multi-class classification case, the labels correspond directly to these four types of attacks.

Their experiments demonstrate that the proposed approach is effective in building models with reasonable detection rates when applied to binary and multi-class classification problems, compared to other detection methods. The results also demonstrate swift training and predicting times.

The previously reviewed articles used or proposed an Offline Learning methodology, where the system trains once using all the available data. In the ever-lasting process of detecting new anomalies, network attacks, and intrusions, using an online learning methodology makes sense when building such a system. So that the model incrementally trains to learn to detect new abnormalities as they are received.

The authors of [60] compare a series of ML-based detection methods in the network anomaly traffic problem. The compared algorithms are: the k-Nearest Neighbour (k-NN) classification algorithm [61], Hoeffding Adaptive Trees (HAT) [62], Random Forests(RF) algorithm [63], and a NN with their respective offline and online learning implementations.

As in the previously reviewed articles [60] implements ML-based supervised algorithms and, as such, requires the usage of a labeled dataset. In this case, the authors choose the MAWILab dataset, a public collection of 15-minute network traces captured daily on a backbone link between Japan and the US since 2001. The MAWILab dataset identifies five types of traffic anomalies: Distributed Denial of Service (DDoS) attacks, HTTP flashcrowds, Flooding attacks, and two different flavors of distributed network scans, one using User Datagram Protocol (UDP) and the other using Transmission Control Protocol (TCP) ACK probing.

Because the proposed solution intends to learn about new data sequentially as it arrives, a way to describe the traffic during a specific period is necessary. The traffic traces get split into consecutive timeslots of five seconds each, and a set of features describing the traffic in each slot is populated.

Their experiments consist of a multi-class classification problem, where the anomaly type can be one of the five previously described for the dataset.

Their results show that the online learning implementation of the previously described algorithms performs similarly, or in some cases, better than the offline learning implementations.

Network anomaly detection is, without doubt, one of the most challenging fields in cyber security due to its many difficulties, such as necessary constant learning and adaptation and the requirement to process a considerable amount of data at any given time. Hence, researchers are always looking for innovative solutions to solve these problems.

To tackle these challenges [64] proposed a novel two-tier classifier model for network anomaly traffic detection employing ML. The first tier consists of standard data pre-processing and classification using the Naive Bayes classification algorithm [65]. The second tier consists of another classifier using the k-NN classification algorithm [61]. The proposed system works such that the

samples classified as standard in the first tier are taken through the second tier to ensure that no anomaly goes without being flagged.

As in previous articles, a labeled dataset is necessary, and the authors chose the NSL-KDD dataset and presented and applied a dimension reduction process for the feature selection process. Their obtained feature space featured four features from the original forty-one distinguished attributes of the NSL-KDD dataset.

The problem is modeled as binary and multi-class classification problems. This modeling entails that the four types of attacks present in the dataset group into the abnormal category in the binary classification. The second multi-class classification traces map directly into these four types of attacks.

Their experiments demonstrate that the proposed solution consisting of two-level classification by Naive Bayes and the k-NN classification algorithm led to a higher detection rate on the U2R and R2L attacks but not in the Probing and DoS attacks types than the compared solutions. The binary classification problem had a higher detection than the compared solutions.

2.2.3 Malware Detection using Machine Learning

APIs are not only present in the cloud. By definition, they are sets of rules and specifications for the ways processes interact. For example, Operating Systems (OSs) offer software APIs that run on top of them, where an API can work to obtain the current time or significantly alter the execution of a process created by the OS.

Naturally, OS APIs get called by malware in their nefarious executions. This way, API calls, and their sequences are essential in the Malware Detection and API abuse problems.

In [66] the authors propose an ML-based system using an SVM classifier that classifies behavior patterns from software as benign or malicious—further illustrating the versatility of the SVM algorithm.

The authors create the labeled dataset used in the experiments by using malware obtained from the 'VirusShare Malware Sharing Platform' [67] and benignware from the system32 folder in Windows installations. The execution of these processes gets mapped into a series of features that describe its run-time behavior, including the sequence of OS API calls made by the process in the form of an n-gram. N-gram is a contiguous sequence of n items from a given sample [68].

The article considers a series of binary classification problems one for each included malware family. Allowing the run-time behavior of the process to be classified into malware or benignware.

Their experiments compared the proposed system with k-NN and RF algorithms applied to the same dataset. The results demonstrate its slight superiority over these other proposed methods, with testing accuracy reaching as high as 98%.

The previous article introduced API call sequences as a feature used to detect malware.

The article [16] elaborates further on this idea, proposing the transformation of the API call sequence graph into an image that gets classified by the proposed ML-based system using the Autoencoder algorithm adapted for classification problems.

The dataset used for the experiments in the article is a set of images representing the features characterizing its behavior, that is, the sequence of API calls mapped into images. The authors create their dataset by using: malware obtained from the Malgenome [69], Contagio Minidump [70], and the ViruShare [67] malware datasets; and benignware collected from the Playdrone Crawler [71] along with the Google Play store.

The authors consider a series of binary classification problems similar to the previous article. The run-time process behavior gets classified into malware or benignware, each problem corresponding to a malware family.

The experiments compare the proposed system with the J48 Classifier [72], Naive Bayes algorithm, and a Deep Neural Network. The results prove that this image classifier can surpass the other classifiers and obtain Accuracy values up to 94%.

2.2.4 Literature Review Summary and Comparison

In table 2.2 we summarize the different articles and draw comparisons.

All the reviewed articles used ML in their proposed solutions and pertained to three topics: API abuse detection - two articles, network anomaly traffic detection - three articles and malware detection - two articles. The most prevalent ML algorithm are SVMs, employed by three papers in total, followed by NNs, and the k-NN classification algorithm by two papers. From the reviewed papers, most problems are formulated as binary classification types, meaning that the implemented solutions will either detect abuse or not. In multi-class classification, a solution would be able to differentiate between different types of abuse.

The publication "Stream-based Machine Learning for Network Security and Anomaly Detection" [60] is the only paper that uses online learning. All other articles implement the chosen algorithms with the offline learning methodology. Observation of the table illustrates that different problems require different datasets. Only on the topic of Network anomaly traffic detections are public datasets used. All articles on other topics use generated datasets that satisfy the specifications of the respective topic. Feature sets are varied and chosen by the different authors either via a feature selection process [64], or through human judgment and trial and error.

Table 2.2: Related work summary.

Publication	Algorithms proposed or used	Dataset	Feature Set
[55] - API abuse	Unsupervised ML approach, Markov chains	N/A	N/A
[15] - API abuse	SVM	Generated	(Client IP address, HTTP request type, HTTP code, number of consecutive requests, bandwidth of the request, connection-token)
[58] - Network anomaly traffic detection	SVM	KDD-CUP 99	(Source IP, Destination IP, Source Port, Destination Port, Packet Size and Packet Type)
[60] - Network traffic detection	k-NN, HAT, RF, NN	MAWILab	Set of features describing the traffic in the timeslot.
[64] - Network anomaly traffic detection	Naive Bayes, k-NN	NSL-KDD	4 total features obtained via a feature selection process.
[66] - Malware detection	SVM	Generated	Set of features describing a process run-time behaviour, including sequence of API calls.
[16] - Malware detection	Autoencoder	Generated	Generated Image set with sequence of API calls information.

Chapter 3

Problem and Proposed Solution

This chapter describes the problem for the reader, which consists of developing an API abuse detection system. Afterward, a requirement analysis of the system is detailed. Furthermore, the proposed solution describes the proposed system architecture, the service different components concerns, modules responsibilities, the feature spaces of the tabular and image data formats, and the chosen development technologies.

It is divided into three sections: the Problem section, the Requirement Analysis section, and the Proposed Solution section.

The Problem section explains the two types of API abuse first identified in the literature review section. It then presents the problem statement as an API abuse detection system. The Requirement Analysis section formulates the different requirement artifacts for such a system. Lastly, the Proposed Solution section presents the proposed solution for the API abuse detection system consisting of its architecture, components concerns, and modules responsibilities, the system's flows, the feature spaces for the tabular and image data formats, and the adopted technologies.

3.1 Problem

As far as this dissertation is concerned, API abuse focuses on the misuse of API functions to carry out nefarious activities against the service or the underlying systems. Examples include, but are not limited to, using a web application API to gain unauthorized administrator access or sending many requests to the micro service API to drain its system resources.

As seen in the literature review section 2.2, in terms of abstraction, the API abuse problem is a higher-level problem than the network traffic anomaly detection problem, mainly done at the packet level. In contrast, the API traffic encapsulates a period with many API requests, causing some API abusive patterns to become more evident in it than in the typically shorter period, network traffic analysis, consisting of traffic packet traces.

3.1.1 API Abuse

According to the authors of [55], API abuse is separable into two types, first introduced in the literature review section: API abuse by volume (volumetric anomaly) and API abuse by sequence (sequential anomaly).

API abuse by volume refers to volumetric anomalies in API traffic; in its simplest form, the amount of calls to a specific API endpoint exceeds a threshold. This threshold represents the maximum amount of hits that any entity can make during a period of time to the API endpoint.

Figure 3.1 illustrates an example of this type of abuse. Three endpoints are listed, exemplifying the login flow in any software application, represented by a circle in the Figure:

- /auth/*/credentials
- /auth/*/verify
- /auth/login-successful

In each circle, there is a textbox representing the volumetric limit of each endpoint for a user entity. In this case, 80 accesses are the threshold for the left and center circles representing the /auth/*/credentials and /auth/*/verify endpoints. The user entity somehow accesses the /auth/*/credentials and /auth/*/verify 92 times, surpassing the threshold and creating an example of API abuse by volume, represented by a blue circle.

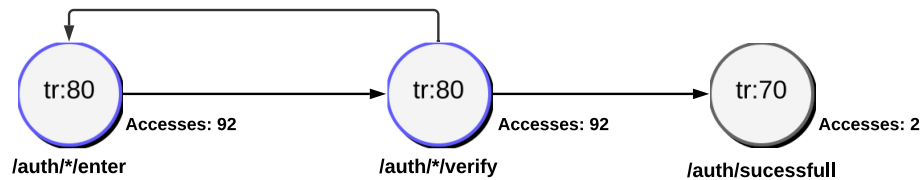


Figure 3.1: Example of API abuse by volume.

API abuse by sequence refers to sequential anomalies in API traffic, that is, the occurrence of an illegal flow of API calls, meaning vulnerability exploitation or probing.

Continuing the previous example, the normal login flow for the software application might consist of the following steps:

- A request is sent to /auth/*/credentials
- It is redirected to /auth/*/verify
- It is finally redirected to /auth/login-successful

In this example, the software application considers any other login flow illegal, constituting a broken user authentication vulnerability.

Figure 3.2 is an example of API abuse by sequence. Black sequence arrows represent the normal login flow. A blue sequence arrow represents the API abuse by sequence flow. Due to a

software vulnerability, the user entity bypasses the `/auth/*/verify/` endpoint before being redirected to the `/auth/login-successful` endpoint, thus creating an API abuse by sequence example.

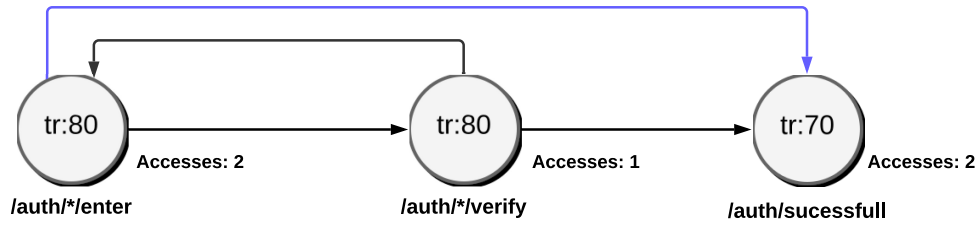


Figure 3.2: Example of API abuse by sequence.

3.1.2 Problem Statement

The problem statement has its basis on the dissertation goals defined in the introductory chapter and consists of the development of an ML-based system comprised of two models, one consuming tabular data and another consuming image data. The system should be capable of evaluating the hypothesis that ML algorithms are capable of devoting themselves to the API abuse field by capturing statistical metrics against a hold-out validation split, containing API abuse by volume and API abuse by sequence attacks examples.

The proposed and developed system should also be able to integrate with existing microservice architecture without interfering with its normal functioning. The API traffic analysis must be time-efficient to allow the system administrator to intervene, preventing continuous API abuse.

3.2 Requirements Analysis

This section presents the API abuse detection system requirement analysis.

It first introduces the domain concepts, transcribes them into a domain model example, then maps the dissertation objectives into functional and non-functional requirements for the proposed system, based on the FURPS+ model [73].

3.2.1 Domain Concepts

An API abuse detection system deals with different actors, entities, and interactions throughout the different steps to provide API traffic classifications.

This dissertation considers two main actors:

- The pre-existing microservice architecture.
- The API abuse detection system.

User entities create and send API requests to the microservice(s), which are processed, recorded, and aggregated—this is a pre-condition of the pre-existing microservice architecture: recording and aggregating API requests for the different microservices.

API traffic data is a set of API requests belonging to a singular user entity for a period. Furthermore, the pre-existing microservice architecture has to facilitate the collection of API traffic data by the proposed system; this data is ingested and pre-processed into tabular and image formats. The system ML-models consume the tabular and image data for training and prediction purposes; when predicting, the service creates API traffic classifications identifying the user entity.

Figure 3.3 presents a possible representation of the domain model of an API abuse detection system, with the primary domain concepts and their interactions.

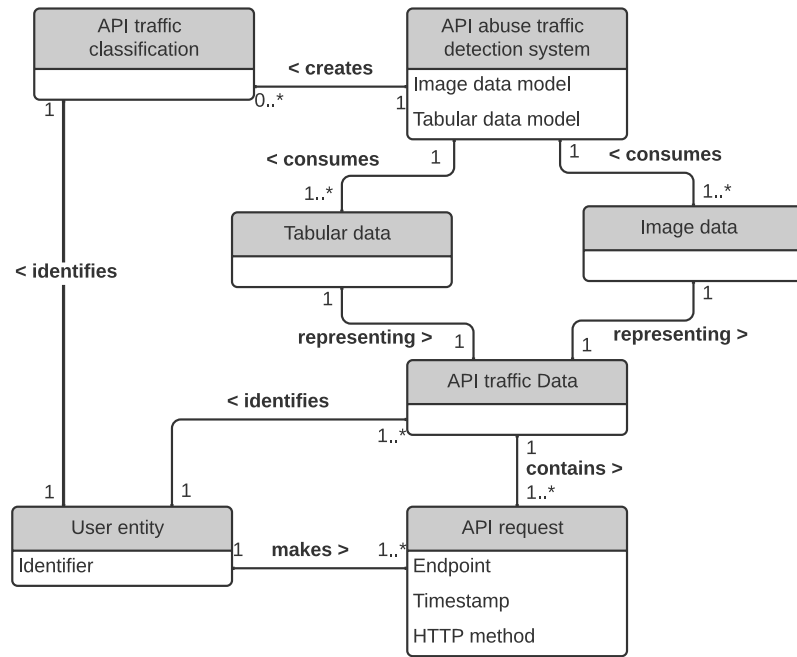


Figure 3.3: Domain model for an API traffic abuse detection system.

3.2.2 Functional Requirements

Functional requirements describe the system's main characteristics and core functionalities.

Use Cases (UC) represent the functional requirements, they are numbered, and the first letter identifies the actor (e.g. P for Project Owner).

Pre-existing microservice architecture Use Cases

- **UC1_S:** Facilitate API traffic data collection by the API abuse detection system.
- **UC2_S:** Receive and process API traffic classifications.

Pre-conditions For the proposed system deployment, the pre-existing infrastructure has the pre-condition of recording and aggregating the microservice(s) received API requests into API traffic data.

API abuse detection service Use Cases

- **UC3_A:** Ingest and pre-process API traffic log data into tabular and image format.
- **UC4_A:** Train and evaluate the models.
- **UC5_A:** Update the models.
- **UC6_A:** Perform API traffic analysis.
- **UC7_A:** Perform periodic API traffic analysis.

Use Cases UC3_A, UC4_A and UC6_A are the main focus of this dissertation; they allow the proposed system to test the dissertation's central hypothesis that ML algorithms lend themselves to the API abuse problem. UC_5A and UC_7A are more complex Use Cases that encompass the other UC; while they can add value to the solution, they are not the main target of this dissertation. The infrastructure in which the system is deployed has the pre-condition of recording and aggregating API requests relative to the different microservices, and its UCs fulfillment falls outside this dissertation's scope; UC1_S fulfillment is, however, required for the system to obtain training and predict data.

3.2.3 Non-Functional Requirements

Non-functional requirements specify criteria used to judge the operation of a system, and follow the FURPS+ specification. In addition to the original non-functional requirements (usability, reliability, performance, and supportability) listed in the original FURPS model [74], the successor model FURPS+ brings the specification of design, implementation, interface, and physical constraints.

Usability Usability requirements concerns are human factors, aesthetics, consistency, and documentation [74]. For the system, they are:

- The system should treat errors and unexpected situations smoothly, and when possible provide details regarding the cause.
- The system interface should be intuitive and easy to use.

Reliability Some of the reliability requirements concerns are frequency/severity of failure, accuracy, and mean time to failure [74]. The reliability requirements for this project include:

- The system should keep itself available after a previous error.
- The system should be able to handle missing data for a API traffic set.
- The system should be able to handle errors and not propagate them needlessly.

Performance Performance requirements concerns are speed, efficiency, resource consumption, throughput, and response time [74]. For the system, they are:

- The system should process and analyze API traffic data in a reasonable time, in the sense that it should give the administrator time to stop continuous abuse behavior.
- Key Performance Indicators (KPI) about a model performance should be measured.

Supportability Supportability requirement's main list of concerns [74] includes: The system's extensibility, compatibility, and installability. For the system, they are:

- The system should be able to integrate with existing microservice architectures.

Design Constraints Design constraints limit the way a system is designed upfront [73]. Their formulation was the author's and the proponent organization's responsibility. They are, however, supported by the literature review. For the system, they are:

- The system detection engine should rely on ML solutions using neural networks, for both the tabular data format, and the image data format.

3.3 Proposed Solution

This section discusses the proposed solution for the API abuse detection system and its Machine Learning approach. It contains three subsections. The first presents the proposed system architecture and its components, concerns, and module. The second discusses the flows that originate from the functional requirements in the previous section. The final subsection explains the Machine learning approach to the API abuse problem, including the proposed algorithms for the system and their feature spaces.

3.3.1 Proposed System Architecture

The proposed system architecture is represented by a Unified Modeling Language (UML) deployment diagram in Figure 3.4. It was designed with the proponents' organization's pre-existing infrastructure in mind, and it identifies three central nodes: the microservices server, where the API gateway and the microservice components reside, the logging server, housing the logging service and the API abuse detection server, containing the API abuse detection system.

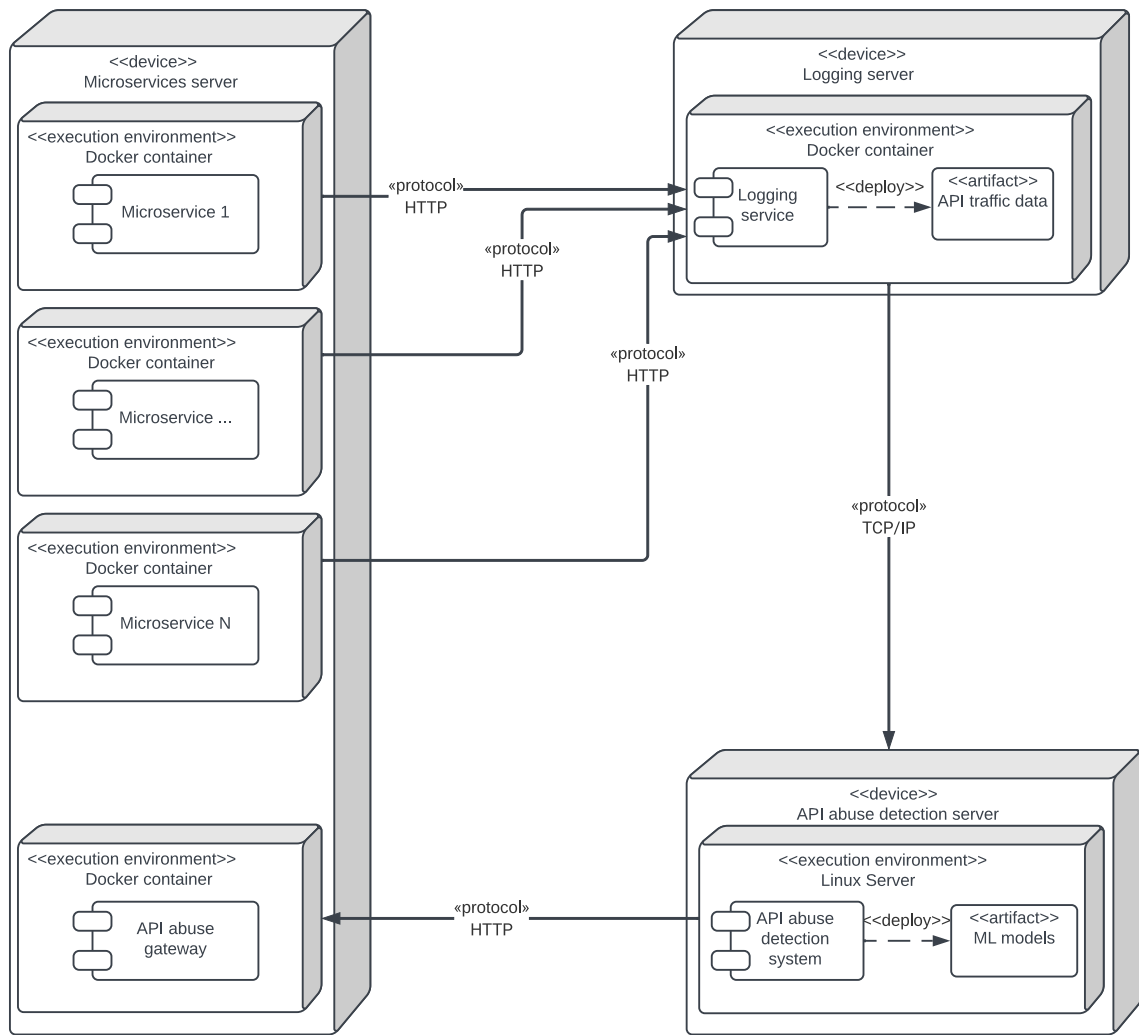


Figure 3.4: Deployment diagram of the proposed system execution architecture

In the system architecture, the different microservices components provide APIs that receive and handle API requests from various user entities. This request data is forwarded to the logging server.

The logging server then stores and aggregates the forwarded API requests creating API traffic data. It provides functionality that allows it to serve the historically recorded requests to the proposed API abuse detection system for training and predicting purposes.

The API abuse system resides in its server and integrates with the rest of the architecture in two sequential and repetitive phases: the training phase and the serving phase, both requiring the acquisition of API traffic data from the logging service; the system transforms the acquired log data into tabular and image formats. The training phase in the proposed system uses an offline learning methodology, detailed in sub-section 2.1.5, where the ML models are created and trained using their respective data formats. In the serving phase, the already trained ML models analyze API traffic and report the abnormalities to the API gateway.

The API abuse detection system separation of concerns appears in a component UML in Figure 3.5, separating the system into two main components: the data component and the analysis component. The description of the two components' concerns and modules follows.

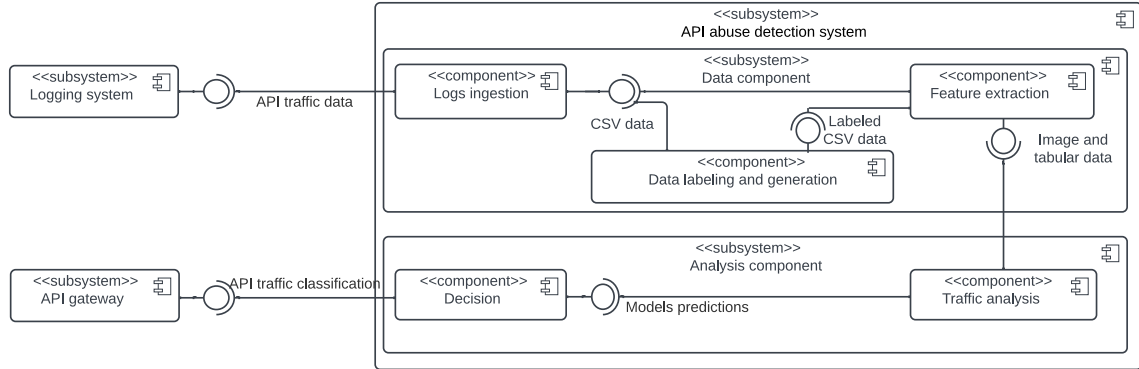


Figure 3.5: Component diagram of the proposed system.

3.3.1.1 Data component concerns and modules

The data component obtains API traffic data from the logging system for training and predicting purposes and is responsible for ingesting and transforming the data into the tabular and image formats fit for the ML-models use. It comprises three modules: logs ingestion, dataset labeling and generation, and feature extraction modules. These modules heavily depend on the business logic and the logging system implementation.

Logs ingestion module The logs ingestion module is responsible for transforming the logs from any custom data format into a Comma Separated Value (CSV) format and applying Path Normalization.

Data labeling and generation module In the proposed system architecture, the system itself is the only source of truth; however, labeled data is necessary for the training process of the ML-based solution. The data labeling and generation module's responsibility is to label the data and generate malicious traffic with human supervision.

Feature extraction module The feature extraction module is responsible for feature extraction from the ingested logs, creating a feature space that the models can consume.

3.3.1.2 Analysis component concerns and modules

The analysis component is responsible for the ML-based solution and its processes, such as model creation, training, evaluation, and predictions; furthermore, it communicates with the API gateway to provide API traffic classifications. It comprises two modules: the traffic analysis and the

decision modules. These modules are remarkably free from business logic and logging service implementation.

Traffic analysis module The traffic analysis module is responsible for the ML-based solution and its processes, including data loading and ingestion, model creation training and evaluation, and model predictions.

Decision module The decision module is responsible for combining the predictions from the models for the API traffic via a decision process to create API traffic classifications and forward them to the API gateway.

3.3.2 Proposed System Flows

The proposed system has five UCs, represented by five flows: the ingestion flow, the training flow, the update flow, the prediction flow, and the analysis flow, representing UC3_A, UC4_A, UC5_A, UC6_A, and UC_7A, respectively.

The proposed system flows and the logging flow originating from the pre-existing microservice architecture pre-condition are described below via UML sequence diagrams.

Logging flow The logging flow represents the typical consumption of the microservice(s) API(s) by the user entities and the respective storage of the API requests in the logging system.

Figure 3.6 illustrates the logging flow; the Figure identifies three actors, the user entity, the microservice, and the logging system. The user entity sends an API request to the microservice, which forwards it to the logging system before processing and sending the API response to the user entity.

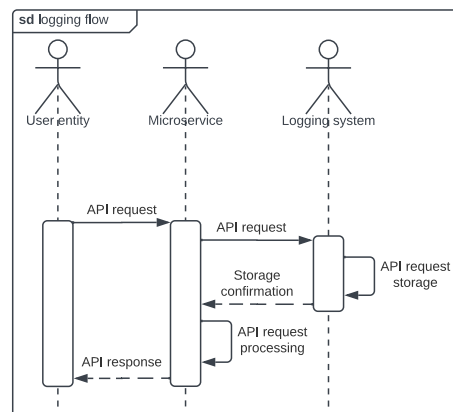


Figure 3.6: Sequence diagram of the pre-condition logging flow.

Ingestion flow The ingestion flow represents UC3_A and is represented in Figure 3.7; in the Figure, the logging system appears as an actor and the system is illustrated as a package and

contains the three modules that participate in this flow, represented by lifelines, in addition to the ingestion controller.

In this flow, the controller retrieves the API traffic data from the logging system, and passes it to the log ingestion module, where it is transformed into CSV data. At this point, the resulting CSV data may be taken through the data labeling and generation module if the system directive is training, producing labeled CSV data, or directly passed to the feature extraction module, where through a series of feature extraction processes, it is transformed into the tabular and image formats, with the respective feature space.

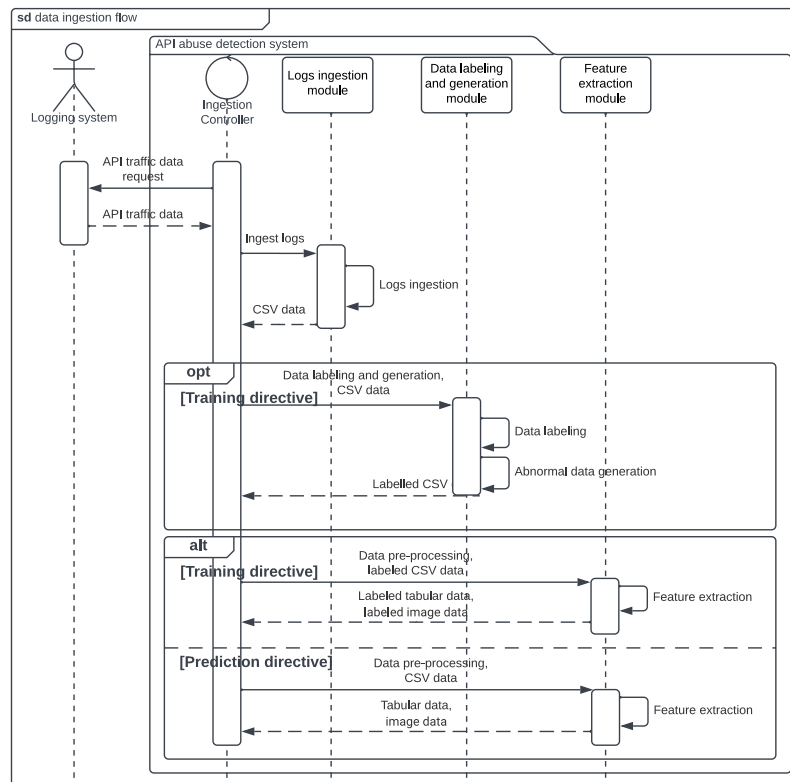


Figure 3.7: Sequence diagram of the ingestion flow.

Training flow The training flow representative of UC4_A is shown in Figure 3.8. The system is shown as a package and includes the traffic analysis module, represented by a lifeline, and the training controller. Also, the model creation process appears represented by two separate create messages, creating two additional lifelines belonging to the image and the tabular model instances.

In this flow, the labeled data originating from the ingestion flow is passed to the traffic analysis module, along with a training directive that specifies the ML model. When the traffic analysis module receives this directive, it creates the pre-determined model and performs a training evaluation split of the training data. The model then trains using the training split. When training finishes, an evaluation command is issued, passing the evaluation split, used for model evaluation. The model is serialized for later use when it completes the evaluation process.

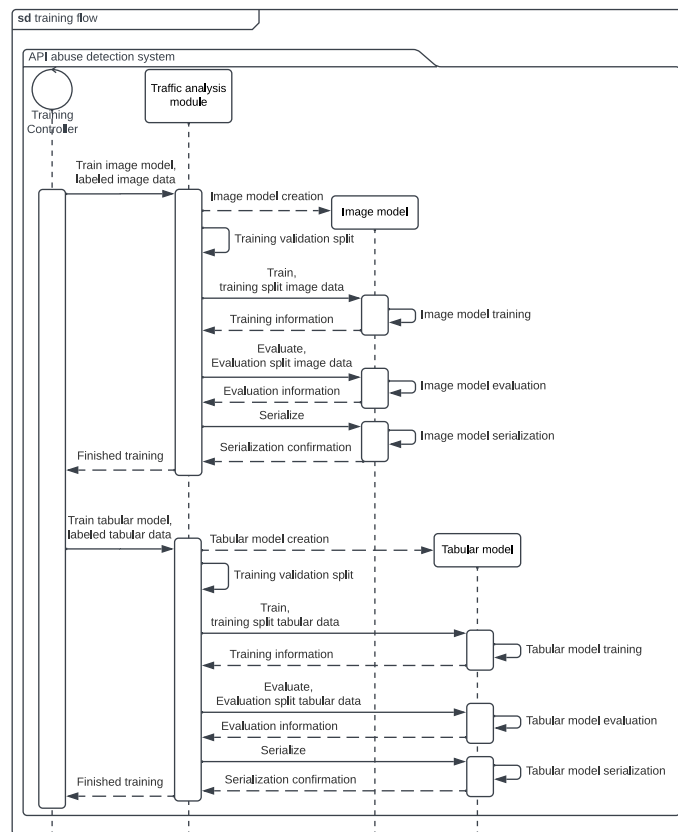


Figure 3.8: Sequence diagram of the training flow.

Update flow The update flow pertains to UC5_A and is in the form of a sequence UML diagram in Figure 3.9. The logging system is shown as an actor and the system as a package. The system includes the four active modules in this flow, represented by lifelines and the update controller. Create messages indicate the model creation process, which separately instantiates the models as lifelines.

As inferred from the Figure, the UC5_A encompasses UC3_A and UC4_A, joining them to represent the training phase in a production setting, where the update controller obtains API traffic data from the logging system via a scheduled task. The data is then ingested and transformed to CSV format by the logs ingestion module and the resulting data passes through the data labeling and generation module, creating labeled data. The labeled data undertakes the feature extraction process of the feature extraction module responsibility, creating the proposed feature spaces for the image and tabular formats. The traffic analysis module creates and performs a training validation split on the received labeled data, the image, and tabular models, which undergo training with the training split, evaluation with the evaluation split, and serialization processes.

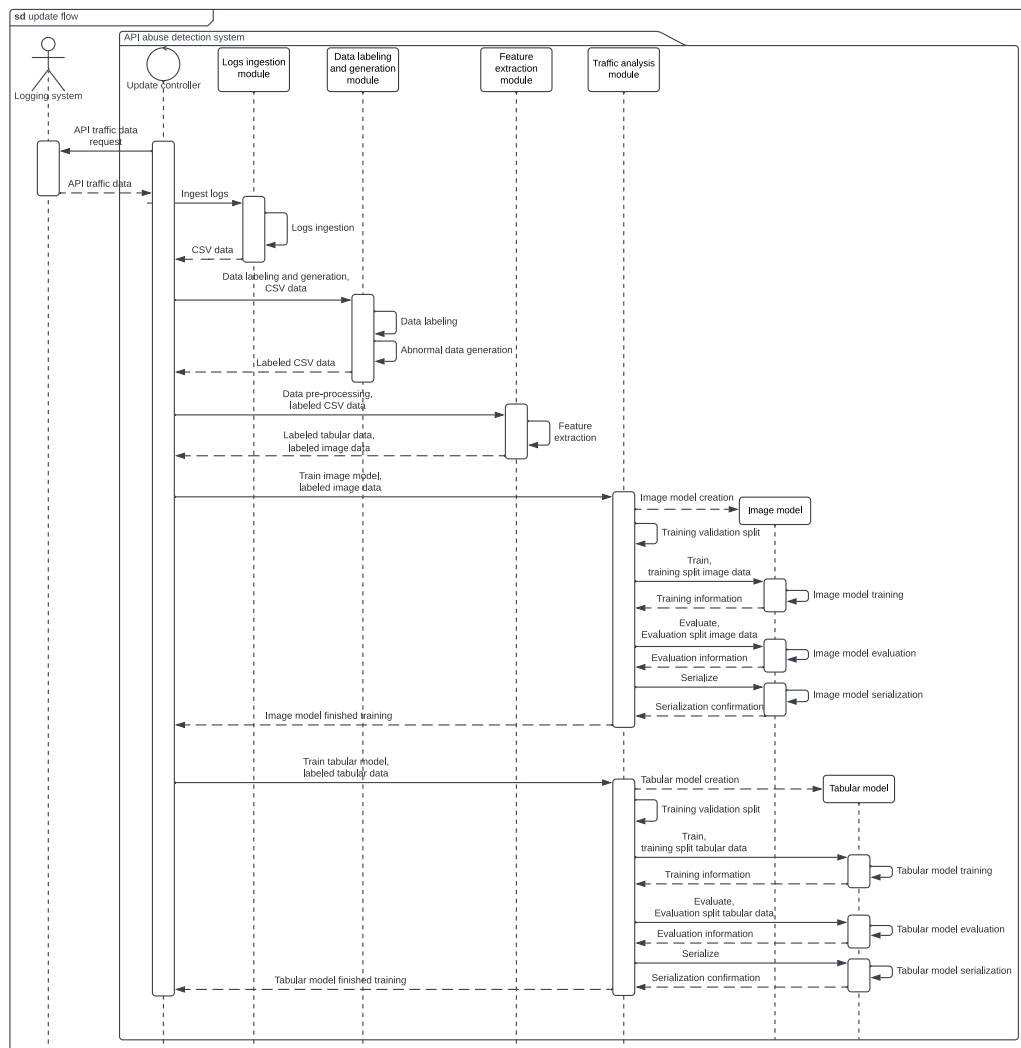


Figure 3.9: Sequence diagram of the update flow.

Prediction flow The prediction flow represents UC6_A, depicted by a sequence UML diagram in Figure 3.10. The system is shown as a package containing the module present in this flow as a lifeline. Creation messages indicate a model instantiation, creating a model lifeline.

In this flow, the predicting data originating from the ingestion flow is sent to the traffic analysis module via a prediction system directive, which contains the pre-determined model corresponding to the specified data format. The traffic analysis model then deserializes the previously trained models and uses them for prediction purposes, creating image model predictions and tabular model predictions.

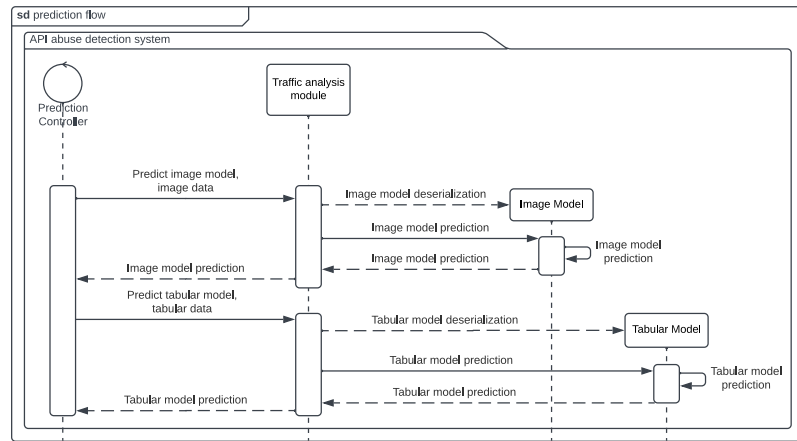


Figure 3.10: Sequence diagram of the prediction flow.

Analysis flow The analysis flow pertains to UC7_A, represented by a sequence UML diagram in Figure 3.9. The logging system and the API gateway are shown as actors and the system as a package. The system encloses the four active modules in this flow, represented by lifelines and the analysis controller. Create messages indicate the model creation process, which jointly instantiates the models as a lifeline.

This flow encompasses UC3_A and UC6_A, combining them to represent the serving phase in a production setting. In the analysis flow, via a scheduled task, the controller obtains API traffic data from the logging system for prediction purposes. This data undertakes the ingestion process in the logs ingestion module and the feature extraction process in the feature extraction module, creating tabular and image data representations with the proposed feature space. The traffic analysis module is sent a directive to deserialize the previously trained models, and a prediction request containing the tabular and image representations is sent to the instantiated ML models, which create two sets of predictions. These predictions are then processed in the decision module and sent to the API gateway, resulting in API traffic classifications.

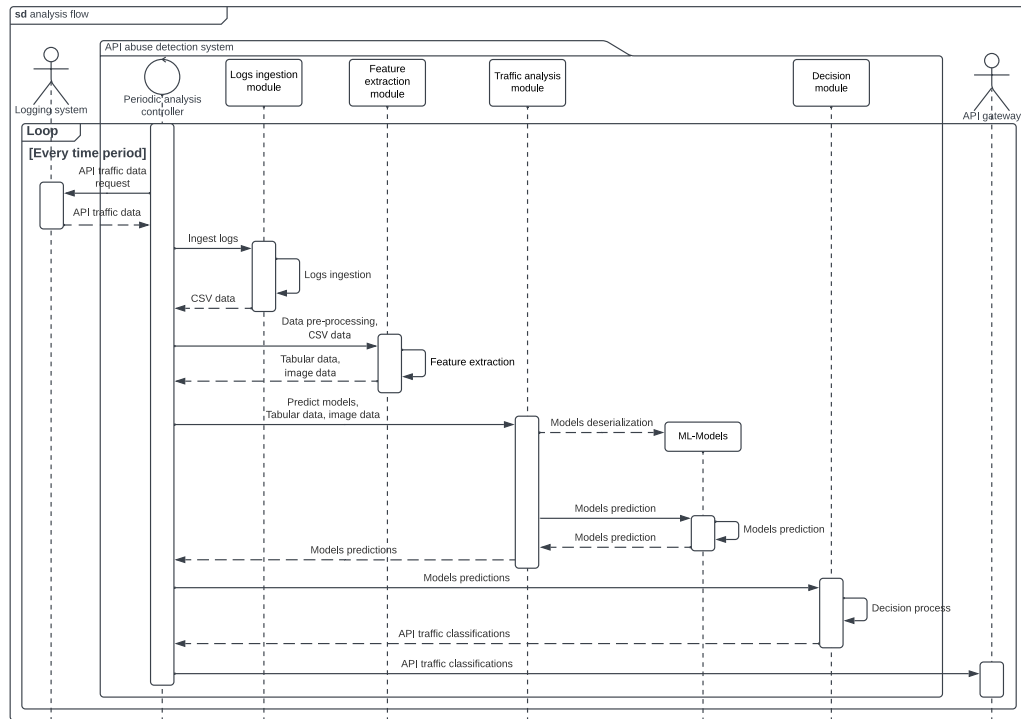


Figure 3.11: Sequence diagram of the analysis flow.

3.3.3 Proposed Formats Feature Spaces

As seen in subsection 2.1.1 supervised learning requires a labeled training dataset. A labeled dataset comprises pairs of a target variable and a set of features. The target variable represents the dependent variable that the algorithm tries to understand. The features represent the independent variables that explain "what we see in the data".

In tabular data, features can be numerical or categorical. Numerical features correspond to quantitative information, represented as numbers; categorical features, on the other hand, correspond to qualitative information, represented by groups or categories [75].

These features can be extracted directly from the data or created by combining different elements from the original dataset by applying mathematical operations. The mathematical operations that generate new features include averages, counts, and maximum and minimum values. This process benefits, typically, from knowledge of the domain. The entire input set of features is called "feature space" [75].

This dissertation proposes using two neural networks for API traffic analysis, a MLP and a CNN, requiring tabular and image data formats respectively; their output combines to determine the API traffic classification, detecting API abuse by sequence and volume.

Below are descriptions of the feature space proposed for both formats.

3.3.3.1 Tabular Feature Space

The MLP consumes data in the tabular format, where each row corresponds to a period of time for a user entity; there are as many rows for a period as possible sequence n-grams. The feature space dimensions depend on the business logic; the proposed features contain:

- A numerical feature representing the number of previous periods when the user entity consumed the microservice API - **nr_total_previous_time_periods**;
- A numerical feature representing the total number of the made requests by the user entity during the analyzed period - **nr_total_api_requests**;
- A categorical feature representing the sequence of requests during the analyzed period in the form of an n-gram - **api_requests_sequence**;
- A numerical feature representing the total number of made requests by the user entity during the analyzed period for each possible API request - **A_incidence**, ..., **Z_incidence**;

Table 3.1 is a example representation of the tabular feature space; it contains one row of data.

Table 3.1: Tabular feature space example representation.

nr_total_previous_time_periods	nr_total_api_requests	api_requests_sequence
2	10	"A-B-C-A-B-C"
A_incidence	B_incidence	C_incidence
3	3	3
D_incidence	E_incidence	
1	0	

3.3.3.2 Image Feature Space

The CNN consumes data in an image format, where each image corresponds to a period of time for a user entity. So there exists only one image for each period.

Each image contains two planes. The primary plane sits at the image's center, while the secondary plane is positioned at the top.

The primary plane represents the API requests made by the user entity during the analyzed period and their sequence. The Y-axis scales with each possible API request; each tick represents a different API request. The X-axis scales for the size of the time analyzed period and a red dot represents each request.

The secondary plane represents the past accesses made by the user entity during a sliding time window. The X-axis scales according to the time window size. Every previous period when the user entity consumed the microservice API is represented on the plane by a red dot at Y=1.

Figure 3.12 is a example representation of the image feature space.

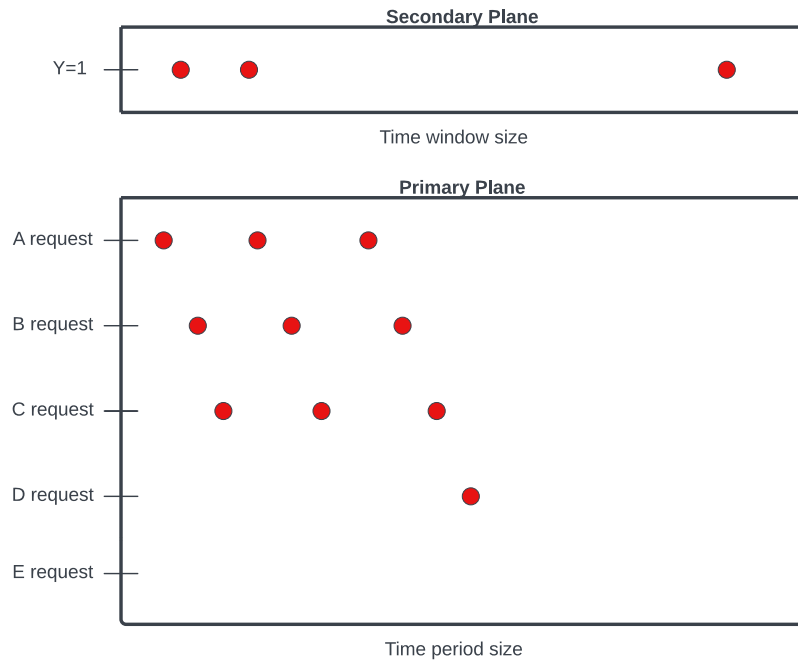


Figure 3.12: Image feature space example representation.

3.3.4 Adopted Technologies

The state-of-the-art chapter previously presented a review of the candidate technologies (see section 2.1.9) for developing the machine learning models: TensorFlow, Keras, SciKit-Learn, and Pytorch. The selected technology was Tensorflow and, by association, Keras due to TensorFlow's reliance on Keras for higher-level functionality. The reasons motivating this decision were the big development community behind the technologies and the ample amount of machine learning algorithms supported by it; the proponent company endorsed this decision.

The different components and modules of the system vary in the chosen programming language; Python was chosen for the analysis component implementation, which is responsible for the TensorFlow models, because of its data-manipulation capabilities, simplicity, and flexibility. The data component does not have a pre-determined chosen programming language, but its implementation should revolve around BASH¹ and Python. The data operations are performed using the pandas² and NumPy frameworks, as they provide access to various functions over large datasets. The Matplotlib³ package is used to generate the images in the proposed feature space.

The software tests were developed using pytest⁴, a simple testing framework for Python; the reasons for this choice include its simple syntax and ease of use.

¹<https://gnu.org/software/bash/>

²<https://pandas.pydata.org/>

³<https://matplotlib.org/>

⁴<https://pytest.org/>

Chapter 4

Implemented Solution

This chapter describes the implemented solution's various components and modules developed in a business setting. Afterward, as the machine learning models faced various dataset-related challenges, several processes were employed to handle these challenges, and the decisions behind these processes' implementation are also detailed. The prototype system can integrate with the pre-existing microservice infrastructure in two deployment phases, additionally described: the training phase and the serving phase, providing periodic API traffic analysis and responding to new abuse patterns by updating the models. The created software tests are also presented; they are used to ensure the system's correctness.

The adopted architecture approach for the solution allows the integration of different microservices. The machine learning models framework, Tensorflow¹, and the supporting data frameworks, pandas² and NumPy³ allow the solution to support various machine learning algorithms and adapt to different data conditions if the business conditions lead to such a decision.

This chapter comprises four sections: the architectural components section, the dataset-related processes section, the system phases section, and the software tests section. The first section presents the developed solution's various components and modules implementation in detail. The second section discusses the decisions behind the dataset-related processes that the different machine learning models require. The third section reviews the two system phases in a production environment, the training and serving phase, responsible for systematic API traffic analysis and updating the models to respond to new abusive patterns. The last section presents the different software tests developed that serve as quality assurance for the system's functioning.

¹<https://www.tensorflow.org/>

²<https://pandas.pydata.org/>

³<https://numpy.org/>

4.1 Architectural Components

Concerning the system integration with the pre-existing microservice architecture present in the business setting of this dissertation (see section 3.3), the architecture included three types of systems: the microservice(s), the logging system, and the API gateway; the logging solution already received and stored API traffic data from the different microservices.

The system requirements analysis section 3.2 artifacts laid the groundwork for the proposal of the API abuse detection system, in the proposed solution section 3.3, that separated concerns in the data and analysis components, prototyped the system flows that represent the systems Use Cases, and presented the proposed feature spaces for the tabular and image data formats. In turn, this section presents a detailed view of the implementation of the different components, modules, and processes that comprise the system.

4.1.1 Data Component

The systems data component concern is the transformation of logging data into formats that the models consume with the proposed feature space, according to section 3.3.3, and, if required, the labeling or generation of this data.

It was developed using BASH⁴, PHP⁵, and Python⁶ and comprises three different modules, each responsible for one of the three main processes: the logs ingestion, the data labeling and generation, and the feature extraction processes. These modules perform according to the specifications in the proposed solution; they require slightly different implementations depending on the business logic; described below.

4.1.1.1 Logs ingestion module

The logs ingestion module is a collection of BASH, PHP, and Python scripts, working together to ingest the logs data from the business-specific custom format to a Comma Separated Value (CSV) format.

The logging system provides access to the logs of the microservice that is the subject of this work; for a determined time window, these logs describe the API calls made by a user entity to the microservice. In this business logic, the API call data includes the timestamp, the HTTP method, the endpoint, the bearer token, and the request Unique Internal Identifier (UID).

Provided below on table 4.1 is a simplified example of the original data format of the logs before the ingestion process, where the original data columns are separated by two "|" characters.

⁴<https://gnu.org/software/bash/>

⁵<https://php.net/>

⁶<https://python.org/>

Table 4.1: Original log data format.

Timestamp	HTTP method	API endpoint	Bearer token	Request uid
2021-10-21 07:51:22	POST	http://api.com/endpoint	dXNlci1lb nRpdHktaWR lbnRpZmlhcjoz	56a3c23

A BASH script is responsible for obtaining this data from the logging server and transforming it into a CSV format in several steps. The first step is transforming "|" delimiter characters into ";" characters. The second step is correcting incorrect timestamps, dropping the request unique internal identifier, and introducing "\$" delimiters around the endpoint string. The final step prepares the bearer token for decryption by removing part of the string.

A PHP script is then responsible for transforming the bearer token into a unique user identifier that identifies the user entity in the existing infrastructure.

At this point the system has a intermediate CSV format, a row corresponds to an API request and identifies the user that created it. A simplified representation of the data at this stage is in table 4.2.

Table 4.2: Intermediate CSV data format.

Timestamp	HTTP method	API endpoint	User entity identifier
2021-10-21 07:51:22	POST	http://api.com/endpoint	3

A BASH script responsible for applying path normalization finalizes the ingestion process, collapsing the endpoints into less possible states. Several unsupervised learning algorithmic solutions were proposed and experimented with for this process. However, they were not able to satisfy the requirements of path normalization. The proponents, the author, and the supervisors agreed that the implemented solution would require knowledge of business logic.

The final CSV product is shown in table 4.3 via a simplified representation.

Table 4.3: Final CSV data format.

Timestamp	HTTP method	API endpoint (collapsed)	User entity identifier
2021-10-21 07:51:22	POST	/endpoint	3

4.1.1.2 Dataset labeling and generation module

This module comprises three scripts developed in BASH and Python. Its responsibility is to label and generate API traffic sets.

Three possible labels exist - the "Normal traffic" label, the "API abuse by volume traffic" label, and the "API abuse by sequence traffic" label. The "Normal traffic" label characterizes API traffic sets, where the user they pertain to is cross-matched against a list of known "well" behaved user identifiers. These user identifiers had been registered with the microservice for more than 24 months and had more than 24 purchases. This criterion was selected alongside the proponent organization to maximize the trustworthiness of the "Normal traffic" label since there is no clear way of generating a list of non-adversary user entities.

According to the business logic, the "API abuse by volume traffic" and the "API abuse by sequence traffic" labels are attributed to API traffic sets representing the two types of abuse. This process implementation required abnormal log creation and data synthesizing to create a balanced dataset. First, the author used the microservice API in abusive by volume and sequence manners for a time period each, creating one general example of the two API abuse types. Second, specialized data synthesizing Python scripts, generated sufficient examples with data variability of abnormal use to ensure a balanced dataset from the original API abuse examples. The data generation process entails a semi-random sampling of the API traffic logs while maintaining and slightly mutating the abusive patterns. The new abusive patterns examples are then attributed to a range of 3000 user entity identifiers using a beta distribution ($\alpha = 5, \beta = 2$); carefully distributing API traffic sets throughout the data time window for a particular user, meaning that different users will have different usage patterns. Figure 4.1 is a picture of the used beta distribution probability density and cumulative distribution functions.

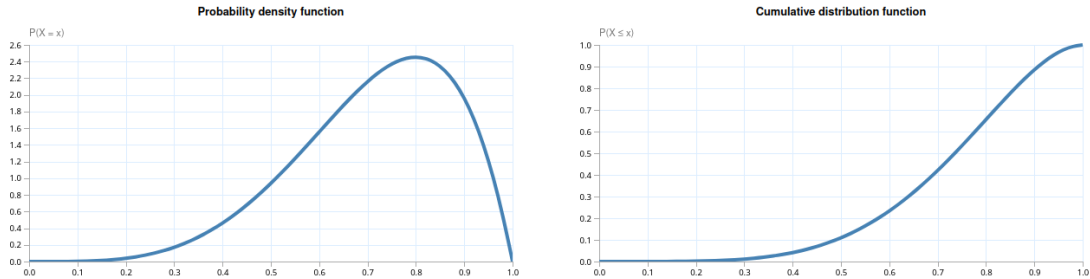


Figure 4.1: Beta distribution probability density and cumulative distribution functions.

The API abuse by volume original pattern was created by repetitively consuming a specific microservice API endpoint. The image representation of this usage pattern in the proposed feature space is in Figure 4.2; the image creation is the feature extraction module responsibility, described in the following sub-subsection.

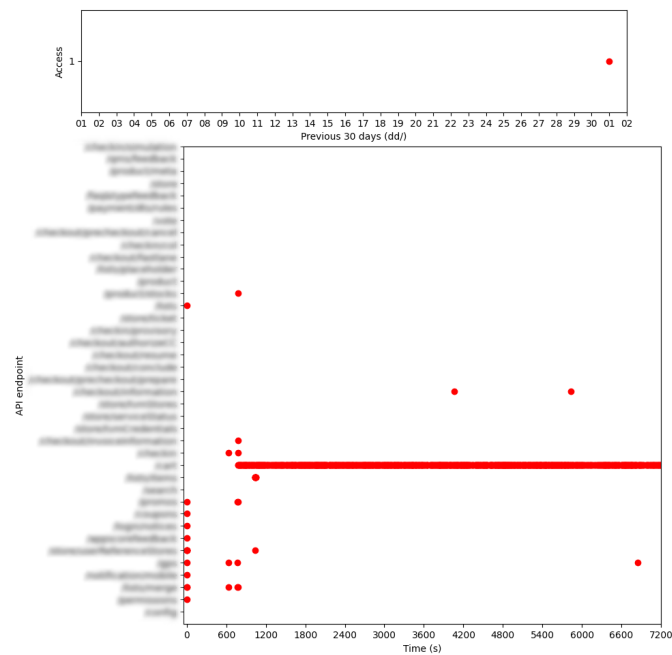


Figure 4.2: Image representation of the original API abuse by volume pattern.

The API abuse by sequence original pattern was created by consuming the microservice API endpoints in a pre-determined malicious sequence. An image representation of this usage pattern in the respectively proposed feature space is in Figure 4.3.

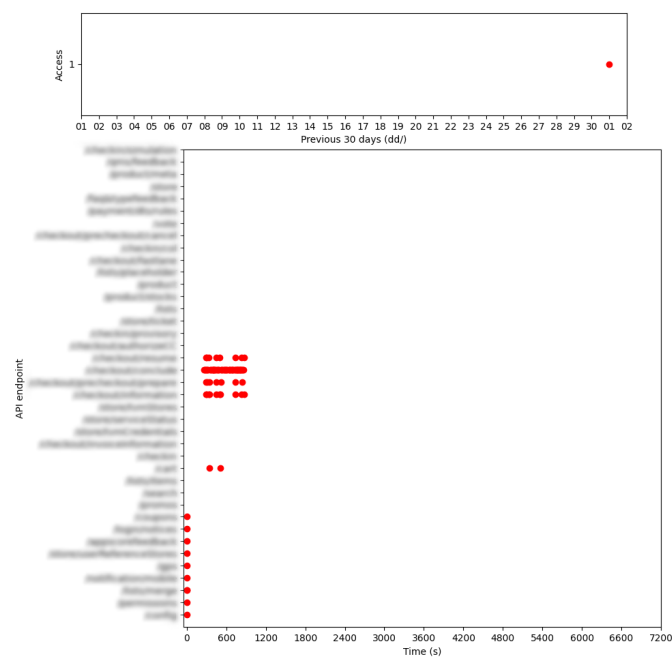


Figure 4.3: Image representation of the original API abuse by sequence pattern.

4.1.1.3 Feature extraction module

This module was implemented in Python and made use of the pandas⁷ and Matplotlib⁸ packages. It is responsible for transforming the CSV log data into representations that the models can consume, through feature extraction, according to the specifications laid out in 3.3.3.2 and 3.3.3.1.

Figure 4.4 showcases a simplified UML class diagram for the module implementation, showing the most important classes and connections.

The module entry point is the Main class for command-line management access. The command-line access allows for the generation of datasets in the tabular and image format. The Main class receives an argument parsed and communicated to the CommandLineInterfaceAPI, which functions as an API layer. It then uses the controller class DatasetController to perform the necessary data-related operations.

The controller class uses other classes to perform operations on the data or access configurations and deals with exceptions.

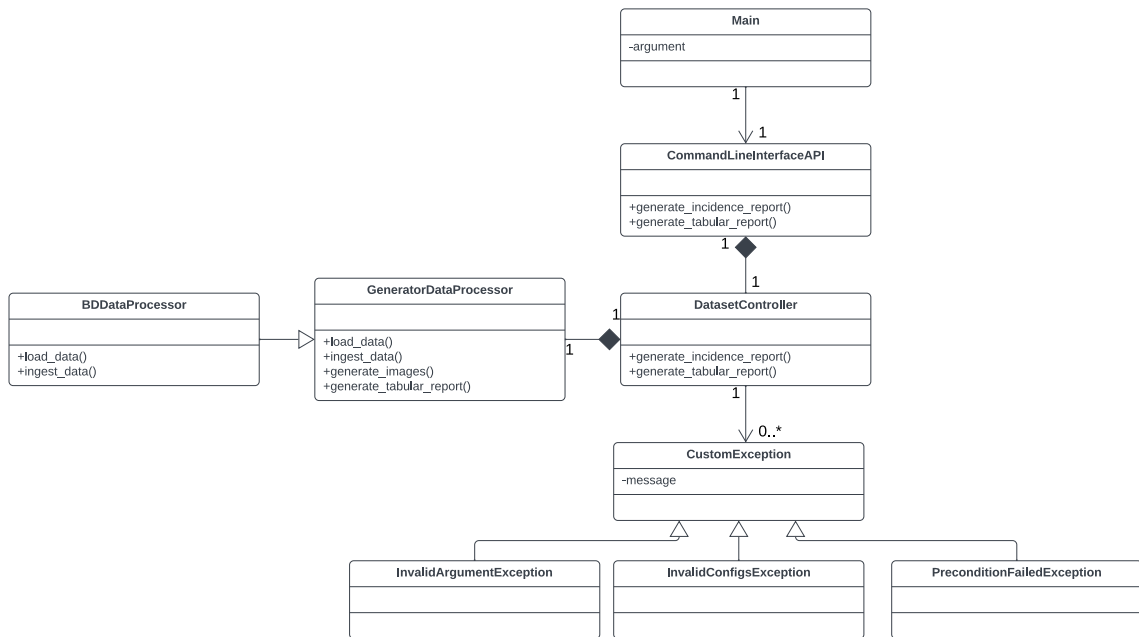


Figure 4.4: Class diagram of the feature extraction module.

The CSV data is loaded into memory and ingested by the **GeneratorDataProcessor** class for generating the tabular and image feature space representations. Furthermore, the **GeneratorDataProcessor** class is also called by the **DatasetController** class to create the data representations; speed concerns relating to the data transformation are addressed by the use of the multiprocessing Python capabilities in conjunction with the partial arguments function.

The produced representations are of the tabular format and the image format; Table 4.4 and Figure 4.5, respectively, provide dataset obfuscated examples of each.

⁷<https://pandas.pydata.org/>

⁸<https://matplotlib.org/>

Table 4.4: Dataset obfuscated tabular format representation.

nr_total_previous_time_periods	nr_total_api_requests	api_requests_sequence
2	60	"A-B-D-E-N-B"
A_incidence	...	Z_incidence
1	...	0

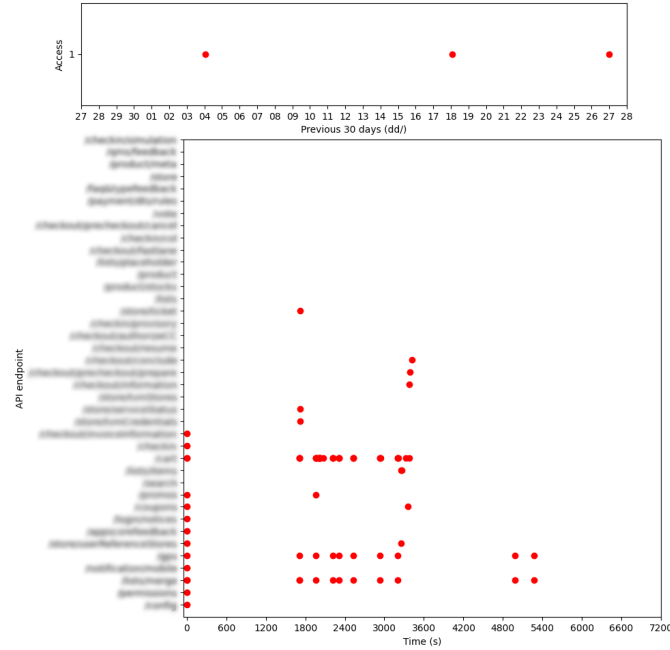


Figure 4.5: Dataset obfuscated image format representation.

4.1.2 Analysis Component

The system analysis component concerns all the ML processes required to train and obtain predictions from the two proposed models, including data pre-processing and the decision process. It comprises two modules: the traffic analysis module responsible for the two models and the decision module, which handles the decision process.

These module implementations are primarily independent of business logic.

4.1.2.1 Traffic analysis module

The Traffic analysis module was developed in Python using the TensorFlow Framework as detailed in 3.3.4, furthermore the category-encoders⁹ package was adopted for some necessary data pre-processing capabilities. The module follows a structure that prioritizes responsibility segregation between the different packages and classes.

⁹https://contrib.scikit-learn.org/category_encoders/

Figure 4.6 presents this module's simplified UML class diagram, showing the most important classes and connections.

The module entry point is the Main class for command-line management access. The command-line access allows new models to train and emit API traffic classifications. The Main class receives an argument; this argument is parsed and communicated to the CommandLineInterfaceAPI functioning as an API layer, which uses the controller class MLModelController to perform the necessary model-related operations.

The controller class uses other classes to perform operations on the data and models or access configurations and deal with exceptions.

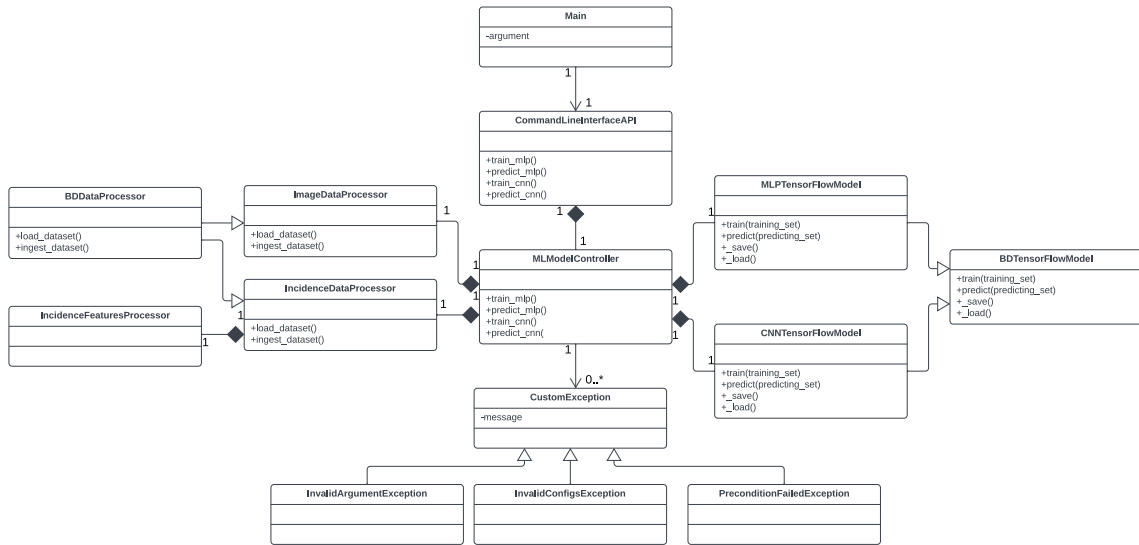


Figure 4.6: Class diagram of the traffic analysis module.

The proposed NN models consume different data formats and require different ML processes, facing various challenges in the two system phases.

The module is responsible for data pre-processing for the two data formats, and the creation, training, evaluation, serialization, and prediction process of the MLP and the CNN models.

MLP model In the training phase, the MLP model consumes a labeled tabular dataset, which is loaded into memory and ingested by the IncidenceDataProcessor class. The IncidenceFeaturesProcessor class provides information about the different features, such as if a feature is numerical or categorical or if it should get discarded. Numerical features require normalization, performed by the IncidenceFeaturesProcessor class, which serializes the normalization parameters during the training phase for later use during the serving phase. Furthermore, the MLModelController class uses the MLPTensorFlowModel class to create the model, perform a training-validation split of the training dataset, and train, evaluate, and serialize the model. In the serving phase, the MLModelControllerClass uses the MLPTensorFlowModel class to deserialize the model. The MLP model predicting tabular set is loaded into memory and ingested by the IncidenceDataProcessor class;

the IncidenceFeaturesProcessor class provides feature information and normalization with the serialized training normalization parameters.

CNN model In the training phase, the CNN model consumes a labeled image dataset for training. The dataset is loaded into memory and undergoes image pre-processing, before being split into training-validation splits by the ImageDataProcessor class. Furthermore, the MLModelController class calls the CNNTensorFlowModel to create, train, evaluate and serialize the model. The module is responsible for the deserialization and prediction processes in the predicting phase. In the serving phase, the predicting image set is loaded into memory and ingested by the ImageProcessorClass; the MLModelControllerClass then uses the deserialized model on the predicting set.

4.1.2.2 Decision module

The decision module implemented in Python is responsible for receiving the classifications of API traffic set for a user entity in the analyzed period, and combining them to create a further refined classification linked to a user entity before sending it to the API gateway.

The module inputs the user entity identifier and the API traffic set predictions from the MLP and the CNN model; they are combined to form a final classification through an interchangeable algorithmic process before being sent to the API gateway.

4.2 Dataset-related Processes

A machine learning model, both when training and predicting, may require specific processes to ensure that the dataset is ready to be used. The MLP model requires handling numerical and categorical features and the training validation split process, while the CNN model requires image pre-processing and the training validation split processes; these processes are further described in the following subsections.

4.2.1 Handling Numerical Features

Numerical features can have a series of unit and scale differences. In order to eliminate the influence of unit and scale differences between numerical features and treat each numerical feature equally, a process called feature normalization exists. There are several techniques of feature normalization. This dissertation employed min-max normalization. This technique performs a linear transformation on the original data, keeping the relationship between the original data [76]. It can rescale the data between a specific range of values or simply between 0 and 1.

The min-max normalization process operates according to the following equation, where x is the original value, x' is the rescaled value, and the range is represented by $[a, b]$.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} * (b - a) + a \quad (4.1)$$

The normalization process was performed in the $[0, 1]$ range, therefore the equation can be simply described as:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \quad (4.2)$$

In order to perform min-max normalization, the minimum and maximum values of each feature are determined; these are called the normalization parameters, and they require saving so that the normalization process can perform similarly during the training and predicting flow.

4.2.2 Handling Categorical Features

Categorical feature values represent qualitative data, which do not have an order concerning one another. They have a finite vocabulary (e.g., a person's gender); these features require encoding before passing to the ML model. Several encoding techniques exist, one of the most popular being "one-hot encoding". This technique removes the original categorical feature column, transcribing the categorical feature vocabulary to different binary columns. The initial values map to these columns [77]. However, the "one-hot encoding" technique has problems when dealing with high-cardinality categorical features, such as the n-grams proposed feature. Efficient processing and storing of data using this technique becomes a severe problem [77], which was encountered during the development process.

This dissertation employs the baseN encoding technique, a variation of the binary encoding technique; this consists in assigning a numerical value to each present category and then converting it into its base N representation; the different digits from that base N representation are then placed into separate columns. For a feature with v -unique values, it results in $\log N(v)$ number of binary discrete values [78].

4.2.3 Image Pre-processing

Challenges are often faced when dealing with image models, such as various image dimensions requiring standardization, or large image dimensions, which increase the computational cost of model operations. For this dissertation, the images are scaled to (500,500) pixels. Furthermore, image data can be represented by a matrix with $(H * W * C)$ dimensions, where H represents height, W width, and C the number of channels. The matrix values belong to a domain specified by the image format, which may cause problems with the network training gradients. In this sense, image-wise normalization is applied, creating a more similar range for each feature. These decisions are supported by similar decisions taken in other state-of-the-art image models [79].

4.2.4 Training Validation Split

The training-validation dataset split is a process that allows for the model evaluation with never-seen data. The training dataset splits into two subsets before the training and evaluation processes, the training and evaluation subsets. The model uses the first subset to train the model. In contrast,

the trained model predicts the validation subset and provides KPI statistical information regarding its performance (e.g., accuracy, precision, recall, and f1 score). These are subsets of the training dataset, so they are labeled. The literature specifies that values between 20% and 25% are ideal [80] [81], furthermore the adopted ML framework, TensorFlow, has a 25% default value for this setting. This dissertation adopted a 20% value for the training validation split in the MLP and the CNN model. Furthermore, due to the n-grams feature in the tabular feature space (3.3.3.1), the tabular dataset may have multiple rows representing a user entity activity during a period. In the image data, only one image represents the user entity activity during a period. The developed solution assures that any period in the tabular training data does not appear in the validation tabular data for any user entity.

4.3 System Phases

The system works two sequential and repeating phases when deployed in a production environment, the training phase and the serving phase, corresponding to the update flow (UC5_A) and the analysis flow (UC7_A).

The training phase happens whenever the system is deployed and repeats via a scheduled task, notifying whenever updated training data is available, and serves as a precursor to the serving phase, which deploys the models for periodic traffic analysis. These phases are represented via UML machine state diagrams in Figures 4.7 and 4.8.

The previously reviewed implemented system components, modules and processes are observable in the Figures.

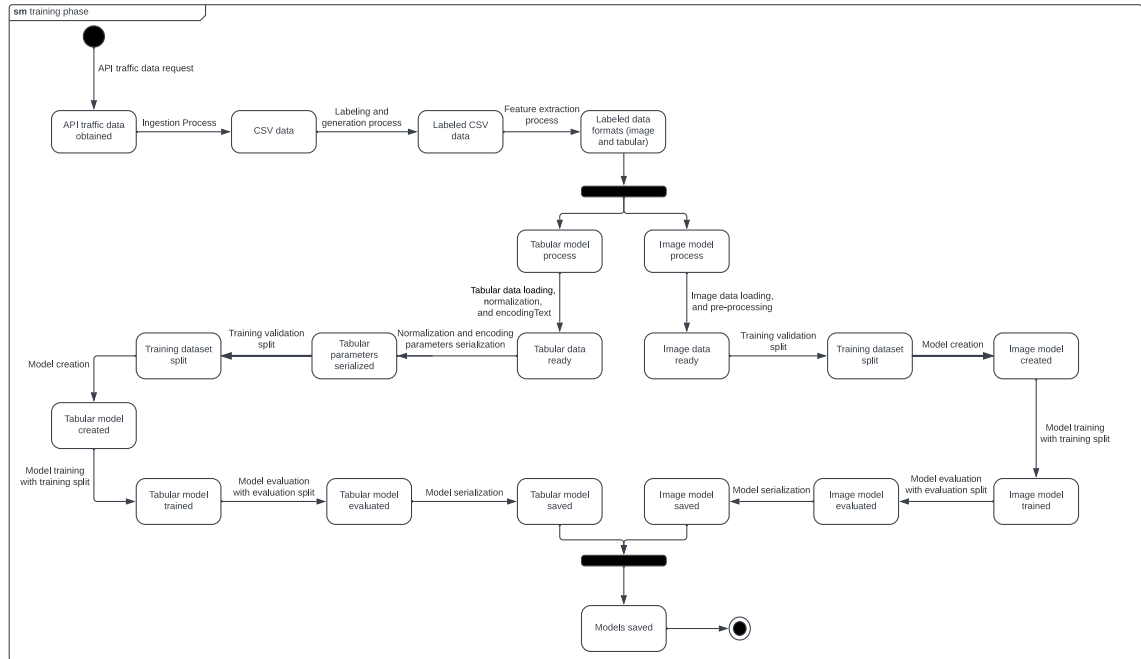


Figure 4.7: State machine diagram of the system's training phase.

In the training phase, API traffic data is obtained from the logging system and ingested by the ingestion module from the systems data component before undergoing the labeling and generation process to label the data. The labeled data undertakes the feature extraction process, coordinated by the feature extraction module, transforming the data into the image and tabular formats with the respective feature spaces. The traffic analysis module then creates a process for each model. The tabular data requires categorical feature encoding and numerical feature normalization, and the encoding and normalization parameters are serialized for use in the serving phase. In comparison, the image data undergoes pre-processing. The training data is then split into training and validation sets. The models are created and trained with the training split and evaluated with the evaluation split before being serialized for use in the serving phase.

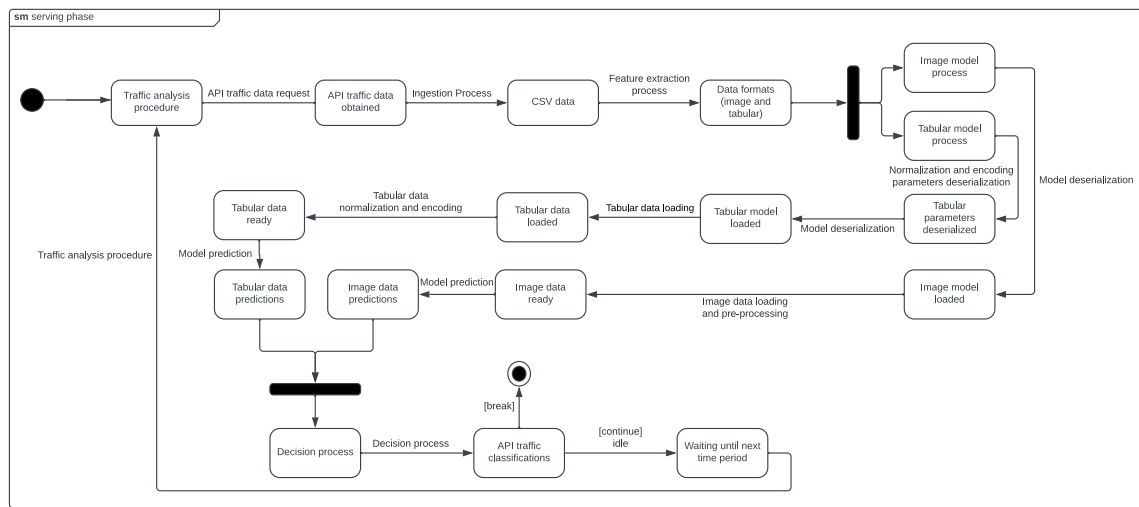


Figure 4.8: State machine diagram of system's serving phase.

The serving phase occurs after training and uses the previously trained and serialized models. This phase periodically, via a scheduled repetitive task, obtain API traffic data for prediction purposes of the currently under analysis period, which undergoes the ingestion and feature extraction processes, creating the image and tabular data formats with the proposed feature space. Then the controller deserializes both models in separate processes; the tabular model requires that the numerical data features are normalized, and the categorical features are encoded with the previously deserialized parameters. In comparison, the image model requires image pre-processing. The models then create predictions on the API traffic data, which are passed to the decision module for the decision process, creating API traffic classifications, passed to the API gateway; the process then may continue until the scheduled task triggers the training phase.

4.4 Software Tests

Software testing is a procedure used to ensure the code's efficiency and its operation's correctness. It serves as a quality assurance technique for the components of a software project. During testing

runs, in some instances, input data is passed to the application, which is invoked in its totality or only to run a specific part of the code base. After the code testing run ends, the output is checked for compliance with the expected result. If this comparison passes, the test is passed [82].

Several types of tests exist, including but not limited to: unit tests, integration tests, and system tests. Unit tests work on the smallest testable portions of code, called units; integration tests ensure the main goals of the different modules are full-filled; system tests provide confidence that the systems' requirements are fulfilled [83].

During the development of the API abuse detection system, integration and system tests were developed, aiming to reduce the possible number of software flaws and to ensure that future code-base modifications do not compromise the overall solution.

As detailed in the adopted technologies sub-section (3.3.4) the test framework chosen for this dissertation was the open-source pytest¹⁰ framework.

4.4.1 Integration Tests

During the development stage of this dissertation, no unit tests were written, which work to test the smallest possible parts of the codebase, such as functions or methods. However, integration tests were developed, which work as an extension of unit tests, ensuring that the functionality of a module and the communication between module units is as expected to test functionality as a whole [83].

Each module includes at least an integration test to ensure it functions as expected; these tests work by either comparing the produced output to the desired outcome of a module or by verifying that there are no raised exceptions during the testing run.

Figure 4.9 presents the output summary of an integration testing run. In total, nine integration testes were developed, while most modules are only tested by one integration test, the traffic analysis module, is tested by four, corresponding to the different processes that it is responsible for, including the training and serialization, and the deserialization and predictions of the two system models. The integration test run required 41m:38s and all tests passed.

```
===== short test summary info =====
PASSED ../test/test_integration.py::test_integration_logs_ingestion
PASSED ../test/test_integration.py::test_integration_labeling_and_generation
PASSED ../test/test_integration.py::test_integration_feature_processing_tabular
PASSED ../test/test_integration.py::test_integration_feature_processing_image
PASSED ../test/test_integration.py::test_integration_training_ann
PASSED ../test/test_integration.py::test_integration_predicting_ann
PASSED ../test/test_integration.py::test_integration_training_cnn
PASSED ../test/test_integration.py::test_integration_predicting_cnn
PASSED ../test/test_integration.py::test_integration_decision
===== 9 passed, 11 warnings in 2498.11s (0:41:38) =====
```

Figure 4.9: Integration testing run output.

¹⁰<https://pytest.org>

4.4.2 System Tests

System tests validate the systems' compliance with the specified requirements. In this case, the functional requirements and the success of system tests represents the success of a UC [83]. After all integration tests had achieved positive results, they were developed in the later stages of the development cycle and integrated the different modules to form the system's flows.

The two UC pertaining to the developed solution tested were UC5_A and UC7_A, corresponding to the update flow and the analysis flow, which encompass the other UC and represent the two system phases in a production environment. The first test aimed at UC5_A tests the training phase of the system, starting from API traffic log data up to the serialization of the trained models; the second test about UC7_A tested the serving phase, from the moment periodic API traffic log data is obtained up until API traffic classifications are emitted.

Figure 4.10 presents the output summary of a system testing run, containing both developed system tests, which took 1h:24m:49s, and both tests passed.

```
===== short test summary info =====
PASSED ../test/test_system.py::test_training_phase
PASSED ../test/test_system.py::test_servicing_phase
===== 2 passed, 11 warnings in 5089.46s (1:24:49) =====
```

Figure 4.10: System testing run output.

Chapter 5

Evaluation and Results

This chapter aims to describe the conducted experiments under the defined test hypothesis and the related experimentation and evaluation methods for these experiments. Furthermore, as the developed system relies on a machine learning-based solution consuming a dataset, a dataset study is also presented, including an exploratory data analysis of the tabular dataset format. The system machine learning-based solution comprises a multilayer perceptron and a convolutional neural network model, requiring slightly different processes for model architecture tuning during the experimentation phase. In addition, the developed system performance is evaluated through an experiment related to the serving phase, detailed in the implemented solution chapter.

It contains five sections: the test hypothesis, the experimentation and evaluation, the dataset study, the network topology and hyperparameter tuning, and the system performance evaluation section. The first section presents the test hypothesis related to the dissertation goals. The second section describes the experiments and evaluation methods used for the conducted experiments. The performed dataset study is detailed in the third section. The last two sections represent the conducted experiments related to the model architecture tuning and the system performance evaluation.

5.1 Test Hypothesis

This dissertation's primary goal is to test the hypothesis that ML algorithms can be used for API abuse detection. The developed solution employs two distinct NN models, which are separately tested but share the KPI used for evaluation.

Furthermore, as the developed solution consists of a system that can integrate into a business production setting in two phases, showcased in 4.3, the serving phase efficiency, responsible for API traffic classifications, is evaluated by measuring the execution time of the involved different data processing and API traffic analysis steps, to ensure the system can produce classifications in a suitable time range.

5.2 Experimentation and Evaluation

This section describes the ML-problem formulation and defines the guidelines for the experimentation of both tabular and image models and the systems performance evaluation, besides presenting the evaluation metrics and experiments conditions.

5.2.1 Evaluation Methodology

In line with the reviewed related work in section 2.2, this dissertation formulates the API abuse detection problem as a binary classification problem. Consequently, the trained ML models cannot distinguish API abuse by sequence from API abuse by volume, only API abuse from regular API traffic.

The system comprises two NN models: an MLP and a CNN model. In experimentation, these models' architectures are tuned by changing the network's topology and adjusting the hyperparameter configurations. The topology, hyperparameter search spaces, and tuning processes are respective to each model and rely on literature standards.

Additionally, the developed system is also evaluated for its performance in the serving phase related to UC7_A (see 3.2 and 4.3 sections) by collecting the execution time of the different involved processes: the logs ingestion process, the feature extraction process, the CNN classification process, the MLP classification process, and the decision process.

5.2.2 Evaluation Metrics

The problem is modeled as a binary classification problem, attempted by the proposed ML classification algorithms, the quality of which, as detailed in 2.1.6.2, can be measured using the statistical metrics of accuracy, precision, recall, and f1-score. In the API abuse domain, the metrics represent:

- **accuracy** - Number of correctly classified API traffic sets, divided by the total number of classified API traffic sets.
- **precision** - Number of correctly classified API traffic sets as API abuse, divided by the total number of API traffic sets classified as API abuse.
- **recall** - Number of correctly classified API traffic sets as API abuse, divided by the total number of API traffic sets that contain API abuse.
- **f1-score** - Harmonic mean of precision and recall.

The system serving phase, responsible for the periodic traffic analysis in a deployment setting, is also evaluated with basis on its analysis processes execution time to ensure the system can produce API traffic classifications in an opportune time range.

5.2.3 Experiments Conditions

The experiments were conducted in a machine of the proponents' organization courtesy, dubbed "Superman", running an Ubuntu LTS distribution, version 18.04.4, with the following specifications: Asus ROG STRIX Z370-G GAMING motherboard, Intel Core i7-8700K CPU, 64 GB Crucial DDR4 4x16GB RAM, Nvidia GeForce GTX 1070 8 GB GDDR5 GPU, and 480 GB Micron 5200 SSD storage.

5.3 Dataset study

As described in 3.3.3, the dataset is transcribed to tabular and image data formats with different feature spaces, which are analyzed in this section. It comprises two sub-sections: the first examines the experiments' dataset dimensions in the tabular and image format, and the second details the performed exploratory data analysis for the proposed tabular format.

5.3.1 Overview

The experiments' training dataset contained 32022 periods, each pertaining to a user entity, and represented 7200s of API usage. Approximately 50% (16112) represent normal API usage, 25% (7956) represent API abuse by volume examples, and the remaining 25% (7943) represent API abuse by sequence examples, creating a balanced dataset when modeling the problem as a binary classification problem.

The dataset, when transcribed to the image format, each image having 500x500 pixel dimensions, required 32022 images, one for each period, and when transcribed to the tabular format required 6151197 total rows to represent the 32022 periods.

5.3.2 Exploratory Data Analysis

The tabular form of the training dataset was the subject of an exploratory data analysis with the intent of better understanding the synthetic dataset. The investigation took into consideration the different labels and the whole dataset. The information acquired during this exploration is in full in appendix A.

The two most relevant numerical features analyzed were `nr_total_api_requests` and `nr_total_previous_time_periods`; while all features are a by-product of the API endpoint consumption, these specific features could leak undesired information to the model.

The maximum and minimum values for the `nr_total_previous_time_periods` feature are:

- 42 and 1, in regard to the entire dataset.
- 42 and 1, regarding to the normal label dataset split.
- 15 and 1, regarding to the abuse by volume label dataset split.
- 17 and 1, regarding to the abuse by sequence label dataset split.

Figure 5.1 presents a column chart representing the feature's average and standard deviation values using colored columns. The colors concern different labels and the entire dataset. It shows that the different splits' average and standard deviation values are very similar and will likely not leak any undesired information to the model.

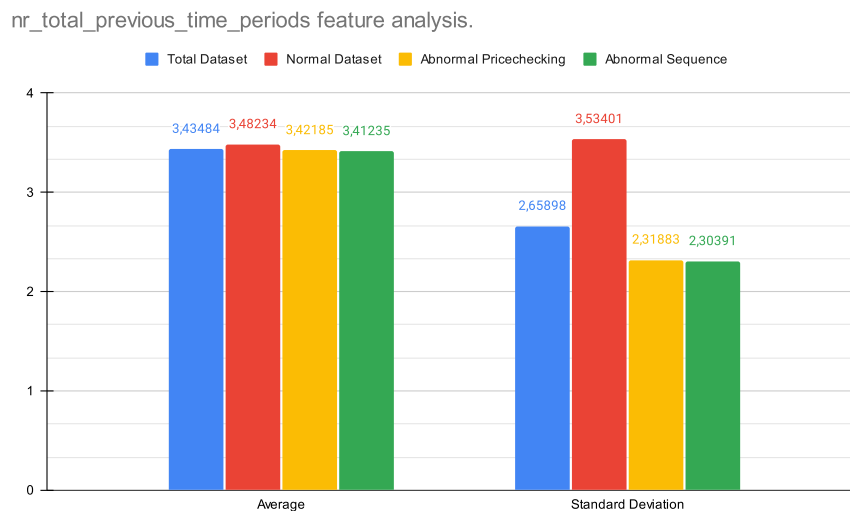


Figure 5.1: Column chart for feature `nr_total_previous_time_periods` analysis.

The maximum and minimum values for the `nr_total_api_requests` feature are:

- 1571 and 15, in regard to the entire dataset.
- 768 and 15, regarding the normal label dataset split.
- 1571 and 28, regarding the abuse by volume label dataset split.
- 400 and 23, regarding the abuse by sequence label dataset split.

Figure 5.2 presents a column chart representing the feature's average and standard deviation values using colored columns. The colors concern the different labels and the entire dataset. It shows that the different splits' average and standard deviation values are inconsistent and could leak some information to the model if left unattended.

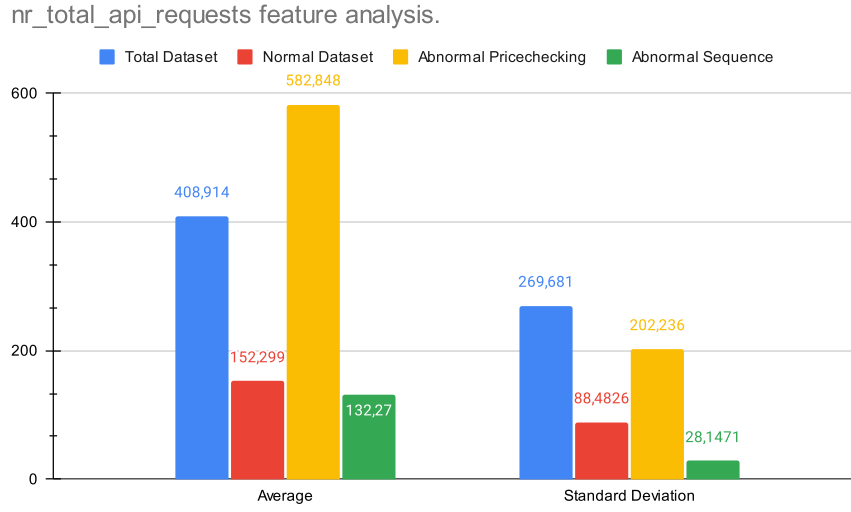


Figure 5.2: Column chart for feature nr_total_api_requests analysis.

As a consequence of the exploratory data analysis, it was concluded that the total number of API requests for a period feature data pattern might have allowed the model to derive whether the use was normal or abnormal due to average and standard deviation values patterns with regard to the different splits, so the decision was made not to use the feature "nr_total_api_requests".

5.4 Network Topology and Hyperparameter Tuning

As described in the previous experimentation and evaluation section, the MLP and CNN models are experimented on separately, partly because, MLP are NNs with very simple architectural specification, while CNNs do not have any architectural specification, causing the two model architecture tuning processes to diverge.

5.4.1 MLP Experiment

As previously studied (see 2.1.1.1), MLPs are feedforward, fully connected, at least one hidden layer deep, NNs; naturally, the most basic MLP architecture contains only one hidden layer, making it the experiment starting MLP topology.

Furthermore, the MLP model includes a set of hyperparameters tuned to perform in the dataset. Below we describe these parameters and, whenever possible, the literature standards.

- The learning rate influences to what extent newly acquired information supersedes old information [84].
- The number of training epochs defines the number of times the learning algorithm will work through the entire training dataset [85].

- The batch size defines the number of samples to work through before updating the internal weights [85].
- The number of neurons in the hidden layers influences the model's ability to capture greater complexity.
- L1 & L2 regularization, respectively known as Lasso and ridge regression, encourages weights to drop to zero or approximations of zero, effectively removing the corresponding feature from the model.

The L1 regularization method adds the sum of the absolute values of the model weights to the objective function as a penalization term [86].

The L2 regularization method adds the sum of the squares of the model weights to the objective function as a penalization term [86].

In other words, L1 regularization drives the weight given to irrelevant features to zero, while L2 regularization drives the weight given to irrelevant features to an approximation of zero [86].

- The Dropout regularization method, introduced by Hinton [87] in 2012, provides a simple technique for avoiding overfitting in feedforward neural networks. In each training iteration, any neuron has a probability p of being discarded from the network. When the network finishes training, the neuron's output multiplies by the likelihood that the neuron was omitted in a training iteration, creating an average over the possible networks used during training. According to the original paper, the dropout rate (p) should be between 0.2 and 0.5.

5.4.1.1 Experiment method

The simplest MLP model contains one hidden layer and is the experiment starting topology. The MLP model hyper-parameter are tuned according to the bayesian optimization hyperparameter tuning method described in the state of the art section (see 2.1.8), widely regarded as the optimal hyperparameter tuning strategy, the designated objective of this process was to maximize the accuracy in the validation split, each step contains a maximum of thirty-two trials: the number of trials provides statistical confidence that the model is optimally tuned [88]. More hidden layers may be added if the model cannot capture the dataset complexity.

The defined possible values for the different hyperparameters search spaces are exposed in table 5.1. The first trial chosen value for any hyperparameter setting does not limit or influence the other hyperparameter selected values.

Table 5.1: Hyperparameters search spaces.

Learning Rate	0.01		0.05		0.1		
Training Epochs	32		64		96	128	
No. Neurons Hidden Layer	64	128	192	256	320	384	512
L1 regularization parameter	0		0.001		0.025		0.05
L2 regularization parameter	0		0.001		0.025		0.05
Dropout Rate	0.2		0.3		0.4		0.5

5.4.1.2 Experiment results

The trials of the bayesian optimization process are described in full in the appendix A. Here table 5.2 presents the three best performing hyperparameter configurations yielded from the 32 trials - **lr** represents the learning rate, **epochs** the number of training epochs, **h1** the number of neurons in the hidden layer, **L1** the L1 regularization parameter, **L2** the L2 regularization parameter, **p** the dropout rate; furthermore, it details the training duration and the accuracy in the validation set of each trial in columns **duration** and **validation accuracy** respectively.

The output of the process is limited to the metric we are trying to optimize; in this case, the model's accuracy against the validation split, the output further confines the metrics' display to two decimal places. The hyperparameter optimization process took in a total of 9h:38m:24s.

Table 5.2: Bayesian optimization process best performing hyperparameter configurations.

Trial #	lr	epochs	h1	L1	L2	p	duration	validation accuracy
7	0.01	128	64	0	0	0.3	31m:21s	99.99%
23	0.01	96	64	0	0	0.5	23m:15s	99.99%
28	0.05	64	512	0	0	0.4	15m:39s	99.99%

The three hyperparameter configurations yielded from the Bayesian optimization strategy were exported and tested again on fresh models, trained on the training split, and evaluated on the validation split to obtain the remainder of the evaluation metrics. The results are present in Table 5.3.

Table 5.3: Best performing hyperparameter configurations evaluation metrics.

Trial #	validation accuracy	validation precision	validation recall	validation f1-score	training time
7	99.98%	99.96%	99.99%	99.97%	24m:11s
23	99.99%	99.99%	99.99%	99.99%	18m:12s
28	99.99%	99.99%	99.99%	99.99%	12m:40s

5.4.2 CNN Experiment

As previously studied (see 2.1.1.1) CNNs are NNs that specialize in image pattern recognition; unlike MLPs, they do not have any specifications other than being suited for image-focused tasks [27]. For this, they use convolutional and pooling layers, the number of which and their arrangement does not yet have a practical solution other than the researcher's experience [89].

Neural Architecture Search (NAS) for CNNs is a very complex problem, the main focus of the current research is RL and Evolutionary algorithms [90], [91]. However, the produced CNN architectures are not yet able to surpass manually designed models in the ImageNet¹ dataset. Neural architectures that demonstrate positive results against the ImageNet dataset can generally transfer to new image datasets. The implicit assumption is that better performing CNN architectures on the ImageNet dataset transfer better to other vision tasks [92], [93].

As the LeNet-5 (1998) [94] CNN architecture marks the emergence of the CNN and defines a CNN's essential components, and it is the CNN starting architecture.

In addition to the standard hyperparameters of the MLP model, the CNN model includes an additional set of hyperparameters, reviewed in the state-of-the-art section (see 2.1.1.1). However, the experiment does not directly interfere with these configurations, as the used CNN architectures specify their hyperparameter configurations.

5.4.2.1 Experiment method

The experiment starting CNN architecture, the LeNet-5 CNN architecture, is academically and historically significant. Still, it succeeded in regards to the Modified National Institute of Standards and Technology (MNIST²) dataset and could not extend to more complex image recognition applications.

The MNIST dataset consists of 70000 low-resolution black and white images of handwritten digits, split into 60000 images for training and 10000 for testing purposes.

The previously introduced ImageNet dataset, on the other hand, comprises 14197122 labeled high-resolution images belonging to 21841 different subcategories. The images are hand-labeled and collected from the web.

Since 2010, an annual competition entitled the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) has occurred. The ILSVRC³ uses a subset of ImageNet of 1000 classes, with roughly 1000 examples per class, for the training split, as well as 50000 and 150000 randomly sampled images for validation and testing splits, making up for roughly 1.2 million images in the total training dataset. The winners of the ILSVRC competition are, in academia, regarded as the state of the art in CNN architecture [95], [79], [96], [97].

Since the scope of NAS for CNN falls outside the scope of this dissertation, the experiment uses prominent CNN architectures in the literature throughout the years. Starting with the LeNet-5 architecture and then iterating through the CNN architectures that won the yearly ILSVRC

¹<https://image-net.org/index.php>

²<http://yann.lecun.com/exdb/mnist/>

³<https://image-net.org/challenges/LSVRC/>

competition, while it was computationally feasible, the tested CNN architectures include the ILSVRC winners from 2012 until 2014, respectively, AlexNet (2012) [95], ZfNet (2013) [79], and GoogLeNet (2014) [96].

A short overview of CNN history regarding the ILSVRC competition, including the architectural approach, is provided in Appendix A.

5.4.2.2 Experiment results

The experiment adapted the original CNNs architecture with minimal input and output layers changes to satisfy the dataset and problem modulation requirements. The models were trained during ten epochs.

Table 5.4 presents the experiment conducted on the image form of the dataset, the used architectures and their metrics against the validation set, along with the duration of the training process. The evaluation metrics are limited to two decimal places.

Table 5.4: Experiment architectures and evaluation metrics.

Model	validation accuracy	validation precision	validation recall	validation f1-score	duration
LeNet-5 (1998)	50.94 %	50.94%	100%	67.50%	13m:54s
AlexNet (2012)	99.97 %	99.97%	99.97%	99.97%	28m:37s
ZfNet (2013)	99.98 %	99.99%	99.97%	99.98%	43m:19s
GoogLeNet (2014)	99.84%	99.72%	99.97%	99.84%	1h16m:55s

As displayed in the Table, the LeNet-5 model took only 13m:54s to train, while AlexNet, ZfNet, and GoogLeNet models, took 28m:37s, 43m:19s, and 1h16m:55s, respectively.

The LeNet-5 results demonstrate 50.94% values for accuracy and precision metrics while having 100% recall and an f1-score of 67.50% in the validation set. These values are strange and required further testing involving the trained model. The subsequent tests made it apparent that the network learned only to predict the positive class corresponding to regular use, maintaining the accuracy and precision at close to 50% with a balanced dataset, and creating a meaningless 100% value for the recall metric, also influencing the f1-score.

The other architectures, however, achieved over 99% in all metrics, while all obtained a value of 99.97% in the recall metric; ZfNet very slightly outperformed the rest in all other metrics, with a middle-of-the-pack training time of 43m:19s.

5.5 System Performance Evaluation

The developed system could be deployed in production, and would function in two sequential and repetitive phases, described in section 4.3. In this setting the serving phase would expectantly be predominant, as it provides constant API traffic analysis.

An experiment was developed and conducted to evaluate the system's performance in such a setting, evaluating the systems involved processes' efficiency based on their execution time. The MLP and CNN models used in this experimental setting were selected from the previous experiments, using criteria depending on their evaluation metrics and the training time; they are the MLP model corresponding to trial #28 and the CNN model using the AlexNet architecture.

The time-measurements, listed in Table 5.5, pertain to the log ingestion, the feature extraction, the CNN model prediction, the MLP model prediction, and the decision processes; the logs ingestion process took 6.625s, the data pre-processing step had a duration of 22.436s; but the dataset transcription to the image format mainly characterized this execution time costing a total of 19.418s. The CNN classifications for the whole period took 4.3s, the MLP classifications took 0.929s, and the decision module took 0.001s to process the 480 users present during this period. In total, the system required 33.362s to perform the operations involved in the periodic traffic analysis procedure.

Table 5.5: System's periodic traffic analysis procedure execution time.

Logs ingestion	6.625s
Data pre-processing	22.436s
CNN prediction	4.300s
MLP prediction	0.929s
Decision Module	0.001s
Total	33.362s

5.6 Results Discussion

The MLP and CNN models used in the prototype system were selected from the experimented models (see section 5.4), using criteria depending on their evaluation metrics and the training time; they are the MLP model corresponding to trial #28 and the CNN model using the AlexNet architecture. Their captured evaluation metrics and the required training time is displayed in Table 5.6.

Table 5.6: MLP and CNN models in the prototype system.

Model	validation accuracy	validation precision	validation recall	validation f1-score	duration
MLP	99.99%	99.99%	99.99%	99.99%	12m:40s
CNN	99.97 %	99.97%	99.97%	99.97%	28m:37s

The achieved metrics are in line with other work in this field [15]. The API traffic abuse problem, in conjunction with the appropriate feature spaces (see section 3.3.3), creates a very simple, concrete, and low ambiguity representation of the data. The evaluation metrics produced by using the tabular and image models on never before seen data validate that there was no overfitting and that the ML models managed to capture the complexity of dataset formats.

Furthermore, an experiment was conducted to evaluate the system performance under a production setting, where the routine API traffic analysis procedure steps execution time was collected. The time measurements were described in section 5.5 and summarized in Table 5.5; in total, the system required 33.362 to perform the operations involved in the routine traffic analysis procedure.

This result serves as a baseline for the periodic traffic analysis procedure. It is two orders of magnitude smaller than the duration of the API traffic analyzed, making adopting the API abuse detection system feasible for the considered period duration.

Theoretically, a case of API abuse by sequence is not necessarily a volumetric anomaly of API traffic. However, the fantastic ability of the system ML models to capture the dataset complexity, the time period duration used in this dissertation - 7200s, and the exploratory data analysis of the experiments tabular dataset format leads to the hypothesis that in practice and especially when dealing with extensive enough periods it almost always is. Because to create a sequential anomaly, an attacker has to probe the system; when he finally discovers an attack vector, the already generated traffic on top of the traffic generated by the exploitation of the attack vector will possibly cause a case of API abuse by volume.

Chapter 6

Conclusions and Future Work

The API abuse field has been gaining momentum recently: first, in 2017, Gartner predicted that API abuse would be the most common computer attack by 2022 [98]; in 2019, the OWASP released the first iteration of their API Security list [6]; a report released in 2022 by Radware informed that in the previous year, more than 50% of surveyed organizations suffered monthly API attacks [14].

Companies who have to offer multiple solutions will, usually, distribute these services throughout a architectural approach, called microservices architecture, containing the loosely coupled and distinct services, called microservices [99]. And because different microservices offer distinct APIs, these vulnerabilities manifest in various forms; furthermore, API abuse in a specific API might be regular traffic in another API. As such, one of the solutions to the API abuse problem might be to study the underlying patterns of API traffic, which is what ML algorithms do; they learn from the underlying patterns in the data to make predictions about a topic.

This dissertation's goals were within this line and consisted of developing an API abuse detection system containing an ML-based solution to test the hypothesis that ML algorithms could efficiently and successfully apply to the API abuse problem.

The state-of-the-art study considered several machine learning concepts and studied related academic works to deepen the author's understanding of the problem and the related challenges. The decision to study the related fields of network traffic anomaly detection and malware detection was influenced by the lack of academic and scientific work regarding the API abuse detection field and helped crystallize the innovative tabular and image feature spaces used in this dissertation, allowing both API abuse by volume and API abuse by sequence detection.

The API traffic dataset used in this dissertation was extracted from real-world API traffic log data facilitated by the proponent organization, which included two business-specific data points of the two types of API abuse. The abusive patterns undergo a process designed to introduce data variability and create more data points while ensuring their integrity, quality, and veracity before being transcribed into the proposed formats.

Several development artifacts were created during the design and development of this dissertation's API Abuse detection system. The system follows an offline learning methodology (see

section 2.1.5) and would work in a production setting in two sequential and repetitive phases, the training phase and the serving phase, with the serving phase being responsible for systematic API traffic analysis. Moreover, the state-of-the-art study supported the decision to rely on the TensorFlow framework for the system MLP and CNN models, which consume the tabular and image dataset formats.

ML models require several processes to prepare a dataset for use, creating some implementation challenges. In particular, the handling of categorical features and the training validation split of the dataset tabular data format proved quite problematic due to the dataset's dimensions and complexity.

The MLP and CNN experiments dataset was studied, being the tabular dataset format subjected to an exploratory data analysis, which allowed a better understanding of the data, and permitted the blocking of features that could leak undesirable information to the models.

For the experiments, the API abuse problem was modeled as a binary classification problem and attempted by the systems' MLP and CNN models, similarly to related work. The model's architecture was tuned, attempting to maximize the evaluation metrics by changing the network topology and hyperparameter configurations according to literature standards. The results show that both MLP and CNN models could capture the experiment's dataset complexity, achieving excellent marks in both data formats hold-out evaluation splits, supporting the test hypothesis confirmation.

As the prototype would provide API traffic analysis while in the serving phase, an experiment was conducted to evaluate the system's performance in this phase, ensuring the system could function in a production setting.

6.1 Achieved Requirements

This dissertation's goals consisted of the development of an ML-based API abuse detection system capable of API traffic analysis. The problem and proposed solution chapter included a requirement analysis section, 3.2, which identified the systems' functional and non-functional requirements.

The system specific functional requirements were represented by five Use Cases and their fulfillment was validated by the system tests (see section 4.4) pertaining to UC5_A and UC7_A, as these UC encompass the other auxiliary UC, and represent the system phases in a production environment. They are:

- **UC5_A:** Update the models.
- **UC7_A:** Perform periodic API traffic analysis.

The non-functional requirements followed the FURPS+ [73] specification and were deeply taken into account during every development phase of the implemented solution. As such, they are assured and reflect themselves in the final quality of the solution.

6.2 Limitations and Further Work

While this dissertation was successful regarding the presented ML approaches to the API abuse problem and successfully achieved the proposed goals, the developed body of work has some limitations.

This dissertation explored the concept of path normalization from the related work section. The path normalization process allows for the collapse of the variable parts of a state. Cloudflare [55] proposes an unsupervised approach to the problem, leading to some attempts to apply this proposed solution. The attempts experimented with clustering algorithms to attempt a more general approach to path normalization, but perhaps because not much information is available, the attempts were unsuccessful. This process uses business logic for the developed system and remains a systems limitation. It makes for an exciting avenue for further work. A possible solution would be a naming convention for API endpoints, perhaps in the form of a Request for Comments (RFC) document.

Attackers could take extra precautions to make their abusive traffic look more natural, attempting to remove the data patterns that the models rely on. In this line, the feature space is another camp that requires further work. New previously unconsidered features might allow for a more accurate representation of the characteristics of an API traffic period, allowing ML models to better respond to the API abuse detection problem.

The data formats vary in their data representation. The tabular data format presents a very "factual" interpretation of the data, in contrast with the more interpretable image data format, where API traffic period features are imprinted into an image. The image format may lose some exactness due to representation constraints such as the image dimension versus the considered period duration. This could be considered a limitation.

Additionally, different period durations might yield different experiment results. On the one hand, shortening the period might allow for a more fine-grained analysis of API traffic; on the other hand, sequential and volumetric anomalies might be cut off for analysis earlier and miss the system due to a lack of "context". So experimenting with different periods is yet another avenue for further work.

API vulnerabilities are ever-changing. Fixing an API vulnerability might cause another due to the underlying codebase structure. Even a sequence that might have previously been abusive may turn into a regular sequence. In addition, it might be faster to fix a specific vulnerability in an API than to document it and introduce it to the API dataset. These factors led to the author's belief that the most significant barrier to adopting ML-based API abuse detection systems is the creation and maintainability or availability of a dataset capable of applying to the underlying API solutions. The system is also limited in this regard, as changing or adopting more API abuse patterns requires significant manual intervention. Further work simplifying and automating the dataset creation pipeline processes could mitigate this limitation.

While the dataset creation pipeline already has data generation capabilities, which rely on several statistical methods to introduce data variability to the original API abusive patterns. They

require human intervention to ensure the correctness of the generation parameters configurations for each considered API abuse pattern, which constitutes an API vulnerability documentation limitation and concerns the automation of the dataset creation processes. Developing a data generation solution that can infer what methods and parameters to apply to the input API abuse pattern to create more quality data points would undoubtedly help, perhaps again using an ML solution, and is another avenue of further work.

The system performance evaluation posted results that support the prototyped system's potential. However, the system could also be improved in this regard; the offline learning methodology limits the potential system up-time. An avenue of further work would be improving the system capabilities by implementing an online learning methodology.

References

- [1] Combating threats in the cyber world: Outsmarting terrorists, hackers, and spies, Mar 2012. <https://bit.ly/3Achgek>.
- [2] Krassimira Schwertner. Digital transformation of business. *Trakia Journal of Sciences*, 15(1):388–393, 2017.
- [3] Cisco. Digital vortex. <https://www.cisco.com/c/dam/en/us/solutions/collateral/industry-solutions/digital-vortex-report.pdf> Last accessed on 2022-06-07.
- [4] Daniel Jacobson, Greg Brail, and Dan Woods. *APIs: A strategy guide*. " O'Reilly Media, Inc.", 2012.
- [5] Dong Qiu, Bixin Li, and Hareton Leung. Understanding the api usage in java. *Information and software technology*, 73:81–100, 2016.
- [6] OWASP. Owasp api security project. <https://owasp.org/www-project-api-security/> Last accessed on 2022-06-07.
- [7] The hidden costs of cybercrime, Jun 2022. <https://www.csis.org/analysis/hidden-costs-cybercrime>.
- [8] Gartner. Gartner forecasts worldwide security and risk management spending to exceed \$150 billion in 2021. <https://www.gartner.com/en/newsroom/press-releases/2021-05-17-gartner-forecasts-worldwide-security-and-risk-managem> Last accessed on 2022-06-27.
- [9] Gartner. Forecast: Information security and risk management, worldwide, 2019-2025, 4q21 update. <https://www.gartner.com/en/documents/4009714> Last accessed on 2022-06-27.
- [10] Idan Amit, John Matherly, William Hewlett, Zhi Xu, Yinnon Meshi, and Yigal Weinberger. Machine learning in cyber-security-problems, challenges and data sets. *arXiv preprint arXiv:1812.07858*, 2018.
- [11] Uzair Bashir and Manzoor Chachoo. Intrusion detection and prevention system: Challenges & opportunities. In *2014 International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 806–809. IEEE, 2014.
- [12] Batta Mahesh. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).[Internet]*, 9:381–386, 2020.

- [13] Issam El Naqa and Martin J Murphy. What is machine learning? In *machine learning in radiation oncology*, pages 3–11. Springer, 2015.
- [14] Radware. The state of web application and api protection, Jan 2022. <https://www.radware.com/2020-application-security/>.
- [15] Gaspard Baye, Fatima Hussain, Alma Oracevic, Rasheed Hussain, and SM Ahsan Kazmi. Api security in large enterprises: Leveraging machine learning for anomaly detection. In *2021 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6. IEEE, 2021.
- [16] Gianni D’Angelo, Massimo Ficco, and Francesco Palmieri. Malware detection in mobile environments based on autoencoders and api-images. *Journal of Parallel and Distributed Computing*, 137:26–33, 2020.
- [17] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.
- [18] Ahmed Abd Elrahman, Mohamed El Helw, Radwa Elshawy, and Sherif Sakr. D-smartml: A distributed automated machine learning framework. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1215–1218. IEEE, 2020.
- [19] Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, and Thomas B Schön. Supervised machine learning. *Department of Information Technology, Uppsala University: Uppsala, Sweden*, page 112, 2019.
- [20] Facundo Bre, Juan M Gimenez, and Víctor D Fachinotti. Prediction of wind pressure coefficients on building surfaces using artificial neural networks. *Energy and Buildings*, 158:1429–1441, 2018.
- [21] Simon Haykin. *Neural networks and learning machines, 3/E*. Pearson Education India, 2009.
- [22] João Gama, André Ponce de Leon Carvalho, Katti Faceli, Ana Carolina Lorena, and Márcia Oliveira. *Extração de Conhecimento de Dados*. Edições Sílabo, 2017.
- [23] Ahmed Ali Mohammed Al-Saffar, Hai Tao, and Mohammed Ahmed Talab. Review of deep convolution neural network in image classification. In *2017 International conference on radar, antenna, microwave, electronics, and telecommunications (ICRAMET)*, pages 26–31. IEEE, 2017.
- [24] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.
- [25] Shifei Ding, Xinzheng Xu, and Ru Nie. Extreme learning machine and its applications. *Neural Computing and Applications*, 25(3):549–556, 2014.
- [26] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)*, pages 1–6. Ieee, 2017.
- [27] Keiron O’Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

- [28] Shujun Huang, Nianguang Cai, Pedro Penzuti Pacheco, Shavira Narrandes, Yang Wang, and Wayne Xu. Applications of support vector machine (svm) learning in cancer genomics. *Cancer genomics & proteomics*, 15(1):41–51, 2018.
- [29] Qiang He and Jun-Fen Chen. The inverse problem of support vector machines and its solution. In *2005 International Conference on Machine Learning and Cybernetics*, volume 7, pages 4322–4327. IEEE, 2005.
- [30] El-Houssainy A Rady and Ayman S Anwar. Prediction of kidney disease stages using data mining algorithms. *Informatics in Medicine Unlocked*, 15:100178, 2019.
- [31] T. Soni Madhulatha. An overview on clustering methods. *CoRR*, abs/1205.1117, 2012.
- [32] Qingyang Xu, Zhe Wu, Yiqin Yang, and Li Zhang. The difference learning of hidden layer between autoencoder and variational autoencoder. In *2017 29th Chinese Control And Decision Conference (CCDC)*, pages 4801–4804. IEEE, 2017.
- [33] Chaudhari Amol Mohammad. Autoencoder and pca for dimensionality reduction on mnist dataset., Nov 2019. <https://bit.ly/3ypP71M> Last accessed on 2022-06-07.
- [34] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.
- [35] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [36] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2019.
- [37] Venkat Gudivada, Amy Apon, and Junhua Ding. Data quality considerations for big data and machine learning: Going beyond data cleaning and transformations. *International Journal on Advances in Software*, 10(1):1–20, 2017.
- [38] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM computing surveys (CSUR)*, 27(3):326–327, 1995.
- [39] H Jabbar and Rafiqul Zaman Khan. Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study). *Computer Science, Communication and Instrumentation Devices*, 70, 2015.
- [40] João PS Rosa, Daniel JD Guerra, Nuno CG Horta, Ricardo MF Martins, and Nuno CC Lourenço. Overview of artificial neural networks. In *Using artificial neural networks for analog integrated circuit design automation*, pages 21–44. Springer, 2020.
- [41] Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.
- [42] Masupha Lerato, Omobayo A Esan, Ashley-Dejo Ebunoluwa, SM Ngwira, and Tranos Zuva. A survey of recommender system feedback techniques, comparison and evaluation metrics. In *2015 International Conference on Computing, Communication and Security (ICCCS)*, pages 1–4. IEEE, 2015.

- [43] Bob L Sturm. Classification accuracy is not enough. *Journal of Intelligent Information Systems*, 41(3):371–406, 2013.
- [44] Swati Gupta and Sushama Nagpal. An empirical analysis of implicit trust metrics in recommender systems. In *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 636–639. IEEE, 2015.
- [45] Payam Refaeilzadeh, Lei Tang, and Huan Liu. Cross-validation. *Encyclopedia of database systems*, 5:532–538, 2009.
- [46] Elad Hazan, Adam Klivans, and Yang Yuan. Hyperparameter optimization: A spectral approach. *arXiv preprint arXiv:1706.00764*, 2017.
- [47] Rémi Bardenet, Mátyás Brendel, Balázs Kégl, and Michele Sebag. Collaborative hyperparameter tuning. In *International conference on machine learning*, pages 199–207. PMLR, 2013.
- [48] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.
- [49] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.
- [50] Sebastian Raschka and Vahid Mirjalili. *Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2*. Packt Publishing Ltd, 2019.
- [51] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Alvaro Lopez Garcia, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52(1):77–124, 2019.
- [52] Jiangang Hao and Tin Kam Ho. Machine learning made easy: a review of scikit-learn package in python programming language. *Journal of Educational and Behavioral Statistics*, 44(3):348–361, 2019.
- [53] Marcus D Bloice and Andreas Holzinger. A tutorial on machine learning and data science tools with python. *Machine Learning for Health Informatics*, pages 435–480, 2016.
- [54] Eli Stevens, Luca Antiga, and Thomas Viehmann. *Deep learning with PyTorch*. Manning Publications, 2020.
- [55] Ben Solomon. Announcing api abuse detection, Jan 2022. <https://blog.cloudflare.com/api-abuse-detection/>.
- [56] Wai-Ki Ching and Michael K Ng. Markov chains. *Models, algorithms and applications*, 2006.
- [57] Dingde Jiang, Zhen Yuan, Peng Zhang, Lei Miao, and Ting Zhu. A traffic anomaly detection approach in communication networks for applications of multimedia medical devices. *Multimedia Tools and Applications*, 75(22):14281–14305, 2016.
- [58] Gao Yan. Network anomaly traffic detection method based on support vector machine. In *2016 International Conference on Smart City and Systems Engineering (ICSCSE)*, pages 3–6. IEEE, 2016.

- [59] Chie-Hong Lee, Yann-Yean Su, Yu-Chun Lin, and Shie-Jue Lee. Machine learning based network intrusion detection. In *2017 2nd IEEE International conference on computational intelligence and applications (ICCIA)*, pages 79–83. IEEE, 2017.
- [60] Pavol Mulinka and Pedro Casas. Stream-based machine learning for network security and anomaly detection. In *Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, Big-DAMA '18, page 1–7, New York, NY, USA, 2018. Association for Computing Machinery.
- [61] Sayali D Jadhav and HP Channe. Comparative study of k-nn, naive bayes and decision tree classification techniques. *International Journal of Science and Research (IJSR)*, 5(1):1842–1845, 2016.
- [62] Wilfriedzakie. Hoeffding-adaptative-trees-for-streaming-data. <https://github.com/wilfriedzakie/Hoeffding-Adaptative-trees-for-Streaming-data> Last accessed on 2022-06-07.
- [63] Adele Cutler, D Richard Cutler, and John R Stevens. Random forests. In *Ensemble machine learning*, pages 157–175. Springer, 2012.
- [64] Hamed Haddad Pajouh, GholamHossein Dastghaibiyfard, and Sattar Hashemi. Two-tier network anomaly detection model: a machine learning approach. *Journal of Intelligent Information Systems*, 48(1):61–74, 2017.
- [65] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.
- [66] Abdurrahman Pektaş and Tankut Acarman. Malware classification based on api calls and behaviour analysis. *IET Information Security*, 12(2):107–117, 2018.
- [67] Virus share. <https://www.virusshare.com/> Last accessed on 2022-06-07.
- [68] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer networks and ISDN systems*, 29(8-13):1157–1166, 1997.
- [69] Android malware genome project. <http://www.malgenomeproject.org/> Last accessed on 2022-06-07.
- [70] Contagio mobile. <http://contagiomindump.blogspot.com/> Last accessed on 2022-06-07.
- [71] Nviennot. Nviennot/playdrone: Google play crawler. <https://github.com/nviennot/playdrone> Last accessed on 2022-06-07.
- [72] Salvatore Ruggieri. Efficient c4. 5 [classification algorithm]. *IEEE transactions on knowledge and data engineering*, 14(2):438–444, 2002.
- [73] Peter Eeles. Capturing architectural requirements. *IBM Rational developer works*, 2005.
- [74] Robert B Grady and Deborah L Caswell. *Software metrics: establishing a company-wide program*. Prentice-Hall, Inc., 1987.
- [75] Isabelle Guyon, Steve Gunn, Masoud Nikravesh, and Lofti A Zadeh. *Feature extraction: foundations and applications*, volume 207. Springer, 2008.

- [76] S Patro and Kishore Kumar Sahu. Normalization: A preprocessing stage. *arXiv preprint arXiv:1503.06462*, 2015.
- [77] John T Hancock and Taghi M Khoshgoftaar. Survey on categorical data for neural networks. *Journal of Big Data*, 7(1):1–41, 2020.
- [78] Cedric Seger. An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing, 2018.
- [79] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.
- [80] Ramesh Medar, Vijay S Rajpurohit, and B Rashmi. Impact of training and testing data splits on accuracy of time series forecasting in machine learning. In *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, pages 1–6. IEEE, 2017.
- [81] Xavier dos Santos Silva. *Recommender Systems for Grocery Retail-A Machine Learning Approach*. PhD thesis, 2020.
- [82] Sanjay Kumar Singh and Amarjeet Singh. *Software testing*. Vandana Publications, 2012.
- [83] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software testing techniques: A literature review. In *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*, pages 177–182. IEEE, 2016.
- [84] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.
- [85] Jason Brownlee. What is the difference between a batch and an epoch in a neural network. *Machine Learning Mastery*, 20, 2018.
- [86] Ozgur Demir-Kavuk, Mayumi Kamada, Tatsuya Akutsu, and Ernst-Walter Knapp. Prediction using step-wise l1, l2 regularization and feature selection for small data sets with large number of features. *BMC bioinformatics*, 12(1):1–10, 2011.
- [87] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.
- [88] Ryan Turner, David Eriksson, Michael McCourt, Juha Kiili, Eero Laaksonen, Zhen Xu, and Isabelle Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In *NeurIPS 2020 Competition and Demonstration Track*, pages 3–26. PMLR, 2021.
- [89] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.
- [90] Yesmina Jaafra, Jean Luc Laurent, Aline Deruyver, and Mohamed Saber Naceur. Reinforcement learning for neural architecture search: A review. *Image and Vision Computing*, 89:57–66, 2019.

- [91] Tahereh Hassanzadeh, Daryl Essam, and Ruhul Sarker. Evodcnn: An evolutionary deep convolutional neural network for image classification. *Neurocomputing*, 488:271–283, 2022.
- [92] Simon Kornblith, Jonathon Shlens, and Quoc V Le. Do better imagenet models transfer better? In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2661–2671, 2019.
- [93] Lukas Tuggerer, Thilo Stadelmann, and Jürgen Schmidhuber. Imagenet as a representative basis for deriving generally effective CNN architectures, 2022.
- [94] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [95] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.
- [96] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
- [97] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.
- [98] Gartner. How to build an effective api security strategy. <https://www.gartner.com/en/documents/3834704> Last accessed on 2022-07-01.
- [99] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.
- [100] Yuanqing Lin, F Lv, S Zhu, M Yang, T Cour, K Yu, L Cao, Z Li, MH Tsai, X Zhou, et al. Imagenet classification: fast descriptor coding and large-scale svm training. *Large scale visual recognition challenge*, 2010.
- [101] Jorge Sánchez and Florent Perronnin. High-dimensional signature compression for large-scale image classification. In *CVPR 2011*, pages 1665–1672. IEEE, 2011.

Chapter 7

Appendix A

7.1 Exploratory Data Analysis

This section includes the information acquired during the exploratory data analysis done on the tabular form of the experiments dataset. It comprises three subsections, one considering the entire training dataset and three further sub-sections respective to each label in the dataset.

7.1.1 Total Dataset Analysis

The several descriptive statistics captured about the entire dataset, summarizing the central tendency, dispersion, and shape of the dataset distribution, are present in Table 7.1.

Table 7.1: Descriptive statistics about the total experiment training dataset.

feature	mean	std	min	25%	50%	75%	max
nr_total_timeslots	3.43484	2.65898	1	1	3	5	42
nr_total_incidence	408.914	269.681	15	147	375	662	1571
AA_incidence	0.333982	1.06864	0	0	0	1	56
AB_incidence	0.759631	1.07262	0	0	1	1	56
AC_incidence	8.47635	11.6972	0	3	5	5	170
AD_incidence	0.936096	1.32214	0	0	1	1	55
AE_incidence	6.18521	10.4006	0	1	3	3	82
AF_incidence	2.54491	3.042	0	1	1	5	75
AG_incidence	0.75742	1.0548	0	0	1	1	55
AH_incidence	0.757403	1.05479	0	0	1	1	55
AL_incidence	1.0413	1.49553	0	0	1	1	56
AM_incidence	2.25032	1.67169	0	2	2	3	56
AN_incidence	0.262499	3.13298	0	0	0	0	140
AO_incidence	5.83082	5.09003	0	0	9	9	81

Continued on next page

feature	mean	std	min	25%	50%	75%	max
AP_incidence	354.758	293.104	0	61	341	633	1521
AQ_incidence	1.51673	0.915683	0	1	2	2	29
AR_incidence	0.901984	0.600622	0	1	1	1	18
AS_incidence	0.0465241	0.554547	0	0	0	0	33
AT_incidence	0.0502104	0.890738	0	0	0	0	133
AU_incidence	0.0185575	0.236149	0	0	0	0	9
AV_incidence	3.29233	5.31015	0	1	1	2	44
AX_incidence	2.05097	4.2135	0	0	0	1	34
AZ_incidence	13.1418	30.7417	0	0	0	1	257
BA_incidence	1.93805	4.32938	0	0	0	0	34
BB_incidence	0.00258698	0.0690895	0	0	0	0	6
BC_incidence	0.0105934	0.1114	0	0	0	0	6
BD_incidence	0.0339118	0.425217	0	0	0	0	32
BE_incidence	0.322714	0.523869	0	0	0	1	18
BF_incidence	0.660715	1.73893	0	0	1	1	143
BG_incidence	0.0159273	1.16887	0	0	0	0	125
BH_incidence	0.000494701	0.0384955	0	0	0	0	4
BI_incidence	0.00160473	0.0400269	0	0	0	0	1
BJ_incidence	0.00204724	0.134949	0	0	0	0	15
BL_incidence	0.0103872	0.114341	0	0	0	0	6
BM_incidence	0.000121115	0.0110045	0	0	0	0	1
BN_incidence	0.00070718	0.0322684	0	0	0	0	3
BO_incidence	0.000546235	0.023738	0	0	0	0	2
BP_incidence	0.000452107	0.0335547	0	0	0	0	4
BQ_incidence	0	0	0	0	0	0	0
BR_incidence	9.9818e-05	0.0099904	0	0	0	0	1
BS_incidence	0.000423495	0.0409677	0	0	0	0	5
n-gram_0	0.111555	0.314818	0	0	0	0	1
n-gram_1	0.0460137	0.209515	0	0	0	0	1
n-gram_2	0.155637	0.362511	0	0	0	0	1
n-gram_3	0.150439	0.357501	0	0	0	0	1
n-gram_4	0.0779504	0.268093	0	0	0	0	1
n-gram_5	0.178883	0.383254	0	0	0	0	1
n-gram_6	0.128326	0.334452	0	0	0	0	1
n-gram_7	0.155011	0.361915	0	0	0	0	1
n-gram_8	0.157106	0.363901	0	0	0	0	1
n-gram_9	0.82647	0.378705	0	1	1	1	1

Continued on next page

feature	mean	std	min	25%	50%	75%	max
BA_incidence	0.402902	0.526666	0	0	0	1	4
BB_incidence	0.0108019	0.140863	0	0	0	0	6
BC_incidence	0.0442328	0.224343	0	0	0	0	6
BD_incidence	0.141599	0.860072	0	0	0	0	32
BE_incidence	0.0529039	0.506578	0	0	0	0	18
BF_incidence	0.202596	3.42908	0	0	0	0	143
BG_incidence	0.0665046	2.38778	0	0	0	0	125
BH_incidence	0.00206563	0.0786414	0	0	0	0	4
BI_incidence	0.00670056	0.0815823	0	0	0	0	1
BJ_incidence	0.00854828	0.275655	0	0	0	0	15
BL_incidence	0.043372	0.230564	0	0	0	0	6
BM_incidence	0.000505715	0.0224824	0	0	0	0	1
BN_incidence	0.00295283	0.0658871	0	0	0	0	3
BO_incidence	0.00228081	0.0484657	0	0	0	0	2
BP_incidence	0.00188778	0.0685461	0	0	0	0	4
BQ_incidence	0	0	0	0	0	0	0
BR_incidence	0.000416791	0.0204112	0	0	0	0	1
BS_incidence	0.00176831	0.0836996	0	0	0	0	5
n-gram_0	0.0107381	0.103067	0	0	0	0	1
n-gram_1	0.023195	0.150523	0	0	0	0	1
n-gram_2	0.0502165	0.218391	0	0	0	0	1
n-gram_3	0.0579366	0.233624	0	0	0	0	1
n-gram_4	0.12807	0.334168	0	0	0	0	1
n-gram_5	0.146056	0.353162	0	0	0	0	1
n-gram_6	0.260411	0.438859	0	0	0	1	1
n-gram_7	0.37478	0.484066	0	0	0	1	1
n-gram_8	0.451166	0.49761	0	0	0	1	1
n-gram_9	0.46046	0.498434	0	0	0	1	1
n-gram_10	0.57526	0.494304	0	0	1	1	1
n-gram_11	0.568525	0.495282	0	0	1	1	1
n-gram_12	0.418271	0.493275	0	0	0	1	1
n-gram_13	0.584164	0.492866	0	0	1	1	1
n-gram_14	0.506486	0.499958	0	0	1	1	1
n-gram_15	0.451214	0.497614	0	0	0	1	1

feature	mean	std	min	25%	50%	75%	max
BH_incidence	0	0	0	0	0	0	0
BI_incidence	0	0	0	0	0	0	0
BJ_incidence	0	0	0	0	0	0	0
BL_incidence	0	0	0	0	0	0	0
BM_incidence	0	0	0	0	0	0	0
BN_incidence	0	0	0	0	0	0	0
BO_incidence	0	0	0	0	0	0	0
BP_incidence	0	0	0	0	0	0	0
BQ_incidence	0	0	0	0	0	0	0
BR_incidence	0	0	0	0	0	0	0
BS_incidence	0	0	0	0	0	0	0
n-gram_0	0.0162629	0.126485	0	0	0	0	1
n-gram_1	0.00328061	0.0571826	0	0	0	0	1
n-gram_2	0.0152406	0.122509	0	0	0	0	1
n-gram_3	0.0271036	0.162385	0	0	0	0	1
n-gram_4	0.00766652	0.0872224	0	0	0	0	1
n-gram_5	0.0343037	0.182008	0	0	0	0	1
n-gram_6	0.0569376	0.231723	0	0	0	0	1
n-gram_7	0.0448111	0.206889	0	0	0	0	1
n-gram_8	0.0457577	0.208959	0	0	0	0	1
n-gram_9	0.969947	0.170733	0	1	1	1	1
n-gram_10	0.0307871	0.17274	0	0	0	0	1
n-gram_11	0.0417024	0.199908	0	0	0	0	1
n-gram_12	0.940006	0.237476	0	1	1	1	1
n-gram_13	0.964612	0.184759	0	1	1	1	1
n-gram_14	0.0455252	0.208453	0	0	0	0	1
n-gram_15	0.0444778	0.206154	0	0	0	0	1

7.1.4 Data Analysis of the API abuse by sequence split

The several descriptive statistics captured about the experiments training dataset regular API abuse by sequence split, summarizing the split distribution’s central tendency, dispersion and shape are present in Table 7.4.

Table 7.4: Descriptive statistics about the API abuse by sequence split.

feature	mean	std	min	25%	50%	75%	max
nr_total_timeslots	3.41235	2.30391	1	2	3	5	17

Continued on next page

feature	mean	std	min	25%	50%	75%	max
BQ_incidence	0	0	0	0	0	0	0
BR_incidence	0	0	0	0	0	0	0
BS_incidence	0	0	0	0	0	0	0
n-gram_0	0.630948	0.482548	0	0	1	1	1
n-gram_1	0.244813	0.429977	0	0	0	0	1
n-gram_2	0.855179	0.351921	0	1	1	1	1
n-gram_3	0.764808	0.424119	0	1	1	1	1
n-gram_4	0.271368	0.444666	0	0	0	1	1
n-gram_5	0.783871	0.411604	0	1	1	1	1
n-gram_6	0.201092	0.400817	0	0	0	0	1
n-gram_7	0.243154	0.428988	0	0	0	0	1
n-gram_8	0.136462	0.343279	0	0	0	0	1
n-gram_9	0.833417	0.372603	0	1	1	1	1
n-gram_10	0.204143	0.403074	0	0	0	0	1
n-gram_11	0.650386	0.476848	0	0	1	1	1
n-gram_12	0.697211	0.459465	0	0	1	1	1
n-gram_13	0.605778	0.488683	0	0	1	1	1
n-gram_14	0.329579	0.47006	0	0	0	1	1
n-gram_15	0.358995	0.479706	0	0	0	1	1

7.2 MLP Bayesian Optimization Process Trials

The MLP bayesian optimization process thirty-two trials pertaining to the MLP experiment are present in the next page's table 7.5, **lr** represents the learning rate, **epochs** the number of training epochs, **h1** the number of neurons in the hidden layer, **L1** the L1 regularization parameter, **L2** the L2 regularization parameter, **p** the dropout rate; furthermore, it details the training duration and the accuracy in the validation set of each trial in columns **duration** and **validation accuracy** respectively. The three candidate hyperparameter configurations (#7, #23, #28), are displayed in **bold**.

Table 7.5: Bayesian optimization process trials.

Trial #	lr	epochs	h1	L1	L2	p	duration	validation accuracy
1	0.1	32	512	0	0	0.3	7m47s	99.99%
2	0.05	64	384	0.05	0.001	0.4	15m45s	99.81%
3	0.1	32	128	0	0	0.3	7m:46s	99.99%
4	0.1	32	512	0	0.05	0.2	7m59s	98.16%
5	0.1	32	128	0	0	0.5	7m50s	99.99%
6	0.1	128	512	0	0	0.5	31m:00s	99.99%
7	0.01	128	64	0	0	0.3	31m:21s	99.99%
8	0.01	32	512	0	0	0.5	8m:00s	99.99%
9	0.1	128	64	0.05	0	0.5	31m:26s	99.93%
10	0.01	32	64	0.05	0	0.2	7m:57s	99.98%
11	0.1	32	512	0.05	0	0.5	8m:3s	99.94%
12	0.01	128	512	0.05	0	0.5	31m:48s	99.96%
13	0.01	32	64	0.05	0	0.5	7m:58s	99.98%
14	0.1	128	64	0.05	0	0.2	32m:08s	99.96%
15	0.01	32	64	0	0	0.3	7m:48s	99.99%
16	0.1	32	512	0	0	0.5	7m:55s	99.99%
17	0.1	32	64	0.05	0	0.3	8m:05s	99.91%
18	0.01	128	512	0.05	0	0.2	32m:24s	99.97%
19	0.1	96	256	0	0	0.3	23m:38s	99.99%
20	0.01	128	512	0.05	0	0.2	32m:10s	99.97%
21	0.01	128	512	0	0	0.4	31m:13s	99.99%
22	0.01	32	512	0.05	0	0.3	8m:03s	99.99%
23	0.01	96	64	0	0	0.5	23m:15s	99.99%
24	0.05	32	384	0.001	0	0.4	7m:55s	99.99%
25	0.1	128	64	0	0	0.5	31m:30s	99.99%
26	0.1	32	320	0	0	0.4	7m:57s	99.99%
27	0.1	128	512	0.05	0	0.3	32m:10s	99.95%
28	0.05	64	512	0	0	0.4	15m:39s	99.99%
29	0.1	128	64	0	0	0.2	31m:24s	99.99%
30	0.1	64	64	0	0	0.3	15m:34s	99.99%
31	0.01	32	512	0.05	0	0.5	8m:01s	99.97%
32	0.1	64	512	0	0	0.4	15m:44s	99.99%

7.3 CNN History in regards to the ILSVRC

While the ILSVRC competition has been running since 2010, the 2010 and 2011 winners used SVM and Fisher Vectors, respectively [100], [101]. These solutions were not experimented with because they did not make use of NNs; furthermore, their results were very much eclipsed by the 2012 CNN solution.

The ILSVRC 2012 winner, dubbed AlexNet [95], won the competition by a significant 10% margin, being a pioneer in deep NN architecture and achieving success in many other image recognition tasks, unlike the historically significant LeNet-5 [94] that only performed in the MNIST dataset.

In 2013 the creators of what would become the winning CNN architecture of that year's ILSVRC competition developed a solution that enabled them to visualize a CNN in action, named "convnet" [79]. Using this solution, they restructured the AlexNet architecture to better suit image recognition problems and created ZfNet, also known as Clarifai [79].

Before 2014, CNN architecture's primary approach was the stacking of convolutional and pooling layers, followed by fully connected layers. GoogLeNet [96], also known as InceptionV1, the winner of the 2014 ILSVRC, elaborated on this concept by creating parallel processing branches with different characteristics, which merge to form the complete network architecture.