

Bachelor's Degree in Artificial Intelligence

Detection of DGA domain names by comparing different approaches

Author:
Lorenzo Uttini

Supervisors:
Prof. Marco Frasca, Unimi
Dr. Giordano Colò, Cy4gate

Academic Year 2023/2024

Bachelor in Artificial Intelligence
Pavia 2024

Bachelor in Artificial Intelligence

Lorenzo Uttini

Detection of DGA domain names by comparing different approaches

Academic Year 2023/2024

Bachelor in Artificial Intelligence
Pavia 2024

Abstract

Over the past decade, malware has constituted a significant cybersecurity threat. When malware infects a machine, it attempts to communicate with a Command and Control (C&C) server, creating a botnet controlled by a botmaster. Cyber defenders try to block this communication by identifying the C&C server's IP address or domain name. To evade these measures, botmasters use Domain Generation Algorithms (DGAs) to create numerous pseudorandom domains, making it difficult to block all potential communication channels.

Research shows that Machine Learning (ML) algorithms are crucial in detecting DGA domains. We developed and tested four ML models of varying complexity: Random Forest (RF), Multilayer Perceptron (MLP), Bidirectional Long Short-Term Memory with Convolutional Neural Network (BiLSTM-CNN), and Transformer. These models are combined with three data preprocessing approaches, resulting in seven different classifiers.

Our classification task involves binary (legit vs dga) and multiclass (nine different classes) categorization. In the binary task, several models, including MLP, BiLSTM-CNN, and Transformer, achieve excellent performance with F1 scores exceeding 0.99. The multiclass task obtains macro-F1 and average-F1 scores between 0.7-0.8, with the Transformer model combined with Embedding preprocessing outperforming other approaches.

To demonstrate a complete ML project lifecycle, we develop a web service allowing users to input any domain and receive a prediction of its class (legit or dga). Testing this service with 200 new domains from various sources, the Transformer architecture again demonstrates superior performance, correctly identifying 167 out of 200 domains.

Overall, in this thesis work we further explore the detection of DGA generated domains by comparing different ML methods. Moreover, we deploy the trained models for a potential future application.

Contents

Chapter 1
Introduction

Accessing the web is now a popular activity, performed by people at all times of the day. The internet is an essential resource used across various fields and for numerous operations. Over the past few decades, the expansion of websites, applications, social networks, and other internet-based platforms has resulted in a significant amount of private data being transferred to the digital world. In response, institutional organizations have recently made efforts to regulate and protect the personal data collected by companies on the web [33].

However, this data is susceptible to theft by cyber attackers. Cybercrime has become widespread throughout the internet revolution and a solid ecosystem has been developed where botnets (i.e., networks of compromised computers) play an important role, being used for malicious online activities such as spamming, espionage, and, more importantly, data theft.

When a malware infects a machine, it attempts to communicate with a specific command and control (C&C) server, creating a network of infected machines, or botnet, controlled by a botmaster [39]. Once the C&C is taken out, the botnet owner will lose the control over the whole botnet. Various techniques aim to block the communication between the C&C server and the bot by identifying the IP addresses or domain names of the C&C servers of a botnet. By doing so, the bots will not be able to receive any instructions from their masters.

1.1 Challenge

A common approach to blocking network traffic towards C&C servers involves implementing a blacklist containing the names of identified C&Cs. However, this strategy is static and requires constant updating whenever new malicious domains are discovered, making it inefficient and resource intensive [23]. Consequently, cyber attackers have started using Domain Generation Algorithms (DGAs), which generate numerous pseudorandom domain names, known as Algorithmically Generated Domains (AGDs), to facilitate communication with the outside. These algorithms dynamically generate new domains based on a continuously changing parameter called seed that can be based on various factors such as the current date and time, predefined lists of words, or even the titles of trending social media posts. To facilitate the general understanding throughout the project, we will call AGDs simply as DGA domains.

The challenge with DGAs is that only a few generated domains are active at any given time, making it difficult and expensive to block thousands of potential domains before identifying the specific one used by the botmaster. This dynamic nature of DGAs complicates efforts to efficiently block malicious communications.

1.2 Goal

Efficient techniques to oppose to these types of malware often rely on anomaly detection in network traffic. In recent years, the use of Machine Learning (ML) algorithms to recognize patterns associated with domain names has become increasingly prevalent [2]. This approach helps defenders automatically identify malicious domains generated by DGAs.

This thesis aims to compare various ML models designed to detect DGA domains. We performed the detection in two ways: binary classification, where a domain is classified as either DGA or non-DGA, and multiclass classification, where each domain is associated with a specific subclass of DGA malware. We implemented different types of model such as Random Forest, Multi-Layer Perceptron, Bidirectional Long Short-Term Memory with Convolution Neural Network and Transformers. Then, we combined these models with three data preprocessing methods. Finally, we discussed and compared the results using various metrics.

The project is concluded with a practical simulation of a deployed model, phase known as ML Operations (MLOps). MLOps encompasses the practices and tools necessary to deploy, monitor, and maintain ML models in production. This production phase ensures that the models can operate reliably in a live environment, adapting to new data and evolving threats.

The work is conducted in collaboration with Cy4gate, an italian cybersecurity company, with the intention of future real-world application. Intersecting the modern fields of AI and Cybersecurity represents a primary goal for any application that involves web and tech security. This thesis aims to advance towards this objective.

1.3 Structure

The thesis has the following structure:

Chapter 2 focuses on the exploration of the technical background necessary to fully understand this work.

Chapter 3 provides a detailed description of the ML algorithms used and the different approaches employed for data preprocessing;

Chapter 4 presents the results obtained and includes a brief discussion of these findings;

Chapter 5 explains how the model can be implemented in production, including a small simulation to demonstrate its application;

Chapter 6 concludes the thesis, summarizing the key findings and discussing potential future work.

Chapter 2
Technical Background

In this chapter we will cover the preliminary concepts in order to completely understand DGA and its issues.

Starting from the malware, that represent the fundamentals of any cyber threat, it is crucial to explain what the components of a botnet are and how DGA are formed.

Finally, we introduce Machine Learning (ML), including the description of its subfields and how a ML project is carried out.

2.1 Malware

A malware, short for malicious software, is defined as any software intentionally designed to cause damage to a computer, server, client, or computer network. There are several types of malware [14]:

- **Viruses**: Programs that attach themselves to legitimate files and spread by infecting other files.
- **Worms**: Standalone programs that replicate themselves to spread to other computers, often through network connections.
- **Trojans**: Malicious software disguised as legitimate software. They do not replicate themselves but can cause harm by allowing unauthorized access to the user's system.
- **Ransomware**: Software that encrypts files on a device, demanding a ransom for their decryption.
- **Spyware**: Software that secretly monitors user activity and collects information without consent.
- **Rootkits**: Software designed to grant unauthorized access to a computer while hiding its presence.

Malware can spread through various means, such as email attachments, infected websites, removable drives, or vulnerabilities in software. Its impact can range from minor annoyances, like displaying unwanted ads, to severe consequences, such as data theft, financial loss, and system damage.

2.2 Botnets and DGA

DGA malware are employed by a botnet as a key technique for maintaining communication with its command and control (C&C) servers. DGAs enable the botnet to dynamically generate pseudorandom domain names, making it more difficult for defenders to block communication by blacklisting static domain names. This dynamic nature of DGAs helps the botnet remain resilient and continue receiving instructions from the botmaster despite efforts to disrupt its operations.

Botnets and DGAs are described in details in the next sections.

2.2.1 Botnet Structure

A botnet, short for "robot network", is a group of internet-connected devices, as computers or smartphone, that have been compromised and controlled by a malicious actor known as botmaster. A compromised device, referred to as a bot, becomes part of a botnet when it is infected by a malware. The individual who controls the botnet can manage the actions of these compromised devices through communication channels established using standard network protocols like IRC (Internet Relay Chat) and HTTP (Hypertext Transfer Protocol).

Cyber criminals can employ a botnet to perform a variety of harmful activities [5], including:

- *Distributed Denial of Service (DDoS) Attacks*: Overloading a target with traffic to make it unavailable.
- *Spam Campaigns*: Sending massive amounts of unsolicited emails.
- *Data Theft*: Stealing sensitive information such as personal data, financial information, or login credentials.
- *Click Fraud*: Generating fake clicks on advertisements.
- *Cryptomining*: Using the bots resources to mine cryptocurrencies without the owners consent.

As mentioned previously, the botmaster execute commands to botnet through a specific server known as the C&C server. Keeping in consideration different factors, the botnet can be implemented in different ways.

Centralized Topology In this type of topology, the botnet is structured in a client-server model where each bot relies on one central C&C server. Although this setup ensures reliable and low-latency communication, it also makes the botnet relatively easy to disable; taking the single C&C server offline will render the entire botnet inoperable.

Over the years, different structures of a centralized topology have been developed. The first and simplest is the *Star Topology*, where each bot depends directly on the C&C server [17]. While this is highly efficient from a performance standpoint, it is also very vulnerable; if the C&C server is blocked (single point of failure), the entire botnet collapses.

To address the single point of failure issue, other centralized topologies have been tested. For example, the *Extended Star Topology* is slightly differs by adding redundant C&C servers. Another variation is the *Hierarchical Topology*, which employs some bots as proxies to obscure the botnet's structure.

Nevertheless, each centralized topology shares a common problem: dependency on a single entity. Therefore, cyber attackers have developed topologies where bots do not rely on a single master, known as Decentralized Topologies.

Decentralized Topology The Decentralized Topology, or Peer-to-Peer (P2P) model, is designed so that each node is only aware of its immediate neighbours, completely ignoring all other nodes [21]. In this setup, each node in the network functions both as a client and a server, allowing a command to enter at any point in the network and propagate along entirely random routes.

Additionally, if a node is compromised, it does not affect the overall functioning of the network, which will continue to expand by recruiting new bots. Recruitment occurs by sequentially querying a series of pseudo randomly generated IP addresses. If the request is successful, the queried machine responds with its software version and a list of other known bots. If one of the two bots in contact has an older version, it is updated. This process enables each bot to expand its list of known machines, thereby increasing the size of the botnet.

This approach has been widely discussed in the literature, highlighting both its strengths and weaknesses. For example, some studies emphasized the resilience of P2P networks against node failures [31]. Similarly, it has been shown how the decentralized nature of P2P systems can lead to robust and scalable networks [10]. However, the complexity of maintaining such systems, especially in terms of secure communication and synchronization, has been noted as a considerable challenge [27].

2.2.2 Rallying Mechanism

The life cycle of a botnet, from the vulnerability's attack to maintaining the botnet update, is described by different steps [22].

The first phase, called *Propagation*, involves identifying and leveraging vulnerabilities in target system to gain unauthorized access. Based on the degree of (human) user intervention required, propagation mechanisms are classified as Active and Passive.

Then, the botnet enters in the most important phase, known as *Rallying*. Here, newly infected bots find and connect to their C&C servers, marking their formal registration within the botnet. Bots can use hardcoded IP addresses or domain names embedded in the bot binary to establish this connection. Static hardcoded IP addresses eliminate the need for the Domain Name System (DNS) but can be exposed through reverse engineering. Alternatively, hardcoded domain names provide more flexibility, as the botmaster can redirect to new IP addresses if the original domains are taken down.

More advanced botnets use DGAs to dynamically generate domain names, making it challenging for authorities to shut down the botnet, as it can quickly move to new domains. Peer-to-peer botnets often use seeding, where bots are given an initial list of active peers to connect with, maintaining decentralized communication and reducing dependence on centralized servers. The technical

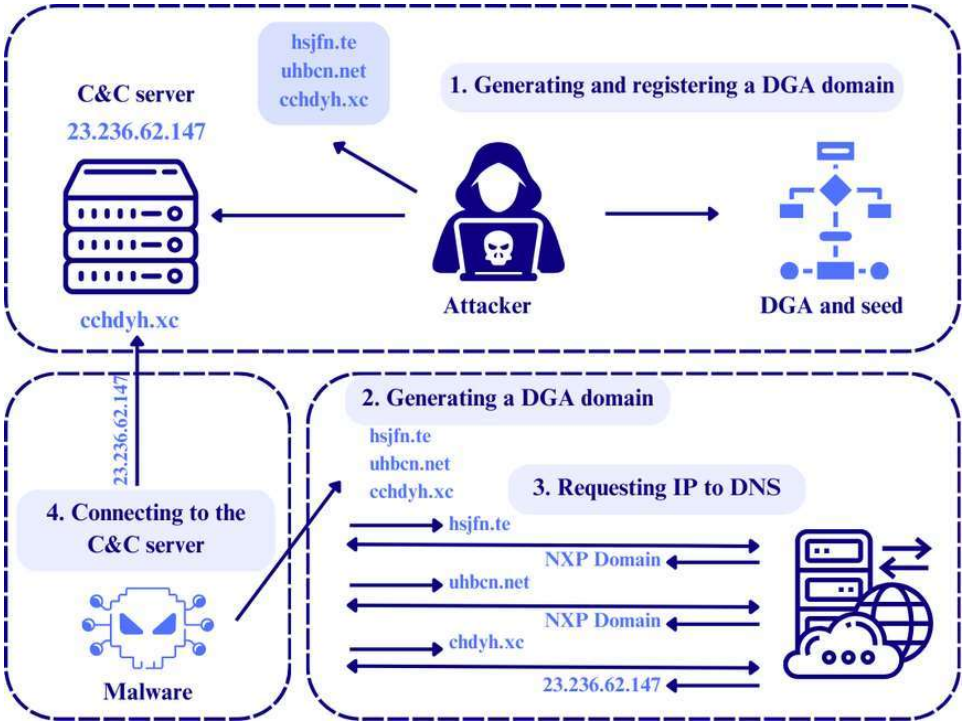


Figure 2.1: DGA attacker procedure. Image adapted [29].

process starts with both the bot and the C&C server running an algorithm with the same seed, which could be based on factors like current time or trending topics. This generates an identical list of domain names on both ends. The server then randomly selects and registers one or more of these domains with a DNS service, linking them to its IP address. The bot queries each domain from the list until it finds a match, allowing it to connect to the C&C server using the resolved IP address. Figure 2.1 shows the attacker process in details.

This combination of methods ensures robust and resilient communication channels for the botnet, complicating efforts to block it.

Finally, there is the *Purpose* phase, where the botmaster manages numerous bots to use their collective power for malicious activities. Additionally, in the *Update* phase each bot receives updates to evade detection by defenders.

2.2.3 DGA

During the Rallying phase, the most efficient way for cyber attackers in order to establish a communication between C&C server and bots is based on DGA malwares.

The main goal of a botmaster is employing DGA to generate a large number of pseudorandom domains, called AGDs or DGA domains, through a mutual unpredictable seed between the bot and server. Out of these domains, only few are registered by the botmaster to point to the actual C&C server. Bots query these domains until one resolves successfully, providing the IP address of the C&C server. This process is repeated periodically, with new domains generated daily, changing the rendezvous domain regularly [6]. For example, a bot may try domains like

ttrnsn.tiz and *yhhncd.biz* without success but eventually resolve *osssinn.info* to get the IP address `131.175.17.35` where the server is hosted.

There are different types and families of DGAs [8] that can be distinguished by:

- the **Relation on time**:
  - *Time-Independent* or Static seed, examples in table 2.1
  - *Time-Dependent* or Dynamic seed, examples in table 2.2
- the **Causality**:
  - *Deterministic*
  - *Non-Deterministic*

DGA family	Domain example
Necurs	qujfnn.to
Dridex	clientalalaxp.mn
Pushdo	jnhutiiv.tv
Srizbi	dfheup.ru
Torpig	xrtykfq.com
Conficker	mbxsrvx.com
Kraken	ujhbqwe.net

Table 2.1: Examples of DGA domains time-dependent

DGA family	Domain example
Zeus	jlkdshsa.com
CryptoLocker	aewqhrq.com
Ramnit	ldjsakj.net
Banjori	mnfdklv.com
Gozi	vbmlkjh.com
Gootkit	ytrewqk.info
Virut	xcvzdfs.net

Table 2.2: Examples of DGA domains time-independent

These DGAs families are composed by specific algorithms, and, once the process is known, everyone might generate these malign domains [7]. Furthermore, it is possible to divide DGAs by its *Generation scheme*:

- *Wordlist-based DGA* generate domain names by combining words from a predefined list.
- *Hash-based DGA* create domain names using hash-functions, which take an input such as a date and produce a fixed-size string of characters.
- *Permutation-based DGA* generate domains by permuting or rearranging parts of a base string or a set of characters.

2.3 Machine Learning

With the rapid advancements in Large Language Models (LLMs) (e.g., GPT [13]), Artificial Intelligence (AI) is experiencing a significant technological evolution. In recent years, there has been an incredible increase in the development of more sophisticated AI models and architectures, such as transformers and neural networks, which have enhanced the capabilities of AI systems. These advancements are supported by improvements in hardware, including the development of specialized AI processors and increased computational power, allowing for more complex and efficient AI models. Furthermore, AI is becoming increasingly integrated into everyday applications, ranging from natural language processing in virtual assistants to advanced computer vision systems in autonomous vehicles.

Machine Learning (ML) is the main field of AI, and it represents the characterized discipline. Through some ML models, it is possible to predict, classify and generate data. In our project, we employed ML to detect whether a domain name is generated by DGA or not.

2.3.1 Overview of ML

ML focuses on the development of algorithms capable of learning from data and making predictions or decisions without being explicitly programmed for specific tasks. In essence, ML systems improve their performance over time as they exposed to more data, making them adaptable to various problems [11].

As its core, ML involves training models using data to discover patterns, generalizing from the training data to new data. This process aims to perform tasks such as classification, regression, clustering and more. The main principles of ML can be resumed in:

- *Data*: high-quality data is essential for training effective models.
- *Algorithms*: the mathematical phase that process data to identify patterns and make predictions.
- *Training*: models are fed with data and adjust its parameters to minimize errors.
- *Evaluation*: assessing the model performance.
- *Iteration*: the process requires continuous refinement and improvement.

Through the years the field of ML has been experienced the development of different subfields.

Classic ML It involves several key components, starting with the input data, which comprises features extracted from raw data. Here, the feature extraction is a critical part as the features represent the information the model uses to learn and make predictions. In the training phase, labelled data (i.e., data where the output is known) is used to teach the model to understand the relationship between input features and the desired output. This process involves algorithms such as Linear Regression for continuous value prediction, Logistic Regression for binary classification, Decision Trees and Random Forest for making decisions based on feature splits and Support Vector Machines for finding optimal boundaries between classes.

Deep Learning Moving from classical ML to more advanced techniques, Deep Learning (DL) is a subset of ML that utilizes neural networks with many layers to learn from data. Unlike classical ML, which often requires manual feature extraction, DL models can automatically learn hierarchical features from raw data. This capability is particularly powerful for tasks involving complex patterns, such as image and speech recognition. These neural networks consist of interconnected nodes that process input data through layers, allowing the model to learn patterns and representations [24]. Key algorithms in DL include convolutional neural networks (CNNs) for image data, recurrent neural networks (RNNs) for sequential data, and transformer networks for processing different types of data.

Natural Language Processing This subfield represents an intersection between AI, Computer Science and Linguistic. It enables computers to understand, interpret, and generate human language and involves tasks such as translation, sentiment analysis, and question-answering. A significant advancement in Natural Language Processing (NLP) has been the development of generative AI and LLMs. Generative AI models are capable of producing human-like text based on prompts, facilitating applications as content creation and conversational agents. LLMs are trained on massive datasets to understand and generate human language, employing architectures like transformers to achieve high performance across different tasks.

2.3.2 Pipeline for a ML project

As we propose a comparison between different ML models for a potential use in cybersecurity operations, each ML project involves in different steps. Defining and adhering to these phases is the key for the success of the project [20].

1. **Problem Definition**: The first step is to clearly define the problem you want to solve. This involves understanding the business or research question, setting goals, and determining the type of ML task.

2. **Data Collection**: Gathering the data necessary to address the problem is one of the most important phases of the process. Data is a precious resource, especially for projects directed towards production, and it can come from various sources such as databases, web scraping, sensors, or public datasets. The quality and quantity of data are critical as they directly impact the model's performance.

3. **Data Exploration and Cleaning**: Explore the dataset to understand its structure and contents. This step involves describing features with descriptive statistics, visualizing plots to identify patterns, and cleaning the data, including handling missing values and duplicates.

4. **Data Preprocessing**: Transform the raw data into a format suitable for modeling. Key preprocessing steps include feature scaling, encoding categorical variables, and splitting the data into training, validation, and test sets.

5. **Feature Engineering**: Create new features or modify existing ones to improve the model's performance. This is a crucial step, especially for classic ML.

6. **Model Selection**: Choose the appropriate ML algorithm based on the problem and data characteristics.

7. **Model Training**: Train the selected models on the training dataset. This involves feeding the data into the algorithms, adjusting parameters (or weights), and optimizing them to improve performance.

8. **Model Evaluation**: Assess the performance of the trained models using the validation set and finally the test set. The most popular evaluation metrics are Accuracy, Precision, Recall, and F1 Score.

9. **Model Deployment**: Deploy the model to a production environment where it can make real time predictions. This includes integrating the model into an application and setting up monitoring and logging to track the model's performance over time.

The Figure 2.2 show the complete process of a ML project.

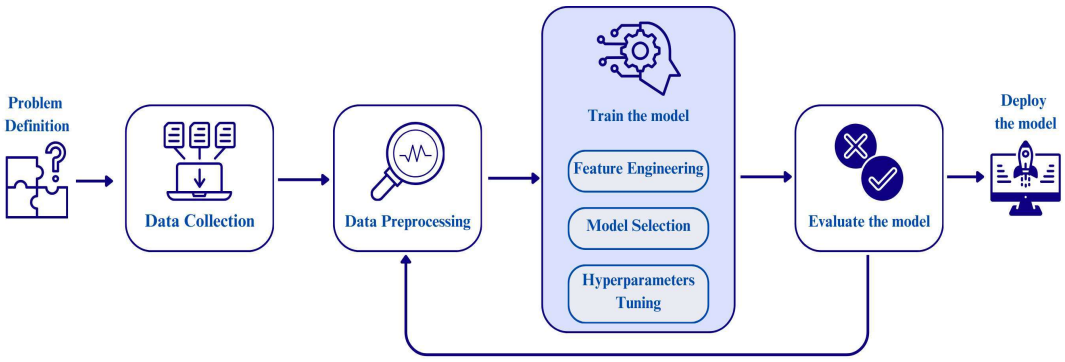


Figure 2.2: The ML life cycle

Depending on context and environments conditions about the developing of a ML project, some of the phases cited above might not be considered. However, especially for students at the beginning stages of ML, it is good habits to check that each step is respected.

2.3.3 Research and Production

In the world of ML there are differences between working towards a model with research objectives and developing a model with real-world practical applications. This thesis is mainly conducted as a research project; nevertheless, as we collaborated with Cy4gate, we decided to perform a toy simulation for a possible feature application (Chapter 5).

There are five main differences, shown in table 2.3, between ML in research and production [20].

Requirements In the research field, when carrying out a project, the most common goal is to obtain an excellent performance on benchmark datasets. This objective often requires techniques that makes the model really complex to use. Contrarily, Developing a ML project for production involves different stakeholders aiming for different objectives.

For instance, considering a fitness app that provides personalized workout plans to users and generates revenue through premium subscriptions. Each stakeholder has its own goal:

- *Data Scientist* prefer a model that accurately predicts users' fitness goals to create personalized workout plans. They believe they can improve accuracy by incorporating more diverse datasets.
- *Marketing Team* wants a model that promotes premium subscriptions. They prioritize features that showcase success stories to drive subscriptions.
- *Product Team* observes that any increase in app complexity or load time leads to a drop in user engagement. They aim for a model that can generate and display workout plans in under 200 milliseconds.
- *Manager* wants to maximize user retention and revenue. They are considering whether to invest in further developing or to explore other revenue options.

Thus, each team has different priorities and goals, highlighting the need for coordination and decision-making. To develop a ML project for production is fundamental to consider a wide range of factors, depending on the final goals.

	Research	**Production**
Requirements	State-of-the-art model performance on benchmark datasets	Different stakeholders have different requirements
Computational priorities	Fast training, high throughput	Fast inference, low latency
Data	Static	Constantly shifting
Fairness	Often not a focus	Must be considered
Interpretability	Often a focus	Must be considered

Table 2.3: Differences between ML in research and production

Computational Priorities When designing an ML system, it's common for people to focus too much on model development and not enough on deployment and maintenance. During development, the focus is on training models, which involves multiple passes over large training datasets and generating predictions on smaller validation datasets. Here, training is the bottleneck. However, in deployment, the model's primary task is to generate predictions, making inference becomes the bottleneck.

In research, the priority is fast training and *high throughput* (processing many samples per second). In production, the priority shifts to fast inference and *low latency* (quickly returning results for each query). For instance, if processing one query takes 10 ms, the throughput is 100 queries/second. However, batching queries can increase both latency and throughput. Processing 10 queries at a time in 10 ms results in 1,000 queries/second, and processing 50 queries in 20 ms yields 2,500 queries/second.

Higher latency can negatively impact user experience and conversion rates, by causing, for instance, users to leave web pages if they load slowly. Therefore, reducing latency often involves processing fewer queries simultaneously, which can use less hardware and increase processing costs per query.

Data Data is often one of the most underrated aspects when developing an ML project for research. Typically, researchers use publicly available datasets that have already been employed by others. These datasets are generally structured, labeled, static, and have minimal missing values. Consequently, in research, data collection is one of the easiest steps, allowing researchers to focus on developing and evaluating the model.

In contrast, developing an ML model for production presents a different scenario. Public datasets are often insufficient, necessitating the collection of new data. This data is usually unstructured, biased, and imbalanced. Additionally, production data must often comply with privacy regulations and is continuously generated from users, making it dynamic and challenging to manage.

Figure 2.3 illustrates a humorous yet perceptive graph by Andrej Karpathy (Director of AI at Tesla in 2020) about the differences between working on an ML project during a PhD and at Tesla. Although humorous, it reflects the real challenges faced in production environments.

Fairness During the research phase of ML development, fairness is often deprioritized, with researchers focusing on achieving state-of-the-art performance first and planning to address fairness later. However, this approach is problematic because, by the time the model reaches production, it is often too late to correct biases. While accuracy and latency are commonly optimized metrics, there is no established standard for fairness metrics, making it difficult to deal with biases effectively.

ML algorithms often present historical biases in the data, leading to large-scale discrimination. While a person can make biased decisions affecting some individuals, an ML algorithm can do so for millions in seconds, impacting minority individuals. Improving model predictions for underrepresented groups can be costly, and many companies may choose not to address these issues.

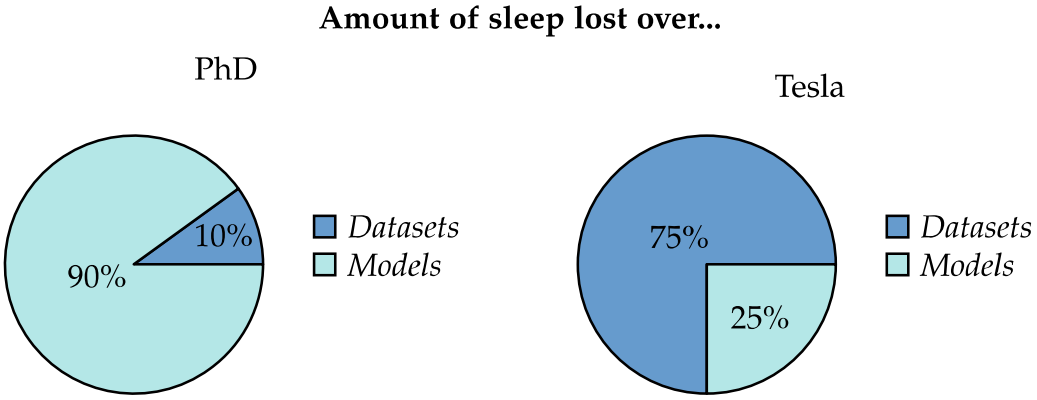


Figure 2.3: Research versus Production. Adapted from an image by Andrej Karpathy.

Explanability The workings of a ML system and the factors influencing its decisions have become really popular topics in recent years. For instance, surveys indicate that, on average, people prefer to undergo surgery performed by a human with a lower success rate rather than by an AI system, perceived as a *"black box"*, even if it has a higher success rate.

A growing field addressing this concern is Explainable Artificial Intelligence (XAI), which tries to develop ML systems that can justify their decisions. While ML research just focuses on performance without prioritizing interpretability, interpretability is a crucial requirement in the industry. Being able to explain how a system makes certain decisions is fundamental for both users and business leaders.

Overall, research in ML is essential, especially within academic environments, to develop and build new models and architectures. However, few companies can afford an AI research department because they prefer to have products ready to generate revenue rather than invest in the research market.

As research in ML becomes more accessible, a large number of companies will require products and applications for users, necessitating ML in production. Consequently, most positions in the ML field will demand production skills.

Chapter 3
Data and Model Descriptions

Process the data, as described in Section 2.3.2 represents one of the crucial step during a ML project. Since the raw data that we utilized are formed by strings of characters (i.e., domains names), there are different approaches that we can try to preprocess the data. In Section 3.2 three strategies are proposed.

As consequential step after data preprocessing, there is the development of the models. The literature proposed several methods on how to detect DGA malwares [45]. We decided to compare four different models and architecture, some of them slightly modified from the originals, to perform our tasks. The structure of the implemented models is given in Section 3.1

3.1 Models

The choice of the model for a ML project is a crucial step to ensure the successful completion of the final task. There are various types of ML models, each one appropriate to different kinds of tasks. For instance, if the goal is to perform classification, as in our case, the models used are known as classifiers. Given preprocessed data, these classifiers are trained on the data and subsequently used to make accurate predictions on new, unseen data.

As already mentioned in Section 2.3, throughout the last century several ML algorithms, some of which are derived from statistics, have been implemented to perform classifications. Each algorithm has various characteristics that make it more or less appropriate for specific tasks. Regarding DGA detection, previous works have explored a wide range of different algorithms and architectures [9, 43, 1, 42, 35]. Moreover, some studies propose comparison between different models [45, 41].

The models we choose present different structures and characteristics: *Random Forest*, included in Trees category, *Multi-Layer Perceptron* and *BiLSTM-CNN*, based on Neural Networks, and *Transformer*, a more complex architecture. Furthermore, we directly associated to each classifier one or more preprocessing data methods.

3.1.1 Random Forest

Random Forest is an ensemble learning method for classification and regression, which was introduced by Breiman in 2001 [12]. It is based on the concept of decision

trees and it exploits the power of multiple trees to create a more robust and accurate model.

The architecture of a Random Forest consists of a collection of decision trees, each trained on a random subset of the data and features. The predictions from all the individual trees are aggregated to produce the final output. This approach helps to reduce overfitting and improves the generalization of the model.

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ be the training dataset, where x_i represents the feature vector and y_i is the target label. The Random Forest algorithm can be described in the following steps:

1. **Bootstrap Sampling**: Generate B bootstrap samples $\mathcal{D}_b$ from the original dataset $\mathcal{D}$.
2. **Train Decision Trees**: For each bootstrap sample $\mathcal{D}_b$, train a decision tree T_b using the following modifications:
   - At each node, randomly select a subset of m features from the total M features.
   - Split the node using the feature that provides the best split according to a predefined criterion (e.g., Gini impurity or information gain).
3. **Aggregation**: For classification, aggregate the predictions of all trees T_b by majority voting.

Random Forest can be applied both for classification and regression tasks. The prediction for a new instance x in a classification task can be expressed in mathematical terms as follows:

$$\hat{y} = \text{mode}\{T_b(x) \mid b = 1, 2, \ldots, B\}$$

where $T_b(x)$ is the prediction of the b-th tree for input x, and mode represents the majority vote.

A simple structure, used also for this work, of the Random Forest model is shown in Figure 3.1. ScikitLearn library provides an implementation for Random Forest that includes a large range of hyperparameter choices. Among them, the number of decision trees T_b (n) and the number of features at each node (m) will be discussed in Section 4.1.1.

Random Forest has several properties that make them particularly well-suited for classification tasks:

- **Robustness to Overfitting**: By averaging multiple trees that are trained on different subsets of data, it reduces the risk of overfitting.
- **Handling High Dimensional Data**: Randomly selecting subsets of features at each split allows Random Forests to handle high-dimensional data effectively.
- **Non-linearity**: It can capture non-linear relationships between features and the target variable.
- **Feature Importance**: It provides estimates of feature importance, which can be useful for understanding the underlying data.

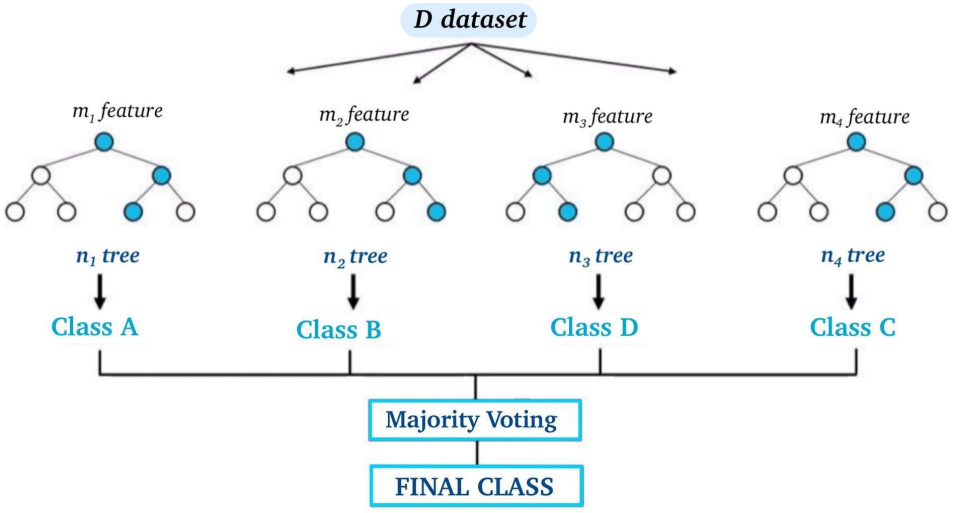


Figure 3.1: Random Forest structure

Random Forest is the most popular algorithm used for DGA detection. This is because it can effectively handle the high-dimensional and complex nature of domain name features, adapt to various patterns in the data. The ensemble approach ensures that the overall model is resilient to the noise and variability causing by DGA detection tasks.

3.1.2 Multi-Layer Perceptron

Multi-Layer Perceptron (MLP) is a class of feedforward artificial neural network (ANN) that has been used in various ML tasks. The concept of MLP dates back to the 1940s, with the development of the perceptron by Rosenblatt [36]. However, the idea of the modern MLP using multiple layers was later developed [3].

MLPs consist of multiple layers of nodes, each fully connected to the nodes in the subsequent layer. The architecture typically includes an input layer, one or more hidden layers, and an output layer. Each node in the network applies a linear transformation to its inputs, followed by a non-linear activation function. Figure 3.2 shows a basic MLP structure.

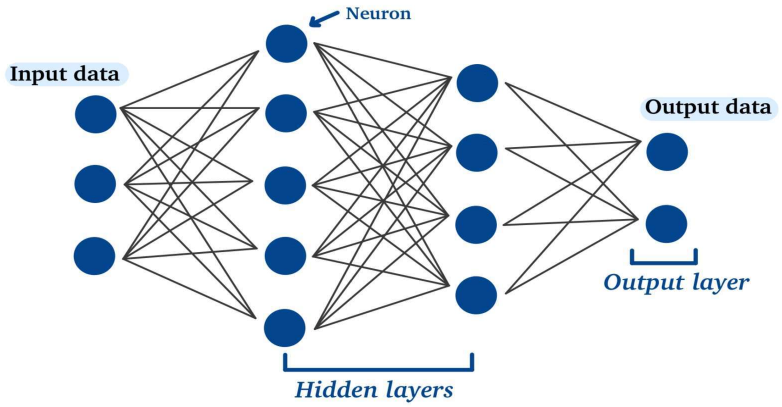


Figure 3.2: MLP structure

The architecture of an MLP can be mathematically described as follows:

- **Input Layer**: Let $x = [x_1, x_2, \ldots, x_n]$ be the input vector.
- **Hidden Layers**: For each hidden layer l, the output $h^{(l)}$ is computed as:

$$h^{(l)} = \sigma(W^{(l)} h^{(l-1)} + b^{(l)})$$

  where $W^{(l)}$ is the weight matrix, $b^{(l)}$ is the bias vector, σ is the activation function, and $h^{(0)} = x$.

- **Output Layer**: The final output y is computed as:

$$y = W^{(L)} h^{(L-1)} + b^{(L)}$$

  where L is the total number of layers.

This section focuses on the mathematical explanation of MLP elements employed in the project. The details about the implementation and parameters used for each MLP model will be treated in Section 4.1.1.

Moreover, MLP functioning can be described by different functions.

Loss Function It quantifies the difference between the predicted output and the actual target value. For the multiclass (binary) classification task, the *categorical (binary) cross-entropy* loss is used:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} [p_i \log(q_i)]$$

where N is the number of samples, p_i is the true label, and q_i is the predicted probability.

Activation Function It introduces non-linearity into the model, allowing it to learn complex patterns. The three activation functions applied to the models include the *Sigmoid* in the final layer of the binary task,

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

the *Softmax* in the final layer of the multiclass task,

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

and the *ReLU (Rectified Linear Unit)* in the hidden layers,

$$\sigma(x) = \max(0, x)$$

Optimizer It updates the model parameters to minimize the loss function. A common choice is the *Adam* optimizer, which combines the advantages of two other extensions of stochastic gradient descent:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where θ_t are the parameters at step t, η is the learning rate, $\hat{m}_t$ and $\hat{v}_t$ are the estimates of the first and second moments of the gradients, and ϵ is a small constant to prevent division by zero.

This simple Neural Network can be efficiently adapted for different tasks. It can automatically learn and extract relevant features from raw input data, which is beneficial for detecting complex patterns in domain names. Moreover, the non-linear activation functions enable it to model complex relationships between input features and the target variable. For these reasons, MLPs are a versatile model that has been employed, especially at the beginning, for DGA domains detection task.

3.1.3 BiLSTM-CNN

Classifying DGA domains with Recurrent Neural Networks (RNN) and Convolutional Neural Network (CNN) has became really popular due to their high performances obtained [42, 47, 35, 44]. The BiLSTM-CNN model was proposed by a research on neural networks methods for DGA domains detection [29]. The architecture is based on two neural networks combined together to form an ensemble model which improves the performances. This section explains each part of the model and how we adapted it to our task.

CNN A Convolutional Neural Network (CNN) is a deep learning model known for its excellent performance in image recognition by exploiting local region features [25]. It consists of a feature extraction network with convolutional and pooling layers, and a classification network with fully connected layers. CNNs can also process one-dimensional (1D) sequence data, known as 1D-CNNs, which are useful for tasks such as sensor data processing and natural language processing.

For DGA domains classification, the original model applied four 1D-convolutional layers with kernel sizes from 2 to 5, each with n filters, followed by max pooling layers. We decided to keep only two 1D-CNN layers with its respective pooling layers. In the classification step, the outputs of these layers are concatenated and passed through three hidden layers, each consisting of a fully connected layer, an exponential linear unit (ELU) activation function, batch normalization, and dropout to prevent overfitting. This approach helps the model to learn robust features from sequential data for effective DGA detection.

BiLSTM Long Short-Term Memory (LSTM) networks are a type of RNN that excel at learning from sequential data due to their ability to retain information over long periods [19]. An LSTM network updates its hidden state based on the current input and the previous hidden state, which can be described as:

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = W_{hy} h_t + b_y$$

where, h_t represents the hidden state at time t, x and y are the input and output, respectively, and W and b are the weights and biases to be learned.

However, LSTMs can be limited by their unidirectional nature, processing information only in a forward direction. To address this, Bidirectional LSTMs (BiLSTMs) are employed, which process information in both forward and backward directions:

$$\overrightarrow{h_t} = \sigma(W_{x\overrightarrow{h}} x_t + W_{\overrightarrow{h}\overrightarrow{h}} \overrightarrow{h}_{t-1} + b_{\overrightarrow{h}})$$
$$\overleftarrow{h_t} = \sigma(W_{x\overleftarrow{h}} x_t + W_{\overleftarrow{h}\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}})$$
$$y_t = W_{\overrightarrow{h}y} \overrightarrow{h_t} + W_{\overleftarrow{h}y} \overleftarrow{h_t} + b_y$$

In these equations, $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ denote the forward and backward hidden states, respectively.

Moreover, the BiLSTM model is improved with an Attention mechanism to focus on important parts of the domain sequences. The detailed explanation of the Attention mechanism is provided in the next model.

For DGA domains detection, BiLSTM networks are particularly advantageous as they can capture patterns in domain name sequences by considering context from both past and future characters. This bidirectional processing helps in better understanding and classifying dynamically generated domains.

CNN-BiLSTM Ensemble Finally, combining CNNs and BiLSTMs with Attention using bagging and stacking methods, it is possible to develop an ensemble approach for our classification task.

Firstly, the input domain is tokenized and embedded character by character (this step is part of data preprocessing in Section 3.2). The model utilizes CNNs to capture local sequence information and BiLSTMs to learn global sequence patterns. In the ensemble, intermediate values from both parts are aggregated and processed through a fully connected layer. Figure 3.3 illustrates the conceptual architecture of the proposed model.

To sum up, the CNN component applies four (two in our case) 1D-convolutional operations with filter sizes ranging from 2 to 5, followed by max pooling, while the BiLSTM component incorporates an attention mechanism to improve its sequence learning capability. Finally, outputs from the CNN and BiLSTM layers are concatenated and fed into a fully connected layer to determine the final DGA classification.

This end-to-end training approach combines the strengths of both CNN and BiLSTM models, effectively integrating their outputs to provide a robust solution for detecting DGA domains.

3.1.4 Hybrid Transformer

Transformers, introduced by Vaswani et al. in "Attention is All You Need" [40], revolutionized natural language processing by exploiting self-attention mechanisms to capture long-range dependencies in sequential data. This architecture has become the foundation for part of the most advanced models in AI, demonstrating

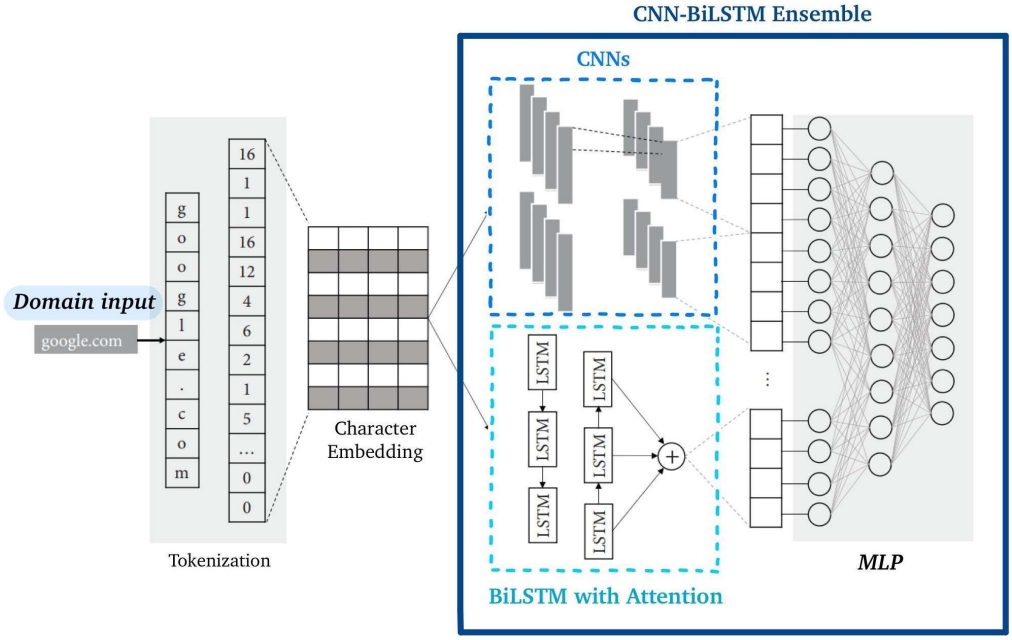


Figure 3.3: CNN-BiLSTM Ensemble architecture. Image adapated from Namgung, 2021 [29].

significant advancements in several tasks involving generating and understanding of human language.

In this work, we implemented a Hybrid Transformer architecture proposed by an innovative research on DGA domains detection in 2023 [16]. The architecture combines character-level and bigram-level embeddings to enhance feature extraction and it consists of two different parts, a modified transformer block and a feed forward network, concatenated together. Each main part of the model is explained in the following paragraphs and the overall architecture is shown in figure 3.4.

Embedding Layer How a classic embedding layer work for preprocessing strings of domains is described in Section 3.2.3. However, the embedding layer in this architecture is more complex and it needs more details. The model comprises two parallel embedding layers: one for character level word vectors and another for bigram level embeddings. The character sequence and bigram sequence of the domain names are generated and padded to a maximum length of L_1 and L_2, respectively, considering also the Vocabulary sizes, V_1 and V_2. Each sequence is embedded into a d-dimensional space, with positional encodings added to retain the sequential information. Finally, the two embeddings are described as:

$$E_1 = W_1 \cdot X_{L_1}$$
$$E_2 = W_2 \cdot X_{L_2}$$

where W_1 and W_2 represent the weighted matrices to be trained in the layer. Both token embeddings and position embeddings are combined to add positional information to each character in the input sequence. In this implementation, positional encodings are represented as a learnable embedding layer, rather than using the fixed sinusoidal functions (cosine and sine) as described in the original paper

[16]. This choice allows the model to learn and update the positional representations during training, potentially improving the encoding of positional information for our task.

Character-part The character embedding is followed by N Transformer blocks that employ an encoder-only architecture for feature extraction in DGA classification. The main component of the transformer block consists in *multi-headed self-attention* layers, employed to calculates the connection between each character (or bigram) in the input domain. The Self-attention and the Multi-head attention are described as:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_h)W^0$$

where Q, K, V represent respectively query, key, value matrices and head_i represents a self-attention component $\mathrm{Attention}(QW^Q, KW^K, VW^V)$.

Contrarily to the original encoder part in a classic transformer architecture [40], here the normalized Multi-head attention part is followed by a 1D-CNN layer. Using a 1D-CNN for this type of task, as already explained in Section 3.1.3, is crucial to detect features at different locations and to automatically discover the implicit patterns in different domain names. Finally, the transformer block ends with a MLP consisting only of one linear transformation. It is fundamental to mention that, as the overall architecture was already enough complex, we choose to utilize just one character block ($N = 1$), differently from the original research where three blocks were employed sequentially.

Bigram-part The bigram embedding is followed by a simpler structure composed just by a 1D-CNN layer and a MLP. Thus, transformer and attention are not used for this part.

Concatenation The final step involves concatenating the features from the character-level and bigram-level sequences. This combined feature vector, with a dimension of d, is passed to a simple linear layer with *softmax* or *sigmoid* as activation functions, depending whether the task is binary or multiclass. As output, we receive the predicted class of the domain. This hybrid approach allows the model to utilize information from different scales, improving its ability to detect DGA domains.

The proposed Hybrid Transformer model is the most advanced architecture that we use in this work. It represents a significant advancement in DGA domains detection, by integrating character-level and bigram-level embeddings and utilizing the powerful Transformer architecture. Nevertheless, given its complexity, the overall model requires an appropriate hardware to be trained. Thus, we have to consider also the trade-off between computational resources and complexity of a model (a better explanation will be given in Section 4.3).

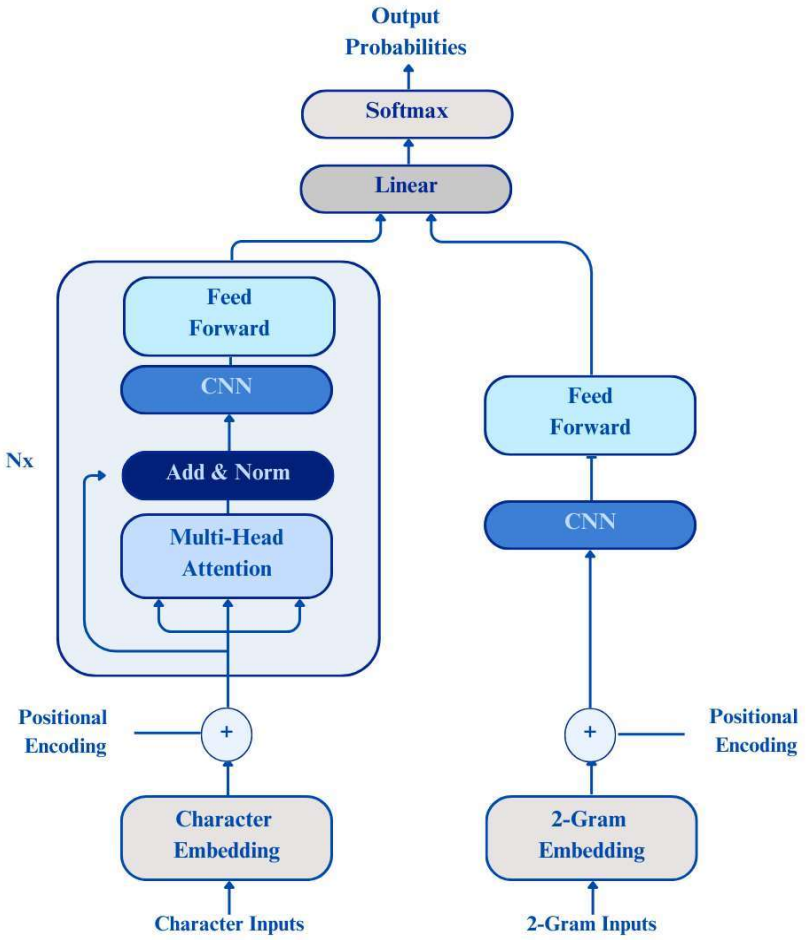


Figure 3.4: Hybrid Transformer architecture. Image adapted from Ding, 2023 [16].

3.2 Data Preprocessing

Machines can't read text or look at images in the same way humans do. To process such data, machines require inputs to be transformed or encoded into numerical representations. This transformation typically involves converting text and images into vectors and matrices, which are structured formats that models can understand and manipulate. By representing inputs as numbers, we enable the training and deployment of machine learning models. This numerical representation is fundamental in various applications, from natural language processing to image recognition, as it allows the models to perform computations and learn patterns from the data.

In the field of NLP, working with text data is a common operation. LLMs like GPT, Mistral, and Claude transform entire pieces of text into vector databases, where each word has a specific representation depending on the context. With this type of transformation, the model is able to understand the prompt and generate an appropriate response.

For this work, the raw data collected are strings of characters, specifically domain names. Unlike typical text data, these domains lack context. Thus, there

are two primary transformations that can be applied to process domain names effectively.

Feature Engineering The selection and extraction of features to detect DGA domains is extensively discussed in the literature [41]. This process is particularly crucial for ML algorithms such as Random Forest, Support Vector Machine, and Logistic Regression, where automatic feature extraction is not performed, unlike in models based on Neural Networks. Consequently, ML models are often referred to as *feature-based* models, whereas neural network models are known as *featureless* models.

Consider the domain `google.com` as an example. A simple feature that can be extracted is the length of the domain (excluding the dot), which is 9. Features can relate to various aspects such as domain/subdomain names, character information, linguistic properties, or even n-grams. As a result, a domain can be represented by a vector of length n (number of features), where each component of the vector provides specific information about the domain.

Embedding A transformation of a word, given a certain context, into a low-dimensional vector through mathematical operations and manipulations is called embedding. Embeddings are useful because they provide a dense representation of words in a continuous vector space, capturing semantic meanings and relationships. Each word in the vocabulary is represented by a low-dimensional vector, embedded into the same space, where similar words have similar vectors, meaning their vectors are close to each other in the vector space.

The evolution of embeddings has seen significant advancements over time.

- Initially, simple techniques like *one-hot encoding* and *index vectors*, shown in figure 3.1-3.2, were used. One-hot encoding represents each word as a vector with a single value (1) and the rest as zeros, which is computationally straightforward but results in sparse, high-dimensional vectors that lack meaningful relationships between words [18].

- The introduction of Word2Vec marked a significant advancement [28]. Word2Vec includes two models: Continuous Bag of Words (*CBOW*), which predicts target words from surrounding context words, and *Skip-Gram*, which predicts context words from a target word. These models generate dense vectors that efficiently capture word similarities but do not account for context variability.

- More recent developments focus on contextualized embeddings. *ELMo* (Embeddings from Language Models) uses a bidirectional LSTM to generate word representations that are sensitive to the context in which words appear [34]. *BERT* (Bidirectional Encoder Representations from Transformers) further improves on this by using transformer-based architectures to produce contextually rich embeddings that are pre-trained on large text corpora and fine-tuned for specific tasks [15]. These advancements in embeddings have significantly improved the performance of NLP models.

For our work, we decided to apply to the data three different preprocessing methods. Each desiegnes approach is explained in the next Sections.

3.2.1 Classic approach

This approach is comparable to the Feature Engineering transformation described in the previous section. We selected and extracted from the data different features each providing specific information about the domains. The features we extracted include:

- **Length**: it indicates the length of the full domain without considering the dot.
- **Length TLD**: it gives the length of the Top Level Domain (TLD), which corresponds to the part of the domain after the dot.
- **Unique characters**: it indicates how many unique characters the domains have.
- **Meaningful word**: given an English vocabulary with over 200000 English words, it specifies whether a domain includes one of those words or not (0 stands for "no", 1 stands for "yes").
- **Index word**: if the meaningful word is included in the domain, given the same English vocabulary, it returns the index of that word.
- **Entropy**: it returns the Shannon Entropy for each domain. It is calculated as: $E = -\sum_{i=1}^{n} p_i \cdot \log_2(p_i)$ where p_i indicates the probability of occurrence of the i-th character in the domain name.
- **Numeric Digits**: it defines how many digits are included in the domain.
- **Ratio vowel-consonants**: it indicates the ratio between the vowels and the consonants.
- **Consecutive consonants**: it defines the longest consonant sequence.
- **Consecutive vowels**: it defines the longest vowel sequence.
- **Consecutive digits**: it defines the longest digit sequence.

Three examples of domains transformed with this approach are shown in Table **??**

Most of the time, this preprocessing method is employed just for ML algorithms, where the extraction of features and parameters is not executed by the algorithm itself. Thus, feature extraction in this case is an operation that is not performed automatically by the model.

In this project, we apply this approach for ***Random Forest*** model (RF-FEAT) and for ***Multi-Layer Perceptron*** model (MLP-FEAT).

3.2.2 Tf-Idf approach

Tf-Idf (Term Frequency - Inverse Document Frequency) is a popular technique used in NLP to attribute the importance of a word in a certain context, given a collection of documents [37].

	Examples		
Feature	*car.com*	*ssdyb4f.er*	*dnote.it*
Length	6	9	7
Length TLD	3	2	2
Unique	5	8	6
Vocab	1	0	1
Index vocab	2032	0	12003
Entropy	2.4	4.32	3.12
Digits	0	1	0
Ratio vow-cons	0.5	0.17	1.33
Seq cons	2	5	2
Seq vow	1	1	3
Seq dig	0	1	0

Table 3.1: Examples of features extraction

It is based on two factors: the **Term Frequency** (TF), which represents the frequency of the word t in a document d, and the **Inverse Document Frequency** (IDF), which is given by the formula

$$\mathrm{idf}_i = \log\left(\frac{N}{\mathrm{df}_i}\right)$$

where N is the number of documents and df_i is the document frequency of the word i.

The final weight of the word i, for a given document j, is

$$w_{ij} = (1 + \mathrm{tf}_{ij}) \cdot \mathrm{idf}_i$$

Consider a collection of Shakespeare's 37 plays:

- The word *Romeo* is extremely frequent but appears in just one play. Consequently, its *idf* will be high.
- The word *action* is not very frequent but is present in each play. Thus, its *idf* will be low.
- The word *football* is not present in any play. In this case, the *idf* is zero.

The same approach can be applied to preprocess domain names. Instead of working with words, it is possible to divide the domains into ngrams (sequences of n letters). Compared to general Tf-Idf, the domains correspond to documents and the ngrams correspond to terms. Thus, there is a collection of N domains, and each one contains i ngrams.

In the original paper, this method was applied to several classifiers. We implemented it through `TfidfVectorizer` method by ScikitLearn that works automatically. The algorithm extracts i most popular n-grams, where in our case we choose

$i = 1000$ and $1 \leqslant n \leqslant 3$, among all the domains, creating *i* new features. Then, it assigns a weight, based on the previous Tf-Idf formula, to each domain (row) for each extracted ngram (column).

Table 3.2 shows an example with three words as vocabulary (*N=3*) and five extracted unigrams (*i=5*, *n=1*).

	Unigrams				
words	**a**	**c**	**i**	**o**	**t**
ciao	0.5	0.5	0.5	0.5	0.
come	0.	0.7	0.	0.7	0.
stai	0.51	0.	0.51	0.	0.68

Table 3.2: Example of Tf-Idf algorithm with three words

The Tf-Idf preprocessing method might be belong to the Feature Engineering class as it transforms domains into vectors, exploiting specific domain aspects using ngrams.

We can consider Tf-Idf preprocessing method part of the Features Engineering set of approaches, as it transforms domains into vectors, exploiting specific domain aspects using ngrams. However, it has some intrinsic characteristics that makes this approach closer to the world of Information Retrieval and NLP.

We applied Tf-Idf as preprocessing of data to ***Random Forest*** model (RF-TFIDF), and to ***Multi Layer Perceptron*** model (MLP-TFIDF).

3.2.3 Embedding approach

As explained in Section 3.2, preprocessing data using embeddings has become crucial in NLP applications (e.g., LLMs).

Pretrained methods, like CBOW or Skip-Gram, are useful for simple tasks with short texts, but they do not account for polysemy and new words. Newer methods, such as BERT and ELMo, are very powerful as they capture intrinsic relationships between words and can handle new words. During training for a specific task, the weights within these models represent the embeddings.

Most of these embedding methods have been extensively used for DGA detection tasks in DL models [41, 35]. Among all these methods, we decided to the `EmbeddingLayer` from Keras to embed the domains [42].

As a starting point, the input domain name must be transformed into a vector, for example, by applying an index transformation converting each character to its ASCII code. The embedding layer then projects these sequences of input vectors, corresponding to the domain names, from the input space $S \supset Z^l$ to a sequence of vectors in $R^{d \times l}$. Here, l is the maximum length determined from the data available. The input domain consists of non-redundant valid domain name characters (lowercase alphanumeric characters and periods) converted into numerical values. The output dimension d is a tunable parameter that defines the size of the embedding vectors. Figure 3.5 illustrates an example of the steps involved in embedding a domain name.

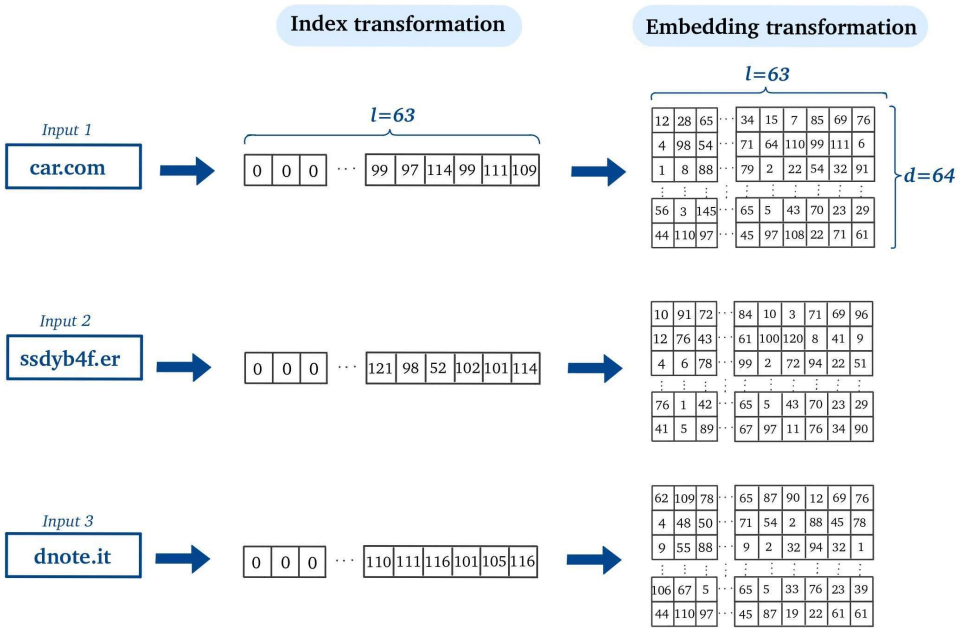


Figure 3.5: Examples of Embedding transformation

This method works exclusively for models based on Neural Networks as it involves an automatic feature extraction of domain patterns. Unlike previous methods, there is no need to divide the domains into ngrams or create features based on domain properties because, during training, the embedding weights are adjusted and defined through the backpropagation.

The input domain dimension (l) for each model is always the same and it corresponds to the maximum length of the longest domain, which is 63 in our case. The embedding dimension (dxl) varies from model to model, as shown in Table 3.3. More details about hyperparameters choices will be given in Section 4.1.1

We applied this last approach to three different models: ***Multi-Layer Perceptron*** (MLP-EMB), ***BiLSTM-CNN*** (BiLSTM-EMB), and ***Transformer Hybrid*** (TRANSF-EMB).

	Input dim (l)	**Embedding dim (dxl)**
MLP-EMB	63	64x63
BiLSTM-EMB	63	32x63
TRANSF-EMB	63	128x63

Table 3.3: Models Embedding dimensions

Chapter 4
Experimental Evaluation

In any ML project, understanding whether a model performs well for a certain task requires thorough evaluation. The metrics used to indicate how a model behaves given some data are called *evaluation metrics*.

Clearly, the performance of a model depends on many different factors, and each phase of an ML project, if not assessed appropriately, might lead to poor results [26].

For instance, the amount and quality of data available are crucial factors that significantly influence performance. Furthermore, without understanding the data we have, including the type of data, feature selection, and class imbalance, expecting good performance is unlikely.

Another possible cause of low evaluation results is the choice of the model. Not all models are appropriate for specific data, so the best solution is to try different systems. Moreover, linked to the choice of the model, we must consider the environment and setup used for evaluation, as, for example, having more GPUs available allows running larger and more complex architectures.

Lastly, an accurate selection of the evaluation metrics to use is fundamental to address the task for which the model is trained. Each type of task can be evaluated with several metrics:

- *Classification task*: Accuracy, Recall (Sensitivity), Precision, F1, etc.
- *Regression task*: Mean Squared Error, Mean Absolute Error, R-squared, etc.
- *Translation task*: BLEU, ROUGE, METEOR, etc.
- *LLM Tuning task*: Perplexity, Human Evaluation, BERTScore, etc.

In each of these tasks, there can be situations in which one metric is more adequate than another. For instance, if we are conducting an experiment on cancer diagnosis (classification type), *Sensitivity* is the most important measurement criterion. Conversely, if we are testing for a common cold (classification), *Precision* would be more appropriate. A detailed explanation of classification evaluation metrics will be provided in section 4.2.

In the next sections, we will highlight some points cited above: *Data* and *Setup* employed, *Results* obtained through evaluation metrics, and a brief *Discussion* on the performances.

4.1 Dataset and Setup

Research projects on DGA domains detection typically rely on domain strings and their corresponding labels (DGA/notDGA or specific DGA families). While data preprocessing can vary, as discussed in Section 3.2, obtaining suitable data is crucial for project success. In both academic and some industrial DGA detection projects, researchers often cannot collect or extract DGA domains firsthand. This limitation exists because deliberately receiving cyber-attacks is unsafe and requires highly specialized domain knowledge. As our work falls into the research project category, we relied on existing databases to collect our data.

The choice of the dataset is extremely delicate. High-quality data, as mentioned in the introduction of this chapter, is a prerequisite that ensures a large amount of data, a good balance between classes, and, of course, suitability for the final task.

The dataset we selected, known as "DGA dataset" [32], include domains names from various archive sources like Alexa Top 1 Million (the most popular repository for legitimate domains, handled by Amazon [4]). Table 4.2 and Table 4.2 show the composition of the dataset.

	Features			
	domain	**host**	**isDGA**	**subclass**
Total Values	160,000	159,998	160,000	160,000
Unique Values	160,000	159,235	2	9
Missing Values	0	2	0	0

Table 4.1: Dataset overview

		Unique Values	
		Binary	***Multiclass***
isDGA	*dga*	80,000	
	legit	80,000	
subclass	*alexa*		42,614
	legit		37,384
	cryptlocker		37,254
	newgoz		9276
	gameoverdga		8461
	nivdort		8456
	necurs		8331
	goz		6136
	bamital		2086
		160,000	160,000

Table 4.2: Dataset classes composition

As we can note from Table 4.1, the dataset is composed of four columns. The feature *domain* represents the string of the website without the top-level domain,

while the column *host* indicates the complete domain string, including the top-level domain (Example: *domain* `google`, *host* `google.com`). The last two features are the label columns for the binary and multiclass classification tasks, respectively, *isDGA* and *subclass*.

With 160,000 rows, the dataset can be considered a medium-sized dataset. In *isDGA*, there are two classes, ***dga*** and ***legit***, that indicate whether a domain is DGA or not. *subclass* includes nine different classes; two of them are benign domain families, ***alexa*** and ***legit***, and the other seven are DGA families. There are a few missing values and some duplicates, always in the *host* feature.

Table 4.2 illustrates how the values are distributed in the target columns. For the binary task, it is easy to see that in the target column *isDGA*, the two classes, ***dga*** and ***legit***, are perfectly balanced. Conversely, the classes for the multiclass task are not well balanced. Here, we have some DGA families, like ***bamital*** and ***goz***, which contain only a few thousand domains. In contrast, benign domain classes such as ***alexa*** and ***legit***, or DGA families like ***cryptolocker***, have more than 37,000 domains.

The dataset is divided into **training** and **testing** sets. The training part, which represents 80% of the total, is used for training the model and tuning its hyperparameters. The testing part, 20% of the total, is used to evaluate the final model's performance and assess how well it generalizes to unseen data.

For the training of the model, we used a technique called `KFoldCV` (where CV stands for *cross-validation*). This technique involves splitting the training data into K folds (in our case, we chose K=3), ensuring that each fold is used as a validation set once, while the remaining folds are used for training. In each iteration, the model is trained on 80% of the training data and validated on the remaining 20%, effectively utilizing the entire training dataset for both training and validation. This process helps to provide a robust estimate of the model's performance and to select the best hyperparameters. Overall, this means that around 20% of the training set (which is 20% of the 80% of the total dataset) is used as the **validation** set during each fold of the cross-validation process.

The project ran on a GeForce RTX 2080 Rev. A-8 GB in a Python environment. Considering the hardware component available, a dataset of a few hundred thousand data points (medium-sized dataset) is an appropriate choice. In order to run projects with millions of data points with more complex architectures, we usually need more powerful hardware infrastructure.

4.1.1 Parameters choice

Generally, each ML model has a range of parameters, called hyperparameters, that can be settled in order to obtain optimal results. The main difference between hyperparameters and parameters is that the first ones are the configuration settings used to control the learning process of a ML algorithm, whereas the second ones are the internal coefficients or weights that the model learns during training.

There is no absolute perfect choice for these hyperparameters, but during the training process it is possible to search for the best combination through different trials. There are methods that make this process more automatic, such as `GridSearch`, `RandomSearch`, and `AutoML` (using libraries like ScikitLearn and Au-

toSklearn). However, these alternative strategies can be extremely computationally expensive and are most effective when dealing with large models and datasets.

The following part indicate the parameters selected for each model already explained in Chapter 3. The selected values are both valid for Binary and Multiclass models.

Random Forest (RF) This ML algorithm has a simple structure as shown in Figure 3.1, and consequently it has a range of few hyperparameters. We focused just only the two most important ones, the number of trees (*ntrees*) and the number of feature for each tree (*mtry*).

Table 4.3 illustrates the hyperparameters identified for each RF model with its corresponding preprocessing method.

	Hyperparameters	
	ntrees	***mtry***
RF-FEAT	150	3
RF-TFIDF	200	32
RF-EMB	200	10

Table 4.3: RF models Hyperparameters

Multi-Layer Perceptron (MLP) The choice of hyperparamters for a MLP model is more complicated compared to a classic ML algorithm such as RF. As Figure 3.2 reveals, there are different factors that can be settled: the number of hidden layers (*nlayers*), the number of units in each hidden layer (*units*), the otpimizer (*optimiz*), the learning rate (*learnrate*), the number of epochs for which the model was trained (*epochs*) and the batch size (*batch*).

Table 4.4 show our choices for MLP based models.

	Hyperparameters					
	nlayers	***units***	***optimiz***	***learnrate***	***epochs***	***batch***
MLP-FEAT	2	64,32	Adam	0.001	20	32
MLP-TFIDF	3	128,64,32	Adam	0.001	5	32
MLP-EMB	2	64,32	Adam	0.001	20	32

Table 4.4: MLP models Hyperparameters

CNN-BiLSTM This architecture is formed by different neural network models concatenated as shown in Figure 3.3. Thus, it is composed by a series of hyperparameters that belong specifically both to the single models and to the overall architecture. Following the research that implement this model [29], we decided to focus just only on the core hyperparameters.

This architecture is combined with the *Embedding* preprocessing method resulting in the model that we call *BiLSTM-EMB*. Its hyperparameters are presented in the following list:

- Sequential CNNs blocks (**nCNNs**): *2*
- Filters in CNN (**filtersCNNs**): *2*
- Filter size in layer CNN (**filtsizeCNN**): *256*
- Units of LSTM (**unitsLSTM**): *128*
- Hidden layers in MLP (**nlayMLP**): *3*
- Training epochs (**epochs**): *10*
- Batch size (**batch**): *32*

Hybrid Transformer (TRANSF) As in the previous case of CNN-BiLSTM, the Transformer Hybrid include several single algorithms (MLP, CNN...) inside its architecture. Thus, we decided to select and change just few hyperparameters during the training process. The ones not selected remain unchanged as presented in the original paper [16]. The hyperparameters chosen are the following:

- Sequential transformer blocks in Character part (**transfblock**): *1*
- Attention Heads in Character part (**nheadsChar**): *8*
- CNN filters in Character part (**filtCNNChar**): *128*
- Filter size CNN in Character part (**filtsizeCNNChar**): *3*
- Units of LSTM (**unitsLSTM**): *128*
- Units in MLP Character part (**unitMLPChar**): *128*
- CNN filters in Bigram part (**filtCNNBigr**): *128*
- Filter size CNN in Bigram part (**filtsizeCNNBigr**): *3*
- Units in MLP Bigram part (**unitMLPBigr**): *128*
- Units in MLP Concat part (**unitMLPConc**): *1*
- Training epochs (**epochs**): *3*
- Batch size (**batch**): *32*

4.2 Comparison of results

As already mentioned several times throughout the work, the final goal of this project is to compare different methods designed to detect whether a domain is DGA-generated (malign) or not (benign), and, if it is DGA, to identify to which family the DGA domain belongs.

In general, for a classification task, there is a range of metrics that can be used to evaluate a model's performance. The evaluation metrics we have chosen are very common in the research field and are often the most effective. Before describing each metric one by one, it is necessary to understand the basic components:

- **True Positives (TP)**: The number of correctly predicted benign instances, e.g., the model correctly identifies a legit domain as benign.
- **True Negatives (TN)**: The number of correctly predicted DGA instances, e.g., the model correctly identifies a DGA domain as malign.
- **False Positives (FP)**: The number of incorrectly predicted benign instances, e.g., the model incorrectly identifies a DGA domain as benign.
- **False Negatives (FN)**: The number of incorrectly predicted DGA instances, e.g., the model incorrectly identifies a legit domain as malign.

Based on the definitions above, we are able to define the evaluation metrics.

Accuracy Accuracy measures the overall correctness of the model by calculating the proportion of true results (both true positives and true negatives) among the total number of cases examined. It is given by the formula:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision Precision, also known as positive predictive value, measures the accuracy of the positive predictions. It is the proportion of true positive results among all positive predictions made by the model. It is given by:

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

Recall Recall, also known as sensitivity or true positive rate, measures the ability of the model to correctly identify all relevant instances. It is the proportion of true positive results among all actual positive instances. It is given by:

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

F1 The F1 score is the harmonic mean of precision and recall, providing a single metric that balances both concerns. It is particularly useful when the class distribution is imbalanced. It is given by the formula:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

ROC Curve The Receiver Operating Characteristic (ROC) curve is a graphical plot that illustrates the ability of a binary classifier system as its discrimination threshold is varied. The curve is created by plotting the true positive rate (Recall) against the false positive rate (FPR), where FPR is defined as:

$$\text{False Positive Rate} = \frac{FP}{FP + TN}$$

The ROC curve helps to visualize the trade-off between sensitivity and specificity (1 - FPR) for different threshold values. The Area Under the ROC Curve (**AUC**) summarizes the ROC curve results as a single scalar value. An AUC of 0.5 suggests no discriminative ability (equivalent to random guessing), while an AUC of 1.0 indicates perfect discrimination.

The formulas defined above are valid when we have to discriminate between two classes (i.e., Binary Classification task). When there are more classes (i.e., Multiclass Classification task), each of those evaluation metric is computed for each class, considering one class as positive and all the other classes grouped as negative. To calculate a general metric, then we can define the weighted average (more importance to classes with more instances) or macro average (equal importance to all the classes).

In a potential future cybersecurity application built upon one of the proposed approaches, the main requisite is to not classify domains that are in reality DGA as legit (FN). Otherwise, the malware can still communicate with its masterbot (recall Section 2.2) and the cyberattacker is not neutralized. Thus, we can conclude that Recall might be slightly more critical than Precision for our task. However, it is always cautious to consider metrics that show the general behaviour of a model, such as F1 and ROC-AUC score.

4.2.1 Binary Classification

In the Binary classification task we need to recognize whether a domain is DGA (malign) or legit (benign). Through the preprocessing approaches and models implemented in Chapter 3, Table 4.5 show the results we reached.

	Metrics			
	Accuracy	**Recall**	**Precision**	**F1**
RF-FEAT	0.939	0.942	0.936	0.939
RF-TFIDF	0.984	0.982	0.987	0.984
MLP-FEAT	0.93	0.9	0.967	0.932
MLP-TFIDF	0.992	0.988	**0.996**	0.992
MLP-EMB	0.988	0.983	0.993	0.988
BiSLTM-EMB	0.989	**0.996**	0.984	0.99
TRANSF-EMB	**0.993**	0.992	0.995	**0.993**

Table 4.5: Evaluation metrics (Binary task)

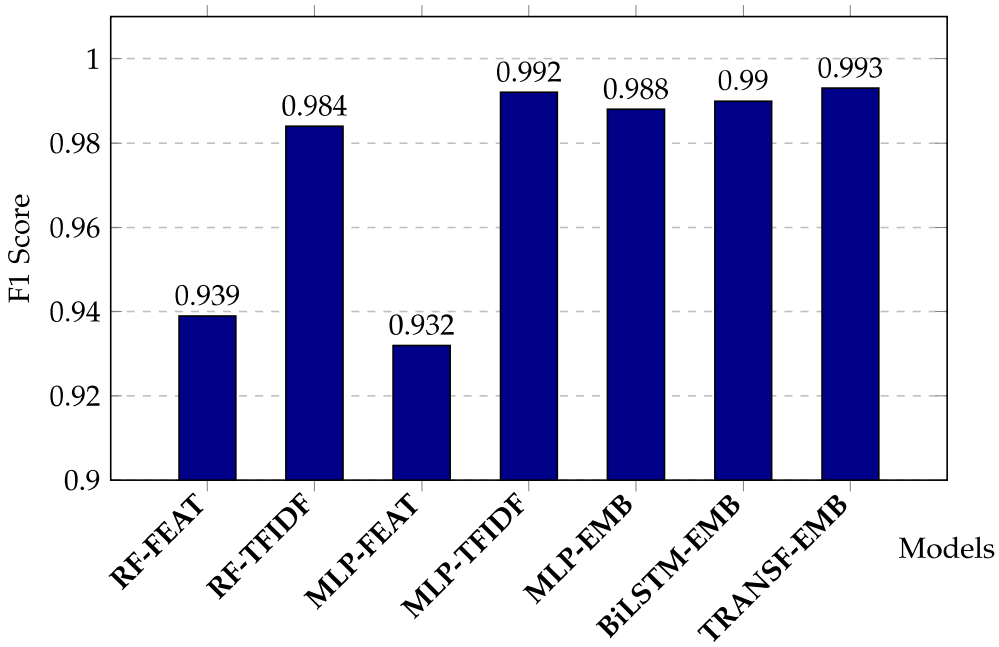


Figure 4.1: Comparison of F1 scores (Binary Task)

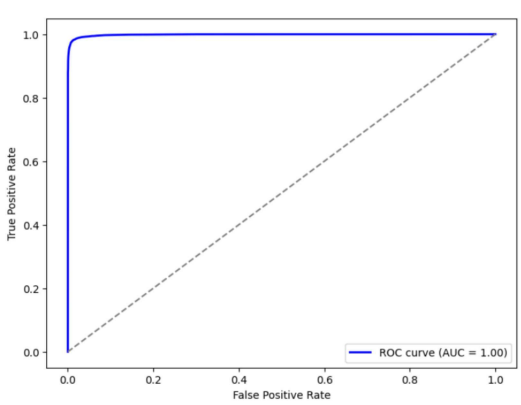
All these values were obtained on the test set with a support (i.e., testing instances) of 31,960 samples, perfectly split between the two classes (*dga*, *legit*).

To make a clearer comparison between the results, Figure 4.1 illustrates how the models scores on F1. According to this metric, that represents the most robust metric we have computed, the best model is the *TRANSF-EMB*, with a F1 score of 0.993. However, all the performances are greater than 0.93, indicating that all the models classify pretty well between the two classes.

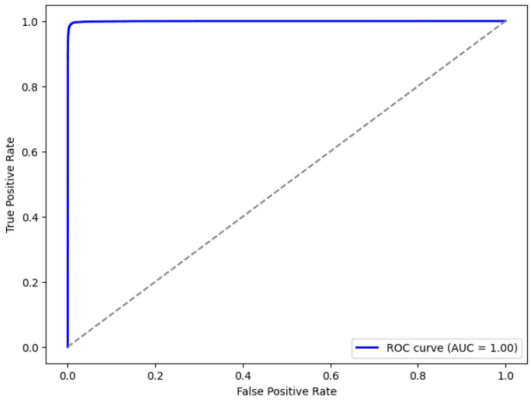
Considering the others metrics, Table 4.1 shows that *TRANSF-EMB* scores almost perfectly also on the Accuracy (0.993), while for Recall and Precision the best classifiers are, respectively, *BiLSTM-EMB* (0.996) and *MLP-TFIDF* (0.996).

Table 4.1 gives a good overview about the performances on classification task. Furthermore, we can analyze how the best three models, according to Recall, Precision and F1, score on ROC curve and AUC. By examining the these metrics, we gain insights into the trade-offs between true positive rates and false positive rates for each model. Additionally, these evaluations help in understanding the robustness and reliability of the models under different thresholds. The figure 4.2 show the ROC curve and the AUC scores of the best models, *MLP-TFIDF*, *BiLSTM-EMB* and *TRANSF-EMB*.

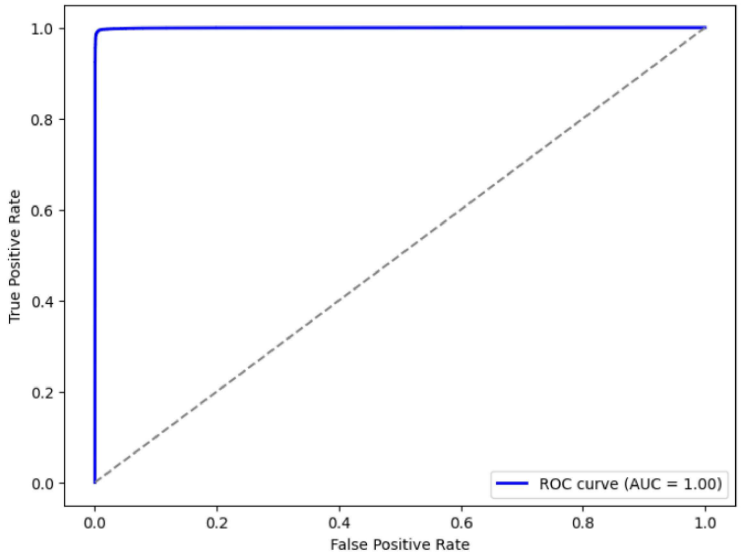
As we can see, each classifier has a perfect ROC curve with an AUC score of 1. These results arise from the high performance in Recall, Precision, and F1 scores, each exceeding 95%. While these values appear ideal, it is important to consider potential limitations. For instance, ML systems that achieve perfect results may overfit, meaning they are too specific for our set of data and they do not generalize well on new completely domains. Additionally, these results could indicate that the classifiers are overly complex for the available data or the specific task. A detailed discussion about these performances is given in Section 4.3.



(a) **MLP-TFIDF** ROC Curve



(b) **BiLSTM-EMB** ROC Curve



(c) **TRANSF-EMB** ROC Curve

Figure 4.2: ROC Curves (Binary Task)

4.2.2 Multiclass Classification

The Multiclass classification task is more complex than the Binary one. Detecting whether a domain belong to a specific DGA family or to category of legit domains is not straightforward and it requires appropriate classifiers and enough data for each class.

We employed the same evaluation metrics defined in the previous sections. As our classes are not perfectly balanced, we decided to calculate both the weighted average and macro average. In this way we can compare the performances from two different perspectives. Before showing the performances, it is necessary to list all the instances employed to achieve the results, that belong to each class (i.e., support):

- *alexa* (Class 0): **7475**
- *legit* (Class 1): **8521**
- *cryptlocker* (Class 2): **1690**

- *newgoz* (Class 3): **7449**
- *gameoverdga* (Class 4): **1854**
- *nivdort* (Class 5): **1689**
- *necurs* (Class 6): **1225**
- *goz* (Class 7): **1664**
- *bamital* (Class 8): **415**

As we can note, the classes are pretty imbalanced. The performances of our models for the Multiclass classification task are represented by Table 4.6 and Table 4.7.

	Metrics			
	Accuracy	**Recall**	**Precision**	**F1**
RF-FEAT	0.647	0.647	0.644	0.645
RF-TFIDF	0.741	0.741	0.741	0.734
MLP-FEAT	0.645	0.532	0.692	0.584
MLP-TFIDF	0.744	0.742	0.743	0.741
MLP-EMB	0.727	0.725	0.727	0.724
BiSLTM-EMB	**0.753**	0.75	0.733	0.733
TRANSF-EMB	**0.753**	**0.752**	**0.769**	**0.749**

Table 4.6: Evaluation metrics - Weighted average (Multiclass task)

	Metrics			
	Accuracy	**Recall**	**Precision**	**F1**
RF-FEAT	0.647	0.688	0.698	0.692
RF-TFIDF	0.741	0.786	0.786	0.782
MLP-FEAT	0.645	0.607	0.723	0.646
MLP-TFIDF	0.744	0.786	0.788	0.786
MLP-EMB	0.727	0.77	0.773	0.771
BiSLTM-EMB	**0.753**	0.791	0.745	0.758
TRANSF-EMB	**0.753**	**0.798**	**0.802**	**0.794**

Table 4.7: Evaluation metrics - Macro average (Multiclass task)

Both tables indicate that *TRANSF-EMB* outperforms all other models in most metrics, except for Accuracy, where *BiLSTM-EMB* matches it with a score of 0.753. This suggests that *TRANSF-EMB*, with its sophisticated architecture, is more suitable for complex tasks such as multiclass classification.

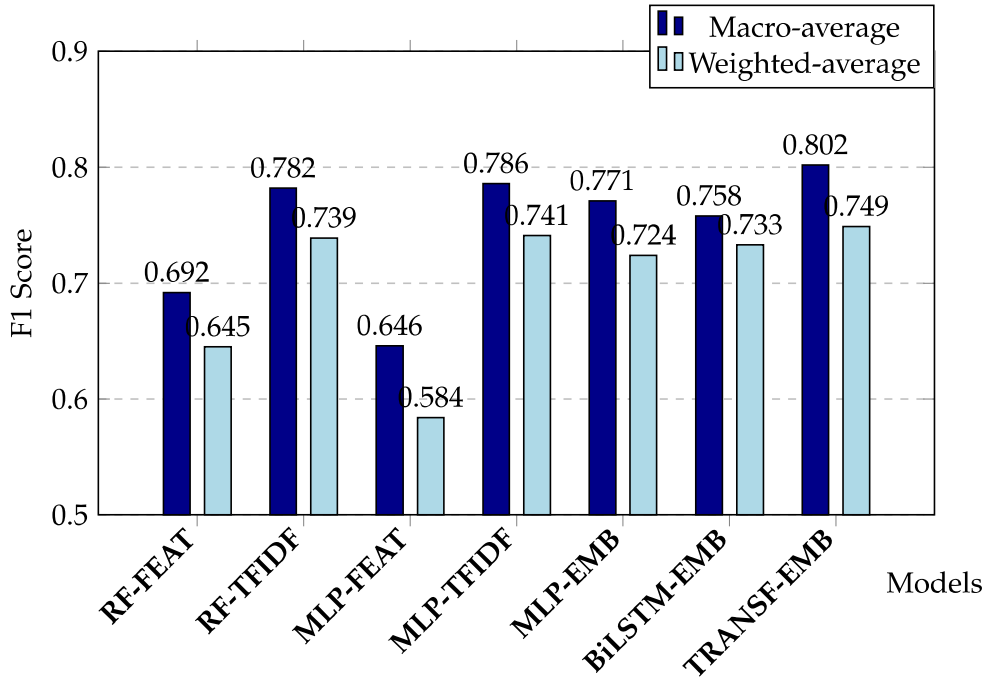


Figure 4.3: Comparison of F1 scores (Multiclass Task)

As shown in Figure 4.3, other models also achieved significant scores, particularly in the F1 metric, with weighted-average scores above 0.70 and macro-average scores above 0.75. These high-performing classifiers, including *BiLSTM-EMB*, *MLP-EMB*, *MLP-TFIDF*, and *TRANSF-EMB*, are all built using neural networks architectures.

Overall, the macro-average values (Table 4.7) are slightly higher than the weighted-average values (Table 4.6). This discrepancy might be due to dataset imbalance, where minor classes (e.g., Bamital, Necurs) achieve higher results but contribute less to the weighted average due to their lower representation. More details will be presented in the next section.

Regarding the best classifier, *TRANSF-EMB*, we can analyze its ROC curve with the AUC score. For a multiclass task, the ROC curve graph is composed by different plots, one for each class. Here, it indicates the trade-off of each class between TPR and FPR, where the positive class is its own class, and the rest of the classes represent the negative class. Consequently, each class has its relative AUC score.

Figure 4.4 displays the ROC curves just described (each curve is computed using weighted-average results).

While our evaluation metrics (Recall, Precision, F1) show maximum values of 0.79, all the AUC scores exceed 0.9. This apparent discrepancy is not unusual and can be explained by the nature of these metrics. The AUC score evaluates the model's ability to discriminate between classes across all possible threshold values, independent of any single chosen threshold. In contrast, Recall, Precision, and F1 are calculated based on a specific threshold, which we left at the default value of 0.5.

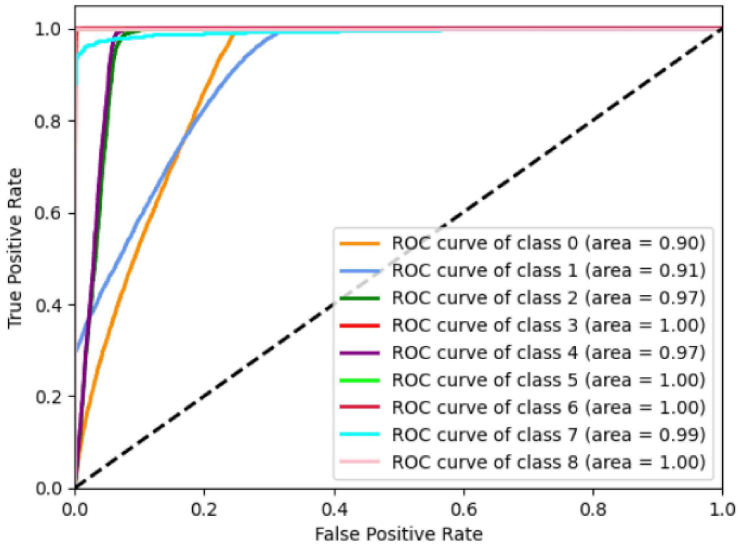


Figure 4.4: ROC Curves TRANSF-EMB (Multiclass task)

4.3 Discussion of results

By looking at the results, it is immediately apparent that the scores for binary classification are significantly higher than those for multiclass classification. This can be attributed to two main reasons. First, the nature of the task itself plays a role. Generally, distinguishing between two classes is simpler than differentiating among multiple classes. This simplicity often results from more effective feature extraction and more efficient weight training within the models. Second, the composition of the data contributes to the discrepancy. The data for the binary classes (*dga* and *legit*) were perfectly balanced, whereas the *DGA families*, representing multiple classes, were not balanced. Imbalanced datasets can negatively impact the performance of any model.

Regarding the binary classification, all models show excellent results. As illustrated in Figure 4.1, the F1 scores are closely grouped within a range of less than one decimal (0.93-0.99), indicating consistently good performance across all classifiers. This consistency is also validated by the observation that different models achieve the highest performance in different evaluation metrics, except for Accuracy and F1 score, where *TRANSF-EMB* is the best in both.

The situation is different for the multiclass task, where the overall performance is lower compared to the binary classification task. There is greater variability in model performance. For instance, in the weighted-average recall, *MLP-FEAT* scored 0.532, while *TRANSF-EMB* achieved 0.752, showing a difference of over two decimal places. A similar pattern is observed in the macro-average recall, where *MLP-FEAT* recorded 0.607, whereas *TRANSF-EMB* scored 0.798, again with a difference of nearly two decimal places. Furthermore, *TRANSF-EMB* continuously outperforms other models by achieving the highest scores between all performance metrics, both for the weighted-average and macro-average. The imbalance in the dataset and the limited amount of data available for some classes need a more complex and sophisticated architecture to achieve better than average performances.

Considering only the preprocessing approaches described in Section 3.2, which include Feature Engineering (*FEAT*), Tf-Idf (*TFIDF*), and Embedding (*EMB*), it is evident that the *FEAT* approach performs the worst in both tasks. One possible reason for this is can be attributed to the inefficient feature extraction. A useful technique available in Random Forests, known as `FeatureImportance`, allows us to see how much each feature contributes to the overall performance. This explainable strategy provides a clear understanding of the significant features within the model.

Figure 4.5 and Figure 4.6 illustrate the `FeatureImportance` plots for *RF-FEAT* in both binary and multiclass tasks. The five most important features are the same in both graphs, as are the five least significant features. Notably, the four least significant features (*numeric digits*, *meaningful word*, *consecutive vow*, *consecutive dig*) appear in the same order in both histograms and collectively contribute less than 0.15 to the importance value. In contrast, the top three features (*consecutive cons*, *index word*, and *length*) together represent more than 0.55 of the importance value in both binary and multiclass tasks. Therefore, to improve model performance with the *FEAT* preprocessing approach, it would be beneficial to replace the four least significant features with more effective ones.

Among the other two preprocessing approaches, the Embedding (*EMB*) method achieved the best performance in both classification tasks. Notably, *TRANSF-EMB* scored high across all metrics. Other models using the *EMB* approach, such as *BiLSTM-EMB*, also delivered important results in some metrics, as shown in Table 4.7 and Table 4.6. Interestingly, the *TFIDF* method produced extremely good performances with Random Forest (*RF*), the simplest model. In the binary classification task, *RF-TFIDF* achieved an F1 score of 0.984. It also performed well in the multiclass task, with an F1 weighted-average of 0.734 and an F1 macro-average of 0.782, closely approaching the best results. This demonstrates that even simple

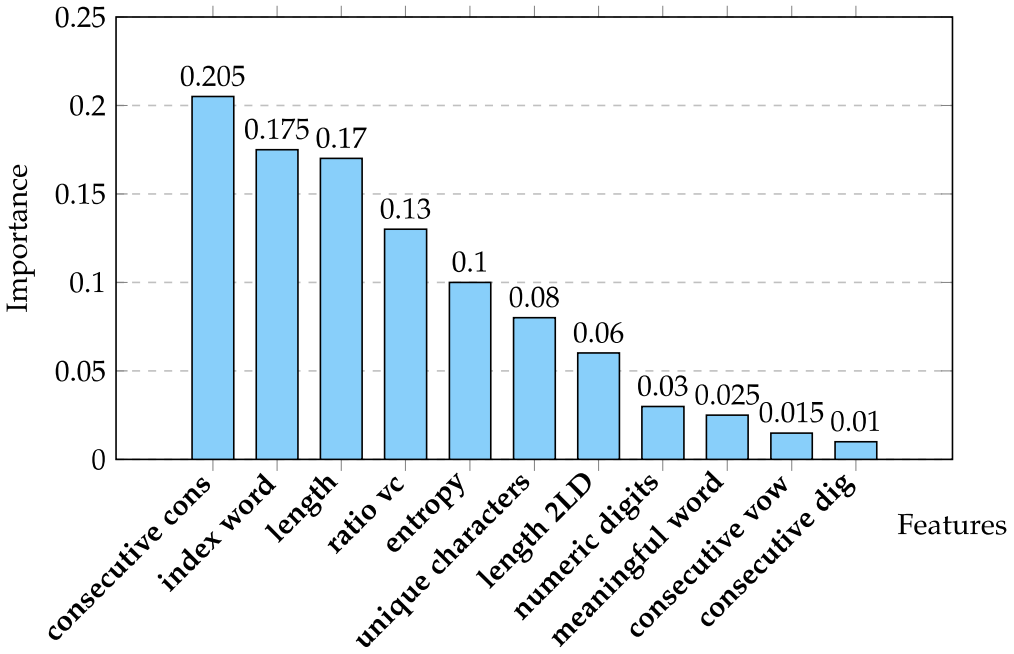


Figure 4.5: Feature Importance FEAT-RF (Binary task)

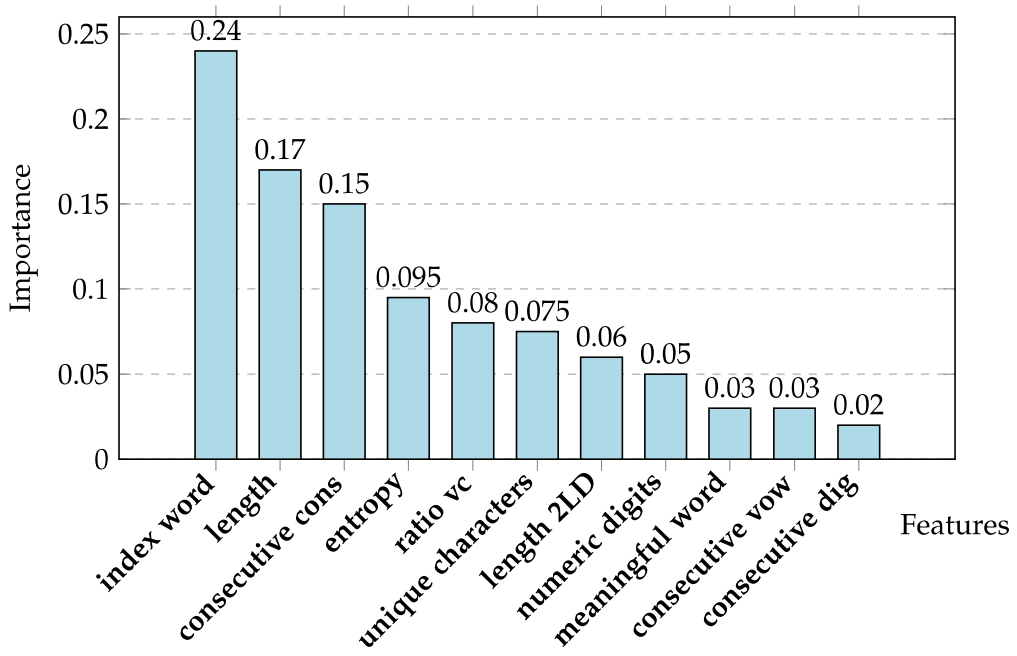


Figure 4.6: Feature Importance FEAT-RF (Multiclass task)

ML algorithms like *RF* can achieve high performance when employed with an appropriate and efficient preprocessing strategy such as *TFIDF*. Therefore, complex architectures are not always necessary to handle classification tasks effectively.

The last point to analyze about model's performances regard ROC curves and AUC scores. As cited in the previous sections, AUC results are almost perfect in any models. Particularly in Multiclass task, where the best classifier performances hit a maximum of 0.8 in F1 macro-average, the AUC value for any classes is always greater than 0.9, as shown in Figure 4.4. To justify these results, we must consider that the ROC curve measures the model's ability to distinguish between classes across all possible threshold values. Unlike precision, recall, and F1 scores, which are calculated at a specific threshold, the ROC curve evaluates the trade-off between true positive and false positive rates over a range of thresholds (in our case we leave as threshold the default value 0.5). This comprehensive evaluation often results in higher AUC values because it captures the model's overall discriminative ability, rather than its performance at a single point. Potentially, by adjusting the 0.5 threshold in the Multiclass models, we could improve these performances obtaining results similar to the Binary classification task.

Chapter 5
Production Phase

Training and evaluating a ML project using different metrics does not represent the final stage in the project life cycle. As mentioned in Section 2.3.3, in research environments, once the model achieves the desired results on the test set, the paper is published, and no further steps are taken. However, since most of the preprocessed methods and models are developed with Cy4gate, a company that provides AI services and applications in cybersecurity and cyberintelligence, we can considered this project as a research work with additional elements focused on deploying the models. Therefore, we decided to include a section on the production phase to fully cover the development of an ML model from start to finish.

In Section 5.1, we will explain how the deployment of an ML model works, including the applications that companies use today to deploy systems. In Section 5.2, we will demonstrate a toy simulation with new data from different sources. Additionally, we will build a small web service using `Streamlit` to showcase some examples. This service is not intended to represent the deployed application developed by the company but rather serves as a practical example.

5.1 Dive into deployment

In Section 2.3.3 we already explained some differences between ML in research and production. The complexity and nature of any ML system, built locally or in cloud, can vary, but the foundational framework remains consistent. Typically, data is sourced, cleaned, and preprocessed before extracting or creating important features to train the ML model for a specific task, as in our case. Once trained, the model is ready to be used in a production environment where it can be exposed to new, unseen data for making predictions through APIs.

However, deploying a model introduces several challenges [38]. As depicted in Figure 5.1, after building a successful ML model, the cycle involves deploying the trained model into production and continuously monitoring its performances. When the model's performances falls below the expected benchmark, it has to be retrained and evaluated to replace the old model. This process includes model management, such as versioning and feature tracking. The new model is then deployed again without breaking existing user requests, continuing the cycle as shown in Figure 5.1.

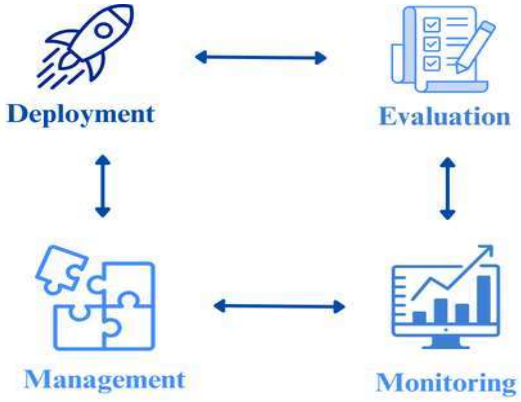


Figure 5.1: Deployment cycle

Although the concept of deployment may seem intuitive due to the large amount of available ML applications, it differs significantly from typical software applications like mobile apps. This difference primarily arises from the underlying model and data. Deploying an ML model in production involves integrating it into an existing business application, usually showing it as *REST API endpoints* to serve requests. There are different approaches to deploy a model, including building a web-service through Flask or Streamlit, using containers systems such as Docker, or exploiting online platform like Azure AI or Amazon Web Service (AWS) through the use of Kuberntess.

The importance of deploying ML models in production lies in extracting real value from them. Successful deployment of an ML model requires collaboration among data scientists, data engineers, application developers, MLOps/DevOps, and business team members, that grouped integrate the model into the application to enhance its functionality with predictions and insights.

Not deploying ML models is like training for a sports event without competing, limiting their potential impact. However, stand-alone models, which operate independently, can be simpler and faster for building, training, and making predictions. This approach works well for quick insights and decision-making. Generally, for large datasets, streaming data, or high user volumes, models must be integrated into applications to handle data efficiently. Integrated models are essential for processing continuous streams and handling numerous simultaneous user requests, ensuring consistent performance and fast responses.

In conclusion, deploying an ML model in production does not ensure consistent prediction quality. Performance often deteriorates over time due to various causes like a phenomenon known as *drift*, where the model encounters real and new data. Effective model management and monitoring are crucial to address all the issues that arise, ensuring the model adapts to new data and maintains its performance.

5.2 Toy simulation

For this practical simulation, we considered just the binary classification task as purpose of simplicity.

As primary step, we need to use new domains, both *dga* and *legit*, to test outside of the developing environment our binary models. We decided to collect the data from different sources, including domains from new datasets [30, 46] and syntethic-domains generated from two different Large Language Models, GPT-4o and Claude. In total, we obtained 200 domain names divided perfectly balanced:

- **dga domains**: *100* samples (90 from datasets, 10 synthetically generated).
- **legit domains**: *100* samples (80 from datasets, 20 synthetically generated).

It is fundamental to note that in this case *dga* domains do not belong just to the seven dga classes of our original dataset. As regards the two new datasets, we extracted DGA domains randomly from 40 different DGA families, whereas for the synthetic-domains, we prompted to generate different dga domains without specifying any type of class. Therefore, the data for this deployment simulation are extremely heterogeneous, crucial side that allows to understand whether a model is still robust with completely different data.

As mentioned in Section 5.1, a continuous management and monitoring of the system is necessary when the production phase starts. Hence, to have a better trace of models results, we stored the data in a log format. We show some examples:

```
2024-07-10 12:00:00, 00 - Domain: battle.net, Prediction: 1 (legit)

2024-07-10 12:00:00, 01 - Domain: nypost.com, Prediction: 1 (legit)

2024-07-10 12:00:00, 02 - Domain: kmocuufys.eu, Prediction: 0 (dga)
```

Table 5.1 illustrates how the trained binary models work on this simulation, where each column represent how many corrected instances the model predict.

	Corrected Predictions		
	dga (0) **100 samples**	**legit (1)** **100 samples**	**overall** **200 samples**
RF-FEAT	74	93	167
RF-TFIDF	67	98	165
MLP-FEAT	68	74	142
MLP-TFIDF	**87**	71	158
MLP-EMB	59	**100**	159
BiSLTM-EMB	67	**100**	167
TRANSF-EMB	72	**100**	**172**

Table 5.1: Corrected predictions for each model

The predictions on the simulation approximately follow the same behavior as the binary classifiers performances illustrated in Table 4.5. Here, *legit* domains are classified more accurately compared to *dga* domains. Notably, three models, *TRANSF-EMB*, *BiLSTM-EMB*, and *MLP-EMB*, perfectly classify all *legit* domains (100/100). The best performance on *dga* samples is achieved by *MLP-TFIDF*, with 87

out of 100 correct classifications. Considering the overall set of new data, *TRANSF-EMB* emerged as the best classifier, classify not correctly only 28 domains. These results correspond perfectly with the performances obtained in the binary classification task (Table 4.5), where *TRANSF-EMB* was also the overall best model.

Nevertheless, always choosing the best overall classifier, *TRANSF-EMB*, for any detection problem might not be the best strategy. For example, if our service prioritizes detecting only *dga* domains over *legit* domains, then *MLP-TFIDF* might be a better choice for our predictions. This could be particularly important in scenarios such as monitoring and securing enterprise networks. By focusing on *dga* detection, network administrators can quickly identify and block these malicious communications, preventing data breaches and other cyber threats.

The production phase ends with a real practical application. We created an interactive web application using the library `Streamlit`, an open-source tool that allows to turn data scripts into shareable web apps. This web-service has been implemented with our trained models (binary classification) that are able to detect whether a domain is *dga* or *legit*. The application shows a basic UI (User Interface) with a bar to prompt a domain name and a button used to output the domain class. Figure 5.2 and Figure 5.3 show the web-application UI.

As we can note, the interface is extremely easy to understand. It is composed of a blue prompt bar where the user enters the domain name, and a red "Classify" button that triggers the classification of the entered domain. Since *TRANSF-EMB* achieved the best overall performance in both the training and production phases, we implemented it to perform two example predictions. We selected two random domains, one from each class, from the new set of domains and computed the predictions. As a result, both predicted classes are correct:

- *jntvwhuwxslkrqtwstl.ru*: **DGA** (predicted class), DGA (actual class)
- *google.com*: **Legit** (predicted class), Legit (actual class)

Clearly, this web service can be employed with any trained model. For instance, to achieve more robust predictions, it is possible to predict the class of a domain using all models and then use a majority voting system to decide the final class. Additionally, any domain name can be entered into the prompt to see the predicted output class.

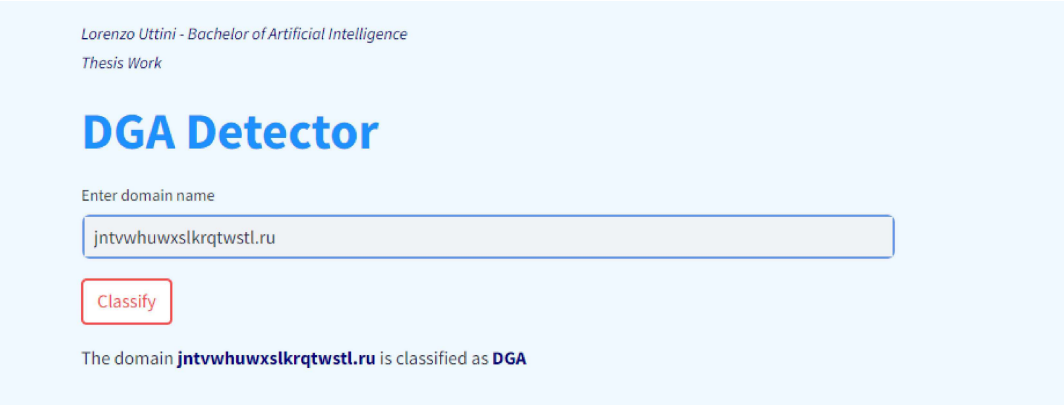


Figure 5.2: DGA Detector: domain classified as DGA

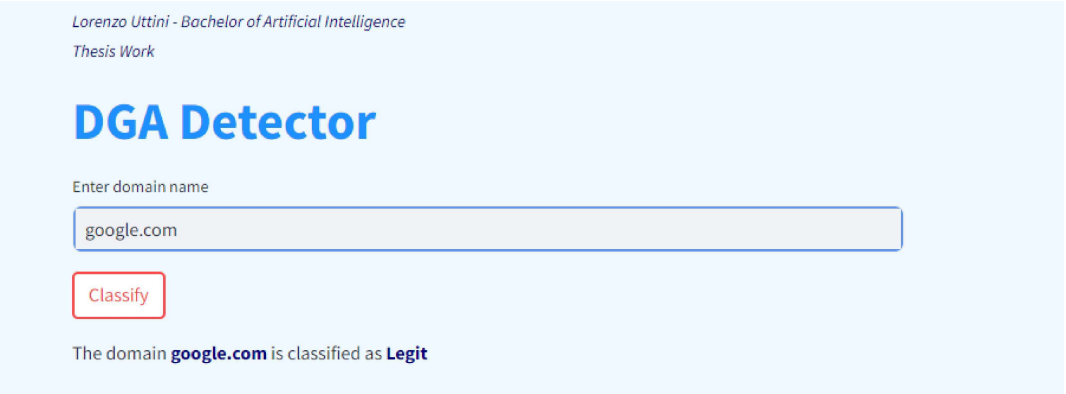


Figure 5.3: DGA Detector: domain classified as Legit

Despite its simplicity, this web service is a real example of a deployed ML model. As explained in the previous section, this model could be implemented using other approaches (Kubernetes, Azure AI, Google Cloud, etc.) and has to be monitored and managed continuously.

Chapter 6
Conclusions

In this work, we extensively applied AI methods to detect a cybersecurity threat known as DGA-based domains. We compared different ML models and architectures for two types of classification tasks: binary and multiclass. In the binary task, domains were divided into two perfectly balanced classes, *dga* and *legit*, while in the multiclass task, the data was composed of nine different classes, some representing DGA families.

We implemented and modified four different classifiers: *Random Forest*, *Multi-Layer Perceptron*, *BiLSTM-CNN*, and *Hybrid Transformer*. Additionally, we employed three different methods to preprocess the domain strings: *Feature Engineering*, *Tf-Idf*, and *Embedding*. Overall, we utilized seven classifier-preprocess combinations: *RF-FEAT*, *MLP-FEAT*, *RF-TDIDF*, *MLP-TFIDF*, *MLP-EMB*, *BiLSTM-EMB* and *TRANSF-EMB*.

In the binary classification task, all models achieved excellent performance across nearly all metrics. Notably, *TRANSF-EMB* obtained the best results in both Accuracy and F1-score (**0.993**), while *BiLSTM-EMB* and *MLP-TFIDF* excelled in Recall (**0.996**) and Precision (**0.996**), respectively. The ROC curves for these three models confirmed their superior performance.

In the multiclass task, we computed two types of evaluation metrics: weighted-average and macro-average. Unlike the binary task, the results here were slightly lower, ranging from 0.7 to 0.8 for each metric. The best performances were again achieved by *TRANSF-EMB*, with an average-F1 score of **0.749**, a macro-F1 score of **0.794**, and an AUC score greater than **0.9** for every class.

Overall, we can state that the most robust model was the *Transformer Hybrid* and the best preprocessing approach was the *Embedding*. The combination of these two factors, *TRANSF-EMB*, represent the best classifier.

To simulate a production environment for our ML project, we conducted a toy simulation to test all models on unseen data. We built a small web service using `Streamlit` where users could input any domain. We then tested all trained models on 200 new domains (100 *dga* and 100 *legit*). The performances confirmed our previous results, with *TRANSF-EMB* obtaining the best results by making **167** correct predictions out of 200. However, when considering only *dga* domains, *MLP-TFIDF* performed best, correctly identifying **87** out of 100 *dga* domains. This indicates that depending on the final goal of the product, *MLP-TFIDF* might be preferred over the overall best classifier *TRANSF-EMB*.

To conclude, we showed how to build a ML project on detecting DGA generated domains from the begin to the end. We utilized different data preprocessing approaches, we built and trained different models obtaining excellent results, and finally we demonstrate how to deploy a ML system.

Clearly, the detection of domains generated by DGAs, and consequently our work, can be improved for future developments. For instance, malware DGAs are dynamically changing very often, making the collection of updated DGA domains tremendously complicated yet crucial for advancing in this field. Additionally, improvements to existing models could involve building new, versatile, and robust architectures that exploit unseen aspects of DGA malware. Furthermore, the deployment of a model must always be followed by constant monitoring and supervision to avoid inefficiencies in our products.

References

[1] A. Ahluwalia, I. Traore, K. Ganame, and N. Agarwal. Detecting broad length algorithmically generated domains. In *Intelligent, Secure, and Dependable Systems in Distributed and Cloud Environments*, chapter 2, pages 19–34. Springer International Publishing, 2017. (Cited on page 17)

[2] D. Alomari, F. Anis, M. Alabdullatif, and H. Aljamaan. A survey on botnets attack detection utilizing machine and deep learning models. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 493–498, June 2023. (Cited on page 5)

[3] Shun'ichi Amari. A theory of adaptive pattern classifier. *IEEE Transactions on Electronic Computers*, EC-16(3):279–307, 1967. (Cited on page 19)

[4] Amazon. Alexa - top sites. (Cited on page 32)

[5] P. Amini, M. A. Araghizadeh, and R. Azmi. A survey on botnet: classification, detection and defense. In *2015 International Electronics Symposium (IES)*, pages 233–238. IEEE, September 2015. (Cited on page 7)

[6] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou, Saeed Abu-Nimeh, Wenke Lee, and David Dagon. From Throw-Away traffic to bots: Detecting the rise of DGA-Based malware. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 491–506, Bellevue, WA, 2012. USENIX Association. (Cited on page 9)

[7] BaderJ. Domain generation algorithms. `https://github.com/baderj/domain_generation_algorithms`, 2024. Accessed: 2024-07-11. (Cited on page 10)

[8] Thomas Barabosch, André Wichmann, Felix Leder, and Elmar Gerhards-Padilla. Automatic extraction of domain name generation algorithms from current malware. 2012. (Cited on page 10)

[9] M. Baruch and G. David. Domain generation algorithm detection using machine learning methods. In *Cyber Security: Power and Technology*, pages 133–161. Springer International Publishing, 2018. (Cited on page 17)

[10] Arpad Berta, Zoltan Szaba, and Mark Jelasity. Modeling peer-to-peer connections over a smartphone network. In *1st International Workshop on Distributed Infrastructure for Common Good (DICG'20)*, page 6, Delft, Netherlands, December 7–11 2020. ACM. (Cited on page 8)

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer New York, 1 edition, 2006. (Cited on page 11)

[12] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. (Cited on page 17)

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. (Cited on page 11)

[14] Cisco. What is the difference: Viruses, worms, trojans, and bots?, 2023. (Cited on page 6)

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. (Cited on page 26)

[16] Ling Ding, Peng Du, Haiwei Hou, Jian Zhang, Di Jin, and Shifei Ding. Botnet dga domain name classification using transformer network with hybrid embedding. *Big Data Research*, 33:100395, 2023. (Cited on pages 23, 24, 25, and 35)

[17] Metehan Gelgi, Yueting Guan, Sanjay Arunachala, Maddi Samba Siva Rao, and Nicola Dragoni. Systematic literature review of iot botnet ddos attacks and evaluation of detection techniques. *Sensors*, 24(11):3571, 2024. (Cited on page 7)

[18] Zellig S. Harris. Distributional structure. 1954. (Cited on page 26)

[19] Sepp Hochreiter and JÃ¼rgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. (Cited on page 21)

[20] C. Huyen. *Designing Machine Learning Systems: An Iterative Process for Production-ready Applications*. O'Reilly Media, Incorporated, 2022. (Cited on pages 12 and 14)

[21] B. B. Kang and C. Nunnery. Decentralized peer-to-peer botnet architectures. In Z. W. Ras and W. Ribarsky, editors, *Advances in Information and Intelligent Systems*, volume 251 of *Studies in Computational Intelligence*, pages 217–234. Springer, Berlin, Heidelberg, 2009. (Cited on page 8)

[22] S. Khattak, N. R. Ramay, K. R. Khan, A. A. Syed, and S. A. Khayam. A taxonomy of botnet behavior, detection, and defense. *IEEE Communications Surveys & Tutorials*, 16(2):898–924, 2014. (Cited on page 8)

[23] M. Kührer, C. Rossow, and T. Holz. Paint it black: evaluating the effectiveness of malware blacklists. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 1–21, Gothenburg, Sweden, September 2014. (Cited on page 4)

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. (Cited on page 12)

[25] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. (Cited on page 21)

[26] Michael A. Lones. How to avoid machine learning pitfalls: a guide for academic researchers. *CoRR*, abs/2108.02497, 2021. (Cited on page 31)

[27] Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph M. Hellerstein. The case for a hybrid p2p search infrastructure. In *Peer-to-Peer Systems III: Third International Workshop, IPTPS 2004, La Jolla, CA, USA, February 26-27, 2004, Revised Selected Papers*, pages 141–150. Springer Berlin Heidelberg, 2005. (Cited on page 8)

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. (Cited on page 26)

[29] Juhong Namgung, Siwoon Son, and Yang-Sae Moon. Efficient deep learning models for dga domain detection. *Security and Communication Networks*, 2021:1–15, 01 2021. (Cited on pages 9, 21, 23, and 34)

[30] Netlab. Netlab360 archive. `https://data.netlab.360.com/dga/`, 2022. Accessed: 12 July 2024. (Cited on page 47)

[31] G. Oikonomou and J. Mirkovic. Modeling human behavior for defense against flash-crowd attacks. In *2009 IEEE International Conference on Communications*, pages 1–6, Dresden, Germany, 2009. (Cited on page 8)

[32] Osunjio. Dga data full, 2020. (Cited on page 32)

[33] European Parliament and Council of the European Union. Regulation (EU) 2016/679 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). Official Journal of the European Union, L 119, 4.5.2016, pp. 1â88, 2016. (Cited on page 4)

[34] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018. (Cited on page 26)

[35] Vinayakumar Ravi, Soman Kp, Prabaharan Poornachandran, and Sachin Kumar S. Evaluating deep learning approaches to characterize and classify the dgas at scale. *Journal of Intelligent Fuzzy Systems*, 34:1265–1276, 03 2018. (Cited on pages 17, 21, and 29)

[36] Frank Rosenblatt. *Principles of Neurodynamics*. Spartan, New York, 1962. (Cited on page 19)

[37] Gerard Salton. *Automatic Information Organization and Retrieval*. McGraw-Hill, New York, NY, USA, 1968. (Cited on page 27)

[38] Pramod Singh. *Deploy Machine Learning Models to Production*. Apress, Berkeley, CA, 1 edition, 2020. (Cited on page 45)

[39] B. Stone-Gross, M. Cova, L. Cavallaro, and B. Gilbert. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 635–647, Chicago, IL, USA, November 2009. (Cited on page 4)

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 6000–6010. Curran Associates Inc., 2017. (Cited on pages 22 and 24)

[41] Harald Vranken and Hassan Alizadeh. Detection of dga-generated domain names with tf-idf. *Electronics*, 11(3), 2022. (Cited on pages 17, 26, and 29)

[42] Jonathan Woodbridge, Hyrum S. Anderson, Anjum Ahuja, and Daniel Grant. Predicting domain generation algorithms with long short-term memory networks, 2016. (Cited on pages 17, 21, and 29)

[43] S. Yadav, A. K. K. Reddy, A. L. N. Reddy, and S. Ranjan. Detecting algorithmically generated malicious domain names. In *10th ACM SIGCOMM Conference on Internet Measurement*, pages 48–61, 2010. (Cited on page 17)

[44] Bin Yu, Jie Pan, Jiaming Hu, Anderson Nascimento, and Martine De Cock. Character level based detection of dga domain names. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2018. (Cited on page 21)

[45] Mattia Zago, Manuel Perez, and Gregorio Martinez Perez. Scalable detection of botnets based on dga: Efficient feature discovery process in machine learning techniques. *Soft Computing*, 24, 01 2019. (Cited on page 17)

[46] Mattia Zago, Manuel Gil Perez, and Gregorio Martinez Perez. UMUDGA - University of Murcia Domain Generation Algorithm Dataset, 2020. (Cited on page 47)

[47] Feng Zeng, Shuo Chang, and Xiaochuan Wan. Classification for dga-based malicious domain names with deep learning architectures. *International Journal of Intelligent Information Systems*, 6(6):67–71, 2017. (Cited on page 21)