Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Detection of network attacks using ensemble learning methods

Bachelor's Thesis

Krisztián Adrián Volentér

Advisor
Dr. András Gergely Mészáros

December 8, 2022

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Volentér Krisztián Adrián*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2022. december 8.

Volentér Krisztián Adrián
hallgató

Description of the task

The deflection of network attacks is a central question of modern computer networks. An important element of this process is performed by the Intrusion Detection System (IDS). A fundamental weakness of classic, signature based IDS solutions is that their application against unknown attacks is limited. Anomaly based IDSs (AIDS) offer an answer to this problem, which is based on the detection of abnormal traffic patterns. AIDSs are often based on machine learning methods. Among them are the ensemble based methods, which are widely popular due to their robustness and high performance both in IDSs and in other fields. The task of the student is to recommend and implement an ensemble based machine learning method which can be used for the efficient detection of network attacks.

As part of the above task, the student has to solve the following subtasks:

- Introduce the most important ensemble methods and give a short summary of the literature of ensemble methods proposed for intrusion detection and the network datasets used for their testing.
- Propose and implement an ensemble based method for intrusion detection.
- Choose and preprocess datasets that can be used to sufficiently test the intrusion detection method.
- Test the proposed intrusion detection method and evaluate it by comparing its performance to other solutions.
- Document the work.

Összefoglaló

Az egyre komplexebb és letisztultabb támadások miatt az illetéktelen hálózati behatolások észlelése egyre fontosabb szerepet tölt be napjainkban. A támadóknak több eszköz áll rendelkezésükre, mint a cégek hálózatának biztonságával foglalkozó alkalmazottaknak. A hálózatok ellen irányuló támadások egyik rejtett veszélye, hogy sok esetben bárki által kivitelezhetőek, mert az egyes támadó eszközök nem feltétlenül igényelnek mélyreható szakmai tudást. A folyamatban lévő támadások időben való észlelése és megakadályozása elengedhetetlen fontosságú a hálózatok alapszintű védelméhez. Az IDS-ek (Intrusion Detection Systems, magyarul: Illetéktelen hálózati behatolást jelző rendszerek) képesek észlelni az egy-egy hoszton, vagy az egész hálózaton történő rosszindulatú eseményeket. Ebben a dolgozatban az anomália-alapú IDS-ekkel kísérleteztem, mivel forradalmi megoldást kínálnak a gépi tanulás támadások észlelésére való felhasználására. A szignatúra-alapú módszerekkel ellentétben ezeknél nincs szükség előre összeállított szabályrendszerre, amelyek mentén a normális viselkedés meg van határozva. Egy másik előnyük, hogy képesek detektálni a zero-day támadásokat is, amire az előbbiek nem alkalmasak. Dolgozatomban be szeretném mutatni a gépi tanulás ilyen környezetben való felhasználását. Különböző klasszifikációs modelleket implementáltam számos ensemble alapú módszer bemutatására és összehasonlítására. Az ehhez felhasznált gépi tanuló módszerek a Random Forest, AdaBoost és az XGBoost voltak. Az eredményeim további árnyalására ad lehetőséget a szintén bemutatott, egy döntési fán alapuló, módszerem. A realisztikus hálózati környezet szimulálására a CICIDS2017 adatsort használtam, amely sok, változatos karakterisztikájú támadást tartalmaz. A modellek igényeihez igazodván szükségessé válik az adatok előfeldolgozása is. Ezen folyamat segítségével a modelljeim futási idejét is csökkentettem. A munkám során szembesültem az előbb említett adatszet bizonyos hibáival. A javított változatának felhasználásával készített modellekel bemutatom, hogy az általam használt gépi tanuló algoritmusok szignifikáns javulást mutatnak az eredeti adaton végzett mérésekhez képest. Ez az eredmény még jobban legitimálja az anomália-alapú IDS-ek használatát.

Abstract

Intrusion detection on networks is gaining importance, which is a result of the ever-evolving sophistication of attacks. Today, attackers have more tools for compromising the network of a company than an employee has for defending it. Also, attacking tools might be used successfully with only a low amount of knowledge about the basis of attacks. The defenders always have to be aware of any ongoing attacks on the network and try to mitigate them. IDSs (Intrusion Detection Systems) come as great help in these defensive approaches as they can detect malicious behaviour on hosts or on the network as well. In this thesis, I experimented with the anomaly-based IDSs, which grant a revolutionary way of using machine learning methods for the detection of attacks. They can be used without exact rules to be predetermined unlike in the case of the signature-based approach. Their benefits include the possibility of detecting zero-day attacks, which cannot be detected by signature-based IDSs. In the thesis, I would like to show the power of machine learning for this use-case. I have implemented several classification engines based on ensemble learning models, which are Random Forest, AdaBoost, and XGBoost. I also presented the performance of a single Decision Tree as a comparison, which shows the improvements ensemble learning makes. To simulate a real network environment I used the CICIDS2017 dataset as it contains various types of attacks to demonstrate the classification success of my models. Datasets usually need to be preprocessed in order to fit the preferences of the chosen model(s), moreover, I used it to reduce the time complexity of my models. Later in my work, I got to know some shortcomings of this dataset and I proposed other models using a corrected dataset. The results on it showed a significant improvement, which legitimates the use of anomaly-based IDSs further.

Chapter 1

Introduction

This thesis is influenced by the conditions of the current fast-growing technological environment, where we face newer and newer difficulties, when trying to protect our devices and personal data from intruders, e.g. as individuals or companies. The race in the IT field forces the developers to work in a hurry, leaving vulnerabilities to be found by a friend or a foe, because products must enter the market as quickly as possible. The growth of IoT market share gave another attack surface for intruders, because these devices tend to have low security measurements built in. To improve our chances against cyber criminals, intrusion detection has become a popular topic in the last decades.

Intrusion is a kind of security incident, when the attacker exploits software or system vulnerabilities and gains access to the system or escalates privilege without permission of the authorization policy. An Intrusion Detection System (IDS) [5] is defined as a hardware or a software, which looks for potential threats in network traffic and is able to respond to them, while reporting them to a supervisor. In cyber security systems, we use it alongside firewalls and antiviruses to make a highly efficient combination of defense.

Although IDSs are the products, they differ in some aspects of implementation. The first aspect is the network location where we use it. When we want to monitor processes and file activities associated with a specific host, we use HIDSs (host-based IDSs) and when we want to monitor the traffic through network devices, we use NIDSs (network-based IDSs). IDSs are sending alarms to the operator in case of an attack – as they are for the detection of an attack – but there are IPSs (Intrusion Prevention Systems), which are also blocking traffic based on malicious characteristics.

The core of the system which makes the decisions is another differentiator. In this case there are three variations: AIDS (anomaly-based IDS), MIDS (misuse-based IDS), and hybrid IDS. The misuse- (or signature-) based technique uses the signatures of known attacks for detection. The main disadvantage of this method is, that it detects only previously known attacks, however, with frequent updates on the attack database it can be a legitimate solution. It is the current industrial standard, when it comes to using these kinds of systems. In my work, I experimented with the anomaly-based method, which models the normal network traffic and detects the possible attacks as anomalies which are deviations from the benign behavior. This technique has a lot of potential because it can detect zero-day attacks, while making signatures of them for misuse-based databases. Its main drawback is the high number of false positive cases.

The focus of this thesis is on measuring whether ensemble methods are viable solutions for intrusion detection. My goal is to get accurate predictions on new attacks with keeping

the false negative cases as low as it is possible. The above mentioned false positive cases will be considered with lower priority.

The methodology used for this thesis is outlined in Chapter 2. Datasets are significant parts of IDS researching, thus I will detail the use of them in Chapter 3. The main dataset for this thesis will be CICIDS2017 [59]. In addition, I will detail its errors and the proposed dataset, which makes corrections on it. The preprocessing of CICIDS2017 will be described in Chapter 4 with the theoretical background of this procedure. In Chapter 5, I will specify the metrics used for the evaluation of classifiers and then I will measure the performance of a single Decision Tree and of several chosen ensemble methods with them on CICIDS2017 dataset. I will use similar approach to the later outlined methods in Chapter 4 and Chapter 5 for the improved dataset in Chapter 6. Lastly, I will summarize my work and make recommendations based on my results in Chapter 7.

Chapter 2

Methodology

In this chapter, I would like to summarize the methodology used for this thesis. As I have mentioned earlier, I used machine learning methods as the predictive core of the IDS. Notedly, I have implemented only the detection engine, as the main focus of my project is anomaly detection.

The data had to be concatenated and analyzed in the first place to get a deeper understanding. This included, e.g. getting a grip on the data types in different columns. Based on this information and knowing the dimensionality of the dataset, I could tell what preprocessing steps might be required.

The next step was the actual preprocessing procedure, which is highly dependent on the behaviour of the chosen classifier: in our case Decision Tree and ensemble methods are used, so e.g. normalization is not needed because they are all based on trees and the construction of trees is insensitive to this kind of alteration. The dataset should also be split into pieces representing training, validation and testing segments. Training dataset is for teaching the given model with the patterns of the dataset. Also, fine-tuning of hyperparameters of the ML model can be achieved with its repeated training on the training set – or on a subset of it (referred to as fine-tuning dataset in the following) – and evaluation on validation dataset with the constant changing of the parameter set in each step. This process will be further detailed in Section 5.2.

Before any fine-tuning and evaluation happened on the train subset, I had to use sampling because of the imbalance of the dataset (see Section 4.3). Feature elimination is also required because there are too many columns and they are just slowing down further procedures, while not making evaluations more accurate. The used procedure for feature selection and elimination is described in Section 4.4.

After each preprocessing and fine-tuning part, the learning process of the classifier can be measured by the validation subset. This process has great importance because machine learning algorithms can be overfitted or underfitted. An overfitted ML model performs well on the training data but generally poorly on the validation subset. A classifier is underfitted if it achieves poor scores either on training and validation data. After achieving a predetermined goal on the validation dataset, the test data were evaluated by the model. This predetermined goal can be, e.g. to reduce misclassifications to an extent and/or to reduce evaluation time. The validation set was used for checking the outcome of each preprocessing step. After preprocessing, the validation data were used to validate classifiers with fine-tuned parameters. The test set was only used for the final evaluation in each case.

In gereral, if the model works well for the use-case, then further adjustments are not necessarily needed. Otherwise, new settings should be tried on the classifier starting again from the end of preprocessing. If neither of these helps, a new model should be tried with the same methodology. The visualization of the descripted process is shown in Figure 2.1.

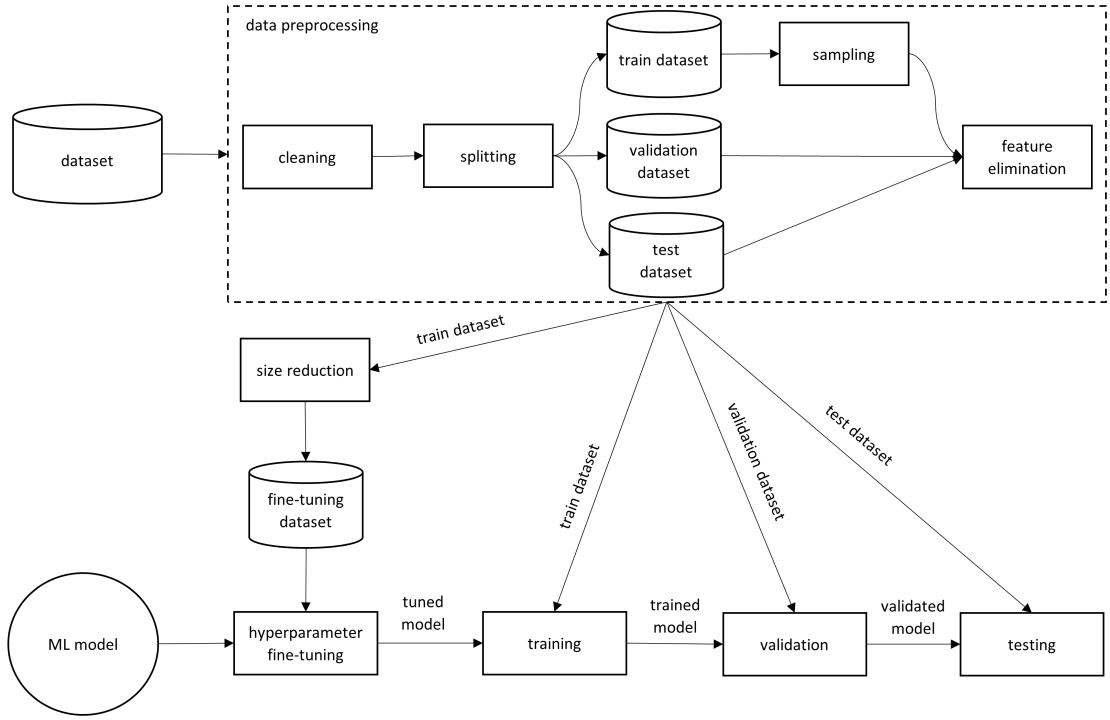


Figure 2.1: The visulization of the proposed methodology

2.1 Tools and platforms used

The main environment I used for this thesis was Jupyter Lab [26]. It is a powerful tool for data science and machine learning because it can run multiple IPython files (ipynb extension) in different kernels. It grants opportunity to work with several file types. It is also modular and can be fully customized, which gives the possibility to make extensions for it. I used several of them, e.g. the GitHub extension[1], which proved to be helpful in version control. This way I could share my weekly process with my advisor. Also, I developed an extension[2] for the platform using TypeScript and React, which can run IPython cells by a specific tag. The layout of this extension can be seen in Figure 2.2.

Using IPython makes work easier, because of its cellular design. Every cell can be executed, stopped and controlled separately. Their usage can be extended inside Jupyter Lab by 'magic functions', e.g. %time can measure the run time of a line and %%time written at the start of a cell can measure the run time of the whole cell. Also, this cellular design lets the users make the code more readable by summarizing it into blocks. IPYNB files let developers make, e.g. Markdown cells, which can provide description for the cells below and help to get a clearer interpretation of the workflow.

[1] Jupyter Lab, Git extension `https://github.com/jupyterlab/jupyterlab-git` (Accessed: 2022-12-03)

[2] Jupyter Lab, Select by Tag extension `https://github.com/volenterk/jupyterlab_selectbytag` (Accessed: 2022-12-03)

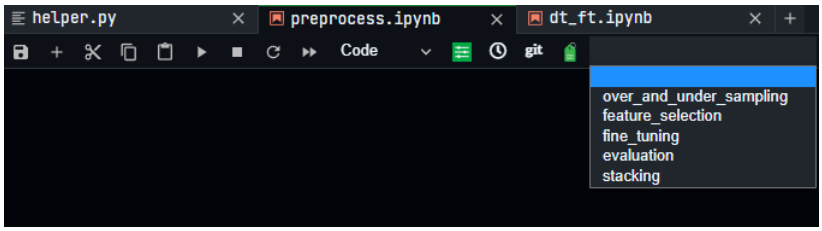


Figure 2.2: The dropdown can be used to select the tag – by its name – and the button on its left side selects and runs those cells, which have the tag

I worked in 2 separate Jupyter Lab environments. One of them was installed on Windows 10, and it was used for CPU intensive processes. The other one was set up inside WSL 2 (Windows Subsystem for Linux version 2) within the Windows 10 host. It was required because I used GPU acceleration for one of the models and the GPU could be accessed within WSL.

2.1.1 Used packages

There are several standard packages that are included in almost every machine learning project, which are the following: numpy, pandas and matplotlib. Numpy [35] is a mathematical library containing different efficient arrays and functions. Pandas [1] is famous for its data structures (`Series` and `Dataframe`) and the ability to widely access and modify them. Matplotlib [31] is a standard visualization tool, which can be used to plot, e.g. 2 and 3 dimensional scatter plots or bar plots. The more detailed description of these standard packages is not in the scope of this thesis.

2.1.1.1 Machine learning packages

For machine learning libraries, there is a high variety to choose from. There are packages, which require more in depth knowledge from the users. On the other hand, there are ones which are preferred by the beginners of this field. I chose to work mainly with scikit-learn [57] as it is a useful package for users of any level of expertise written in Python. It is also widely used within IDS related papers. I consider this as a beginner friendly package, which is due to the simplistic approaches its developers followed, its detailed API and code examples provided for the methods within. It contains techniques for almost all parts of machine learning, including e.g. data preparation, data manipulation and implementations of machine learning models. The used models from this package were `DecisionTreeClassifier` (see Section 5.3), `RandomForestClassifier` (see Section 5.4) and `AdaBoostClassifier` (see Section 5.5). I also used `GridSearchCV` (see Section 5.2) from scikit-learn and some other less important classes and functions, which are not specified here.

Another machine learning package, which was used in my thesis is xgboost [69]. It was made by DMLC (Distributed (Deep) Machine Learning Community) and is an optimized library for gradient boosting. From this package, I used the `XGBClassifier` model, which is an efficient and versatile model. It contains a full set of tools for machine learning with a high level approach. It can be customized to a great extent, which makes it usable in different scenarios. This package also has efficient matrices to store the data and can utilize GPU computation, while scikit-learn only supports CPU.

The last machine learning package I used was imbalanced-learn [24], which provides methods for better classification of imbalanced datasets. Several of these methods are to manipulate datasets but there are also machine learning models from scikit-learn, extended with the controlling of imbalance. From this package I used `Pipeline`, `SMOTE` and `RandomUnderSampler`. The imbalanced-learn and xgboost packages follow the API conventions in scikit-learn, which make them more easy to understand for beginners.

It is important to note that I used the parameter names available in the packages when describing these classifiers (see Chapter 5) and this was a purposeful choice. If the notation of the parameter was a letter from the greek alphabet, I used the written out form of it, e.g. I used `lambda` instead of λ. Sometimes I used the parameter names next to pure mathematical notations, which was also a purposeful choice because I wanted to present these models as simply as possible.

2.1.1.2 Visualization packages

The visualization of the different phases makes the results more digestible. For this purpose I tried and used several useful packages over matplotlib, which was mentioned previously. One of them was seaborn [3], which makes statistical visualization easier than matplotlib – which is one of its dependencies – with out-of-the-box solutions. It includes several more plot types, which provide more impressive visualization tools.

Another visualization package used in this thesis is umap-learn [68]. UMAP is the short for Uniform Manifold Approximation and Projection for Dimension Reduction. I used this package to visualize approximate relations between categories in my datasets. It is an interesting – and also an extensively researched – topic to visualize higher dimension of data into only 2 dimensions. There are also multiple alternative approaches for this problem, however, I chose this because of its promising view in data science. It is also a novel method, which aims to provide a faster alternative for t-SNE (t-Distributed Stochastic Neighbor Embedding) and PCA (Principal Component Analysis) [32].

The last visualization package used was Plotly, more precisely Plotly Express [39]. While the former is a package with highly customizable plots, Plotly Express provides an easier API to it. It automates the generation of several plot types by providing only the sufficient inputs. Users can efficiently make informative plots with it.

Chapter 3

Datasets

IDS researching would be impossible without publicly available datasets. For different use-cases, universities and organizations made comprehensive databases freely accessible, which come as great help. The terminology used for the structure of datasets is the following: the columns are called features and the rows are referred to as samples. A feature is an aspect of a sample, which can be numeric (for example received bytes), or categorical (for example the used protocol). A sample of a dataset is an entity, that can be fully or partially described by the features of the dataset. The features can be abstracted if the authors made steps with a preconception so that a complex feature helps summarizing raw columns and makes detection easier.

A drawback of the popular IDS related datasets is the high number of samples, which causes a problem when comparing related papers because authors may/will use different subsets. The reason is that cloud-based 'unlimited' resources are not cheap to access. The other problem is that – according to my experience with reading IDS related papers and replicating them in the previous semesters – subsetting can cause unlikely scenarios, so unfortunately the table can be tilted for better results. Although the size of the datasets would not be a problem in itself because real-life scenarios require as much relevant data as it is possible to gather.

For this thesis, I chose the CICIDS2017 [59] dataset as it is widely used by researchers of intrusion detection. To highlight the benefits of the dataset, I would like to introduce a dataset that I have used in my semester project which is NSL-KDD [34]. But first – to present the NSL-KDD dataset – KDD99 [27] should be described, which is its predecessor.

3.1 KDD99

This dataset was made from the well known DARPA dataset – which is a raw TCP/IP dump – by extracting features from it. The data was created for a challenge called KDD Cup in 1999, and it is the most cited data in the field of IDS research. It contains 4 weeks of network captures from the DARPA dataset. During the construction of the network captures, there were attack-free periods and periods when attacks occurred in the network. It has features derived from pcap and some additional ones abstracted with the thorough analysis of the raw packet capture. Although it is widely used, it has downsides such as duplicate rows, which are not containing new information and are biasing the evaluation. 70% of the training and 75% of the test data is redundant based on the analysis done by Tavallaee et al. [65]. Another drawback of this dataset is the unrealistically high number

of attacks compared to benign ones. Although it is outdated by now, the dataset was the pioneer of IDS related datasets.

3.2 NSL-KDD

As mentioned, the NSL-KDD datasets were created from selected records of KDD99 (see Section 3.1) providing more realistic measurement results. Redundant samples were removed from the original KDD99 dataset and the number of samples were decreased to provide more practical data [65]. It has separate datasets for training and testing purposes. In my semester project I mainly used the training segment to evaluate machine learning algorithms because there is a high deviation between the training and testing datasets. There are 14 new attack types compared to training [38]. As I have mentioned in Chapter 1, the idea behind anomaly-based models is to detect novel attack types but I found this disparity to be overwhelming for the classifiers I used (`DBSCAN` and `DecisionTreeClassifier`). Another problem is that it contains out-of-date attacks, which is because the original TCP/IP dump was made in 1999. These attacks are shown in Table 3.1.

Dataset	Attack types
train	back, buffer_overflow, ftp_write, guess_passwd, imap, ipsweep, land, loadmodule, multihop, neptune, nmap, normal, perl, phf, pod, portsweep, rootkit, satan, smurf, spy, teardrop, warezclient, warezmaster
test	apache2, back, buffer_overflow, ftp_write, guess_passwd, httptunnel, imap, ipsweep, land, loadmodule, mailbomb, mscan, multihop, named, neptune, nmap, normal, perl, phf, pod, portsweep, processtable, ps, rootkit, saint, satan, sendmail, smurf, snmpgetattack, snmpguess, sqlattack, teardrop, udpstorm, warezmaster, worm, xlock, xsnoop, xterm

Table 3.1: The attack types in NSL-KDD train and test datasets

I categorized these attacks in order to make results easier to interpret. 5 groupings could be made which are the following: Normal, DoS, Probe, U2R, and R2L. Denial of Service (DoS) attacks are covered in Section 3.3.3. Probe attacks are for the reconnaissance of the network, which includes sending network scanning or security testing packets to get to know the weakness(es) of the IDS. When intruders gain access to a remote system as user in a shell or telnet session and then achieving super user permissions is called User to Root (U2R). Gaining unauthorized access to a server is Remote to Local (R2L). Last but not least, in the Normal group I have summarized benign samples. The methodology I used for NSL-KDD was to concatenate the Normal group to each attack group. This transforms the multiclass classification into 4 binary classification tasks. The resulting datasets were evaluated separately. The classifiers showed good performance on the validation data with Decision Tree, however, because of the amount of new attack types, this classifier was not successful on the test data. The DBSCAN performed reasonably only on the validation set of the DoS grouping.

3.3 CICIDS2017

The Canadian Institute for Cybersecurity (CIC) released this dataset in 2017 for intrusion detection purposes [59]. It was produced from pcap files recorded over 5 days with using their network attack analysis tool named CICFlowMeter[1]. This tool works by replaying the captured traffic and then extracting flow-specific features out of it. Several criteria were used in order to create a valuable comprehensive dataset [18]. It was recorded on a real-life based topology, containing a variety of operating systems and network devices. The testbed architecture is shown in Figure 3.1. This dataset contains benign and attacking behaviour collected from victim-side and attacker-side traffic and stored on a dedicated storage server. The dataset is labelled, which makes this dataset available for supervised learning. The communications are present within and between internal LAN. Internet communication is represented as well by using common protocols: HTTP, HTTPS, FTP, SSH, and email protocols. The dataset contains common attacks based on a 2016 McAfee report, which will be detailed later. In the attacking periods, memory dumps and system calls were logged as well. The resulting dataset has more than 2.8 million samples and 80 flow features.

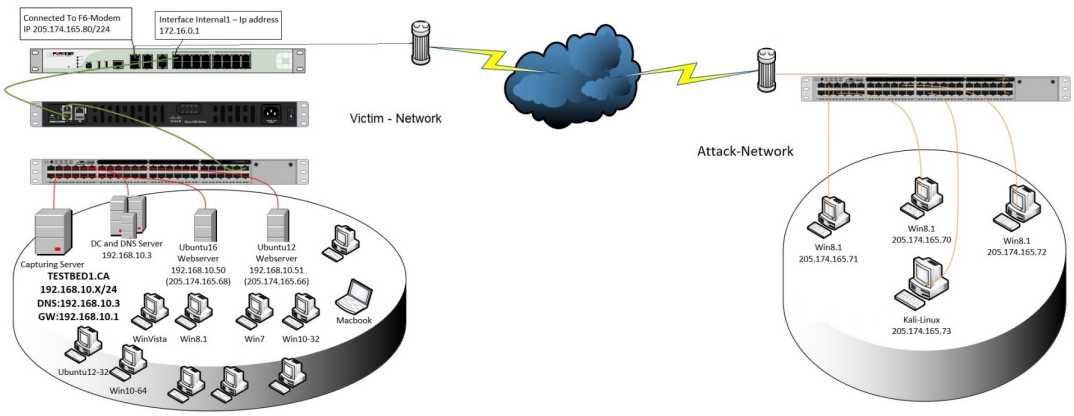


Figure 3.1: The testbed architecture for CICIDS2017 dataset [59]

The main reason behind choosing the CICIDS2017 over NSL-KDD is that I wanted to introduce and use a more recent dataset in this thesis, which is also popular amongst researchers of IDSs. There are 8 years between the two datasets, which results in CICIDS2017 having more modern attack scenarios. Also, CICIDS2017 consists of several pcap files (see Section 4), from which the researchers can choose to include in their work. On the other hand, NSL-KDD has 2 separate datasets for training and testing purposes, which constrains its usage if we use it as it is intended. The training and testing sets of NSL-KDD have high deviation for the used attack types, which results in poor scores and provides a less accurate (i.e., more pessimistic) attack scenario than CICIDS2017. Moreover, CICIDS2017 has more rows than NSL-KDD, which is good for training models accurately as some datasets lack a sufficient amount of samples. Besides the promising features of CICIDS2017 dataset there are also downsides, which will be further detailed in Section 3.4.

In the following sections I will write about the categories within the CICIDS2017 dataset with the purpose of detailing the actual threats which were used by the creators of the

[1]The original CICFlowMeter GitHub repository `https://github.com/ahlashkari/CICFlowMeter` (Accessed: 2022-12-03)

dataset. It is not usual for IDS related papers to conclude details like these, however, I think that readers – and also researchers – should know about the different behaviours represented in the data for a deeper understanding. I am also dedicating these sections to my personal interest in cybersecurity. The tables (e.g. Table 3.2) contain the occurrences of the attacks – based on the time intervals provided by the creators – within the recording of the dataset, source, victim, and the type in each section [25]. In Figure 3.2 a UMAP 2D plot can be seen consisting of 100000 samples from the dataset made after the preprocessing stage (see Chapter 4). Based on this figure the attack categories seem to separate well, which usually indicates that the classification outcome will be good. It is important to note that this is just an approximation of the high dimensional data.

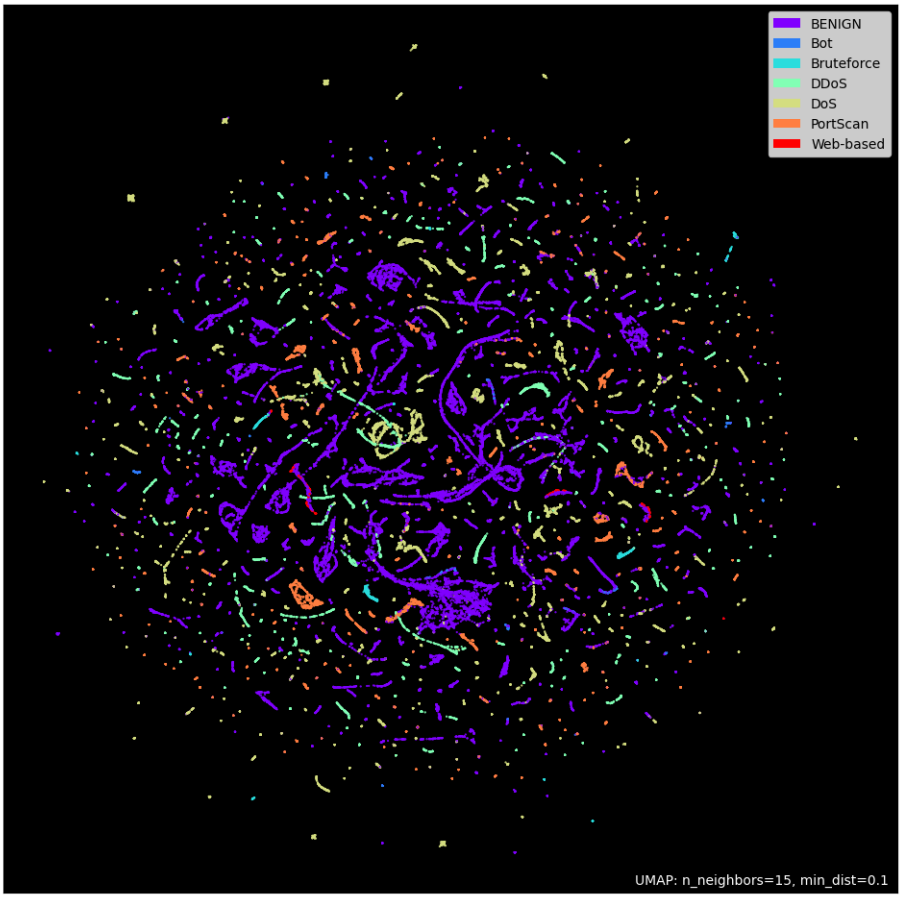


Figure 3.2: The 2 dimensional plot of the CICIDS2017 dataset

3.3.1 Benign

Benign category represents the normal behaviour in IDS datasets. In this case the samples were generated with a system called B-Profile [59]. It mimics human interactions with using machine learning and statistical analysis as it is claimed in the work of Sharafaldin [58] et al. Benign behavior was generated and captured one whole day, July 3, 2017.

3.3.2 Brute force

Brute force is otherwise known as exhaustive key search, where the attacker tries all possible passwords or passphrases to get access to a service using trial-and-error. This

attack can be done in multiple fields, e.g. hash cracking, login form, and service brute forcing. It can be done on services, e.g. by rule-based passwords, general and personalized wordlists. The automated probings are targeted (Table 3.2), meaning that there is only one host and one service at a time on which the attack is performed.

Date	Attacker	Victim	Attack type
July 4, 2017 9:20-10:20 a.m.	Kali	Ubuntu16 WS	FTP-Patator
July 4, 2017 14:00-15:00 p.m.	Kali	Ubuntu16 WS	SSH-Pataror

Table 3.2: Brute force attacks

3.3.2.1 Patator

Patator [37] is a multi-threaded tool written in Python. It aims to be more reliable and flexible than Hydra, Medusa, Ncrack, Metasploit modules, and Nmap NSE scripts. The modular design supports users to write their own brute force segments, though it has more than 30 methods built-in. The creators of the dataset used ftp_login module, which can brute force the credentials of an FTP service and ssh_login, which does the same but for an SSH service.

3.3.3 DoS/DDoS

Denial of Service (DoS) [40] attacks are making a service inaccessible with flooding or sending information to it that triggers a crash. Nowadays, attacks sending huge loads of data from a single point are not effective due to the modern load balancing procedures and improved hardware capabilities on network devices. A more effective strategy is to let multiple systems send malicious traffic to the same target. These attacks are referred to as Distributed Denial of Service (DDoS). The mitigation of these attacks is usually not developed enough to stop malicious actors from making a service outage, however, there are great methods and techniques for defensive purposes today. Nonetheless, it is a fact that attackers have more tools to disrupt than defenders have to prevent, meaning that all possible countermeasures should be considered while trying to make our networks more robust to these attacks. A fundamental approach is to test our systems with the eyes of a hacker by using tools to see how big the load is that our devices can handle before the point of crisis. In my work a theoretical IDS tries to classify these attacks by flow descriptive features. The basic procedure could be to block source IPs when a DoS or DDoS flow happens. This is a good solution for simpler cases when the attacking sources – based on IP addresses – stay the same during attacks. However, i.e. against a vast botnet, which constantly changes attacking pool, this method alone will not be sufficient. The DoS and DDoS attack periods are listed in Table 3.3.

Date	Attacker	Victim	Attack type
July 5, 2017 9:47–10:10 a.m.	Kali	Ubuntu16 WS	DoS Slowloris
July 5, 2017 10:14–10:35 a.m.	Kali	Ubuntu16 WS	DoS slowhttptest
July 5, 2017 10:43–11:00 a.m.	Kali	Ubuntu16 WS	DoS Hulk
July 5, 2017 11:10–11:23 a.m.	Kali	Ubuntu16 WS	DoS GoldenEye
July 5, 2017 14:00-15:00 p.m.	3xWin 8.1	Ubuntu16 WS	DDoS LOIC

Table 3.3: DoS and DDoS attacks

3.3.3.1 Slowloris

Slowloris [62] is a protocol-based exploitation, which was initially released 17 June 2009. It is a very sophisticated attack because one could cause harm without huge loads of traffic [60]. When browsing, we send GET requests (HTTP protocol) to get resources from the webserver to load on our client side. The brilliant trick comes into play when we think about the closing of one request: we have to put two new line characters to the end of a request. The creator(s) of the original attack exploited this behaviour by making hundreds of GET requests to randomized endpoints of the target, while not closing them. An endpoint is e.g. /index.html of the target webserver, which usually contains the home page. In this case `s.send_line(f"GET /?{random.randint(0, 2000)} HTTP/1.1")` was used for the requests. The method constantly sends random numbers on each started request as some sort of a keep-alive signal for the server to not close the connection. It also uses a random pool of UAs (User-Agents), which makes this attack more hidden. When a thread is freed, Slowloris tries to get that thread too. This attack targets mostly Apache webservers because these servers allocated new threads for each request by design. Apache webservers are quite common to these days based on Shodan statistics[2], meaning that this attack can be present nowadays when the Apache server uses mods that are not thread safe. It would be a relatively good fix for the problem to limit the maximum number of threads for an IP address. A simple Python implementation can be found on GitHub which is straightforward to use but it has only a few parameters to set.

3.3.3.2 slowhttptest

The slowhttptest tool [61] contains multiple DoS attacks via prolonged HTTP connections. It has 4 test modes, which are Slowloris (default), R-U-Dead-Yet, Apache killer, and Slow Read. As the paper does not clarify the mode used for this tool, I presume that Slowloris was used again. However, it would not make that much of a difference – in my opinion – to make separate classes for them, but the power of this tool is that it is highly configurable. For example, the number of connections, interval between keep-alive data, and even proxy options are accessible for the users.

3.3.3.3 HULK and GoldenEye

HTTP Unbearable Load King (HULK) [23] is a tool to produce random unique requests and send them in a big load to the targeted webserver making it inaccessible for legitimate users. It obfuscates the source client with setting illegitimate user agent values [20]. The referrer is also modified to major sites. To bypass Content Delivery Networks (CDNs), this method uses the no-cache option for the Cache-Control HTTP header field. This forces the server to complete the request without CDN caching mechanisms. The original tool was written in Python with giving separate thread for each request. A GitHub repository can be found with a GoLang re-implementation, which uses goroutines to make better use of the resources.

GoldenEye [60] works almost the same way as HULK. It uses randomized parameters for the requests and also bypasses CDNs. Basically, these two attacks are exploiting keep-alive and no-cache, when making requests.

[2]Shodan query for Apache web servers `https://www.shodan.io/search?query=Apache+httpd` (Accessed: 2022-12-03)

3.3.3.4 LOIC

Low Orbit Ion Cannon (LOIC)[3] is a binary written in C#. It also has a JavaScript implementation which enables clients to run attacks from their browsers. It sends TCP, UDP or HTTP packets to the target webserver [33]. However, it is found that one attacker is not enough for an attack to succeed. This is why it is used in DDoS context here too. It was originally deployed by the Anonymous hacktivist group in Project Chanology, where people created a botnet with volunteers to attack the targeted infrastructure with this tool.

3.3.4 Web attacks

Web attacks [28] are high variety threats against websites. These attacks occur on a daily basis because the popularity of web applications is on the rise. This comes hand in hand with ever evolving attacking possibilities, which can be exploited by cybercriminals. There are also tools that even a beginner could use for successful attacks. The terrifying fact is that even web apps of tech giants can be successfully exploited[4], in which average people put their trust. That is why big game individuals or cybercriminal groups might preferably target these bigger systems. Another example can be E-Commerce, which is a financially beneficial target. However, web attacks can also result in data breaches, which means that private data gets leaked. This can cause and is causing real threat, e.g. to social media platforms, where people share lots of their personal data. The goal of this thesis is to detect these attacks, however, I think that relying on a flow-based IDS to prevent companies from web attacks must be a last resort. The majority of attacks can be mitigated with writing secure web apps.

The attacks were done on Damn Vulnerable Web App (DVWA) [14] with automation. This app was made for cybersecurity professionals to test their skills and for developers to help them understand the securing of web applications better. The process of making attacks is not detailed enough in the article of Sharafaldin et al. [59], thus I will try to conclude these attacks in general and how they could be leveraged in the app. The DVWA has 4 security levels (namingly easy, medium, hard, and impossible) to choose from; for the sake of illustration, I will only describe the easy level. These levels stands for the difficulty of exploiting a vulnerability. The attack schedule for this category is shown in Table 3.4.

Date	Attacker	Victim	Attack type
July 6, 2017 9:20-10:00 a.m.	Kali	Ubuntu16 WS	Web Brute Force
July 6, 2017 10:15-10:35 a.m.	Kali	Ubuntu16 WS	Web XSS
July 6, 2017 10:40-10:42 a.m.	Kali	Ubuntu16 WS	Web SQLi

Table 3.4: Web-based attacks

3.3.4.1 Brute force

The fundamentals of brute force attacks were covered in Section 3.3.2. In DVWA a login form can be bypassed with guessing the password of the "admin" user. This can be done

[3]LOIC SourceForge page `https://sourceforge.net/projects/loic/` (Accessed: 2022-12-04)

[4]World's biggest data breaches `https://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/` (Accessed: 2022-12-04)

with several tools, e.g. with Hydra. An optimal mitigation strategy would be to make a limit for maximum password tries per session and to use key derivation functions on passwords, e.g. Argon2 [8].

3.3.4.2 XSS

Cross-Site Scripting (XSS) vulnerabilities let attackers execute Javascript code in the browser of the target(s). They contain a wide variety of attacks, which can be categorized into 3 groups: stored-, reflected-, and DOM-based XSS. Stored XSS has the biggest impact of them because the result of the JavaScript code will be stored on the backend server [2]. This means that every visitor will have the injected code running when navigating to the affected endpoint. Having posts and comments makes usually the site vulnerable to this attack. The website is vulnerable to reflected XSS if the used exploit payload gets executed by the backend but it is not a persistent change on the behaviour of the web app [41]. DOM-based (Document Object Model-based) XSS vulnerability is also not persistent but it is processed on the frontend. It can be used for phishing because the domain name of the site will seem secure, however, the url parameters and anchors will be malicious to the user. The majority of XSS attacks could be prevented with, e.g. input and output sanitization, setting certain security headers within the server configuration, and using HTTPS across the entire domain. Out of the three, reflected and stored XSS vulnerabilities are exploitable within the DVWA.

3.3.4.3 SQLi

Web applications in general require a database to store data. To retrieve useful information from it, the site has to use queries real-time on their backend. There are several query languages to choose from but the most common is SQL (Structured Query Language), which works in several DBMS (Database Management System) architectures. Because websites are popular, attackers can more easily find vulnerabilities in the communication with the database. Some of these vulnerabilities are caused by, e.g. the lack of input sanitization and validation, which makes SQL injection possible. It is one of the most common exploitation techniques.

On the easy level of DVWA there is a searchbar for finding a user by its ID [41]. It is a common scenario in which the user input does not go through any checking procedures and is executed straight in the SQL query. Mitigation strategies are, e.g. the formerly mentioned input sanitzation and validation, using WAF (Web Application Firewall), and using a user for the database queries, which has as low privileges as possible.

3.3.5 Ares

A botnet is a network of hosts controlled by a central host [16]. The most common usage of them are DDoS attacks because a large botnet can cause a serious amount of traffic to a single or even to distributed servers. It can also be used, e.g. to get access to remote network or send spams. In this dataset it was used to get remote shell connection and data about the host in the botnet, e.g. logging keystrokes. Ares is a botnet framework implemented in Python. It implements two activities: the server and the agent. The server part implements the CNC (Command and Control) functionality, which is responsible to orchestrate agents[5]. The agent side runs on the compromised hosts and it is used to keep

[5]Ares GitHub repository `https://github.com/sweetsoftware/Ares` (Accessed: 2022-12-04)

the connection up between the CNC server and the client. The schedule for this attack is shown in Table 3.5.

Date	Attacker	Victim	Attack type
July 7, 2017 10:02-11:02 p.m.	Kali	Windows 7, 8.1, 10, Vista	Botnet Ares

Table 3.5: Botnet attack schedule

3.3.6 Port scan

Nmap is the most commonly used tool for different kinds of network scanning. It can, e.g. detect the hosts which are up, see the open ports on them and what might be serving on that port. I listed some of the benefits of using this tool but there is also a wide variety of switches for the customization of the scans. The NSE (Nmap Scripting Engine) gives another layer to this tool as it is able to execute scripts on the found targets, which might result in getting more information on a service or finding an easily detectable vulnerability or misconfiguration. The Nmap switches used[6] for creating malicious traffic are in Table 3.6 and the attack schedule is in Table 3.7. The description of the dataset does not provide a clear concept on what firewall rules were used. The paper mentions only that a Fortinet firewall was used [59].

Switch	Description
-sS	TCP SYN scan
-sT	TCP Connect() scan
-sF	TCP FIN scan
-sX	Xmas scan
-sN	TCP NULL scan
-sP	Skip port scan
-sV	Probe open ports to determine service/version info
-sU	UDP scan
-sO	IP protocol scan
-sA	TCP ACK scan
-sW	TCP Window scan
-sR	An alias to -sV
-sL	List scan
-b	FTP bounce scan

Table 3.6: The list of the used Nmap switches

Date	Attacker	Victim	Attack type
July 7, 2017 13:55-14:35 p.m.	Kali	Ubuntu16 WS	Port scan - Firewall rules on
July 7, 2017 14:51-15:29 p.m.	Kali	Ubuntu16 WS	Port scan - Firewall rule off

Table 3.7: The schedule of port scans

[6]Nmap brief man page `https://nmap.org/book/man-briefoptions.html` (Accessed: 2022-12-04)

3.3.7 Heartbleed

OpenSSL is an open-source cryptographic tool to be used for secured communication over networks. Heartbleed [22] is a widely known vulnerability discovered in the TLS heartbeat extension of this tool. It leaked memory contents from the server, which contained e.g. private keys and other valuable information (e.g. passwords). This vulnerability is referred to as buffer over-read[7]. The significance of this vulnerability is that it is still exploitable on some devices on the internet. The attack also has a website, which states that IDS/IPS devices can only detect but cannot block this attack separately. The attack schedule is shown in Table 3.8.

Date	Attacker	Victim	Attack type
July 5, 2017 15:12-15:32 p.m.	Kali	Ubuntu12 WS	Heartbleed

Table 3.8: The schedule of Heartbleed attacks

3.3.8 Infiltration

Infiltration can be harmful if the target system has outdated software(s). This attack type has multiple stages required for a successful attack. The first main goal is to place a malicious file on one of the corporate computers of the target organization. This can be achieved e.g. by sending a spam e-mail with malicious attachment or link to the malicious file. Then the file should be opened/executed along which the malicious activity takes place. The files were deployed on the systems in 2 various ways [59]. One of this methods was using an uploaded malicious file on Dropbox, the other approach used a USB stick to deploy the file on the computer. The malicious activities vary but during the creation of the dataset a backdoor was opened on the targeted computers. A backdoor grants unauthorized access to a private network with bypassing security mechanisms.

The infiltration attack schedule is listed in Table 3.9. The documentation of the dataset [59] does not specify any other attack that was executed on the network after gaining backdoor access for the first 2 rows in the table. However, with the last one a port scan attack was executed from the Windows Vista to the other network hosts.

Date	Attacker	Victim	Attack type
July 6, 2017 14:19-14:21 p.m.	Kali	Windows Vista	Infiltration - Dropbox
July 6, 2017 14:53-15:00 p.m.	Kali	Macbook	Infiltration - USB
July 6, 2017 15:04-15:45 p.m.	Kali	Windows Vista	Infiltration - Dropbox

Table 3.9: The schedule of Infiltration attacks

3.4 Errors in CICIDS2017

During my work on this thesis – after working with the CICIDS2017 dataset for one and a half months – I realized that this dataset might have some shortcomings. The first time I have noticed this was when I read the documentation of the dataset and the attacks were

[7]Buffer over-read vulnerability `https://cwe.mitre.org/data/definitions/126.html` (Accessed: 2022-12-04)

not detailed enough to get a concept of their execution. Also, some descriptions did not make sense and the naming of network devices was not consistent in some cases.

The problem is that papers properly criticizing this dataset have not been released in a notable number until recently but after I was directly searching for the errors of this dataset, a publication submitted on 12 September 2022 came up made by Lanvin et al. [29]. It is based on the errors found by two previous publications and provides an assumably better solution for the correction of the errors made in the dataset.

One of them was published in 2021 by Engelen et al. [15], which found errors in the pcap processing tool used by the original creators of the dataset, which is CICFlowMeter. The flaws detected by this work include, e.g. that the termination of TCP connections were falsely implemented, the time intervals of the attacks given by the creators were not precise, not executed attacks were labeled as attacks, some attacks were not used correctly, and many others. To answer the flaws found, they introduced their own, corrected CICFlowMeter[8].

The other one was published in 2022 by Rosay et al. [44], which found other flaws in the original tool including, e.g. wrong protocol detection, feature duplication and miscalculation. They proposed LycoSTand[9], which is a flow-based extractor. Further description of these two publications is not in the scope of this thesis.

Lanvin et al. chose to continue the work of Engelen et al. by finding more errors in the data retrieved from the corrected CICFlowMeter. This means that the tool developed by Rosay et al. might contain corrections of flaws that the former two works did not. Lanvin et al. found 4 additional errors in the corrected tool which are the following: errors in the creation of flow descriptions, incoherent timestamps, duplication in the network capture, and incorrect labeling of port scans that happened during infiltration (see Section 3.3.8). The first flaw occurs because the original CICIDS2017 was not sorted by timestamps and CICFlowMeter interprets TCP handshakes in the wrong order. This feature of reordering already got included in the corrected tool developed by Engelen et al. The second error is also in connection with TCP handshakes, more precisely with the example used in the paper: a SYN-ACK packet might have lower timestamps than a SYN packet, which results in the inversion of source and destination IP addresses. This observation could not be fixed because it could not be automated. The third error is the number of duplicates in the network capture, which takes up 4.5% of the packets each day. In one time interval of infiltration attacks detailed in Section 3.3.8, there was a second phase of the attack, which is a port scan on the other hosts from the controlled Windows Vista. These port scans were mislabeled and they came up with an automated correction for this. They propose 6 datasets[10] from the permutation of 3 alterations, which are the following: reordering of the network captures before using the tool (**R**), adding the new port scans (**P**), and keeping duplicate rows (**D**). The best permutation in my opinion is **RP**, which addresses all of the issues. I have also used 2 dimensional UMAP estimation on the first 100000 samples – after the preprocessing stage – which can be seen in Figure 3.3. The improved dataset has an objectively clearer separation of categories than the original had (see Figure 3.2).

Originally, the plan was to evaluate the chosen models only on CICIDS2017, however, after getting to know the problems of the dataset, I chose to write about these flaws and

[8]The GitHub repository of the corrected CICFlowMeter `https://github.com/GintsEngelen/CICFlowMeter` (Accessed: 2022-12-04)

[9]The repository of the LycoSTand tool `http://maupiti.univ-lemans.fr:2443/lycos/lycostand?` (Accessed: 2022-12-04)

[10]The dataset permutations made by Lanvin et al. `https://gitlab.inria.fr/mlanvin/crisis2022/-/tree/main/Labels` (Accessed: 2022-12-04)

their mitigation also. I will make a short chapter at the end of the thesis (Chapter 6) to show how this data could be preprocessed and classified by the same methods as used on the original CICIDS2017.

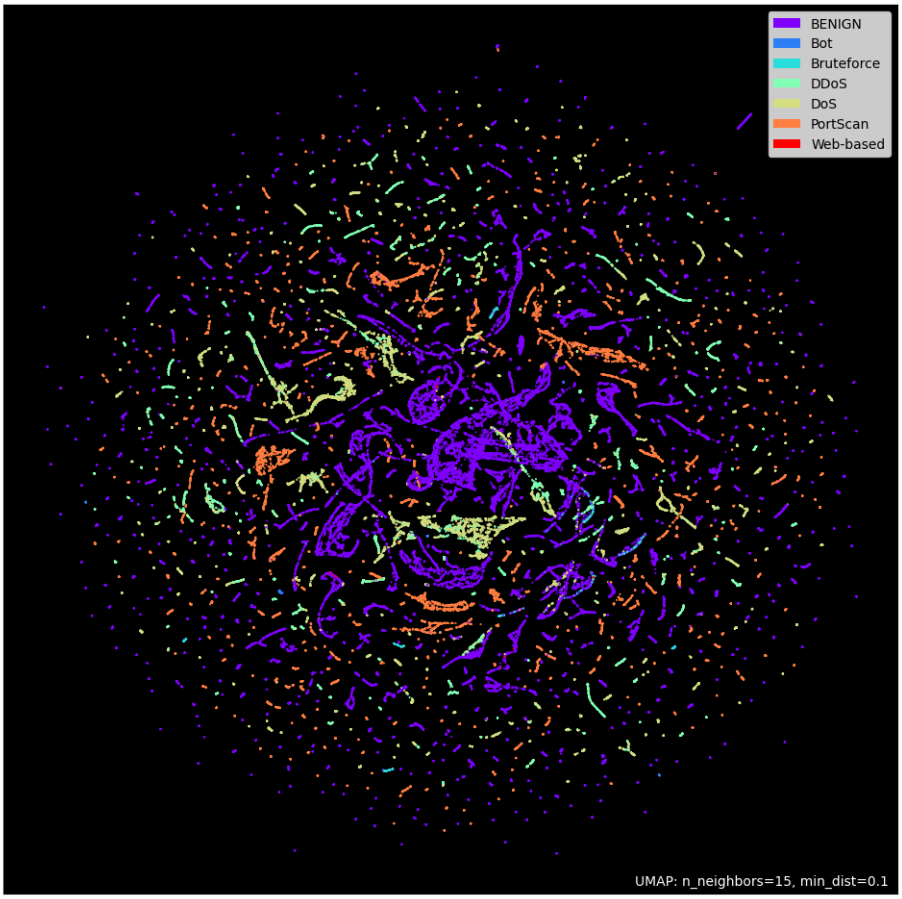


Figure 3.3: The 2 dimensional plot of the **RP** dataset

Chapter 4

Preprocessing

The CIC-IDS2017 dataset – and datasets in general – can be downloaded in a form that usually does not meet the purpose of training classifiers, furthermore, various scenarios might require different modifications on the data. This necessitates preprocessing, which prepares the dataset to be used for classification by changing the characteristics of it e.g. reducing the number of rows and/or columns. By the end of preprocessing the data should be as optimized as possible for the used ML models.

There are IDS related papers based on CICIDS2017 which focus on specific CSV files but I worked with all of the data provided by the authors of this dataset. The list of used files are shown in Table 4.1.

Name of file
Monday-WorkingHours.pcap_ISCX.csv
Tuesday-WorkingHours.pcap_ISCX.csv
Wednesday-workingHours.pcap_ISCX.csv
Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv
Thursday-WorkingHours-Afternoon-Infilteration.pcap_ISCX.csv
Friday-WorkingHours-Morning.pcap_ISCX.csv
Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv
Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv

Table 4.1: The files used for my work

In this case I am working with tree-based models, thus normalization or standardization is not needed because they are insensitive to this aspect. Furthermore, feature encoding is also unnecessary because the dataset does not contain categorical features. However, other generic alteration techniques are present in my preprocessing stage, which will be detailed in the sections below. The dataset contains 14 types of attacks (plus benign samples), which have high diversity in the number of occurrences as shown in Table 4.2.

It can be seen from the table that some attack types have notably less representatives than others. This is concerning when it comes to teaching models because models might need more balanced groups for classification. In order to improve the data in this regard – while also making the attacks as transparent and compact as possible – I came up with the idea of grouping attacks. This generalization will not ruin the outcome of the classifiers, in fact on the contrary, because this might help new attacks to be classified accurately. The resulting groups are more comprehensive and can lead to better understanding of the dataset. *Web Attack - Bruteforce*, *Web Attack - XSS*, and *Web Attack - Sql injection*

Category	Number of occurence
BENIGN	2273097
DoS Hulk	231073
PortScan	158930
DDoS	128027
DoS GoldenEye	10293
FTP-Patator	7938
SSH-Patator	5897
DoS slowloris	5796
DoS Slowhttptest	5499
Bot	1966
Web Attack - Brute Force	1507
Web Attack - XSS	652
Infiltration	36
Web Attack - Sql Injection	21
Heartbleed	11

Table 4.2: The categories of the samples

could be grouped into **Web-based** category because they are exploiting the same web application. **DoS** attacks in general can be executed in widely different manners – as I have detailed in Section 3.3.3 – but the goal with them is to cause outage, i.e. to disrupt a service. From this consideration, *DoS HULK*, *DoS GoldenEye*, *DoS slowloris*, and *DoS Slowhttptest* were grouped into **DoS** category. By the definition of these attacks, *Heartbleed* could also be included in this category, however, I chose to drop this attack type because it is not represented well enough. I dropped *Infiltration* as well for the same reason, moreover, it could not be grouped together with any other attacks. As I have detailed in Section 3.3.2, the attacks executed with Patator are for the brute forcing of a service. These attacks were grouped into **Bruteforce** category.

I dropped 4376 rows, which contained positive or negative infinite values because these values can cause errors during evaluation. To sum up, I had to drop 4423 rows, which is fortunate as it is only $0,156\%$ of the whole dataset. According to one of the easily missed best practices, I randomly shuffled the samples. The results are shown in Table 4.3.

Category	Number of occurence	Numeric notation
BENIGN	2271320	0
DoS	251712	1
PortScan	158804	2
DDoS	128025	3
Bruteforce	13832	4
Web-based	2180	5
Bot	1956	6

Table 4.3: The new groups

4.1 VarianceThreshold

Feature selection will be detailed in Section 4.4, however, I had to make a preliminary selection based on zero variance features. Each of these columns contain only a single value, which makes them unnecessary as they do not provide extra information for the later processes. By dropping them, the dataset will have less dimensions, which reduces run time for the preprocessing phase. I used `VarianceThreshold` from the scikit-learn package and with this approach I could remove all features, which are below a variance threshold. Variance can be calculated for each feature with the following equation:

$$\sigma^2 = \frac{1}{n} * \sum_{i=1}^{n} (x_i - \mu)^2, \tag{4.1}$$

where n is the number of samples and μ is the mean of the values in the column.

8 features were dropped, making further evaluations faster. The variance of each feature is shown in Figure 4.1.

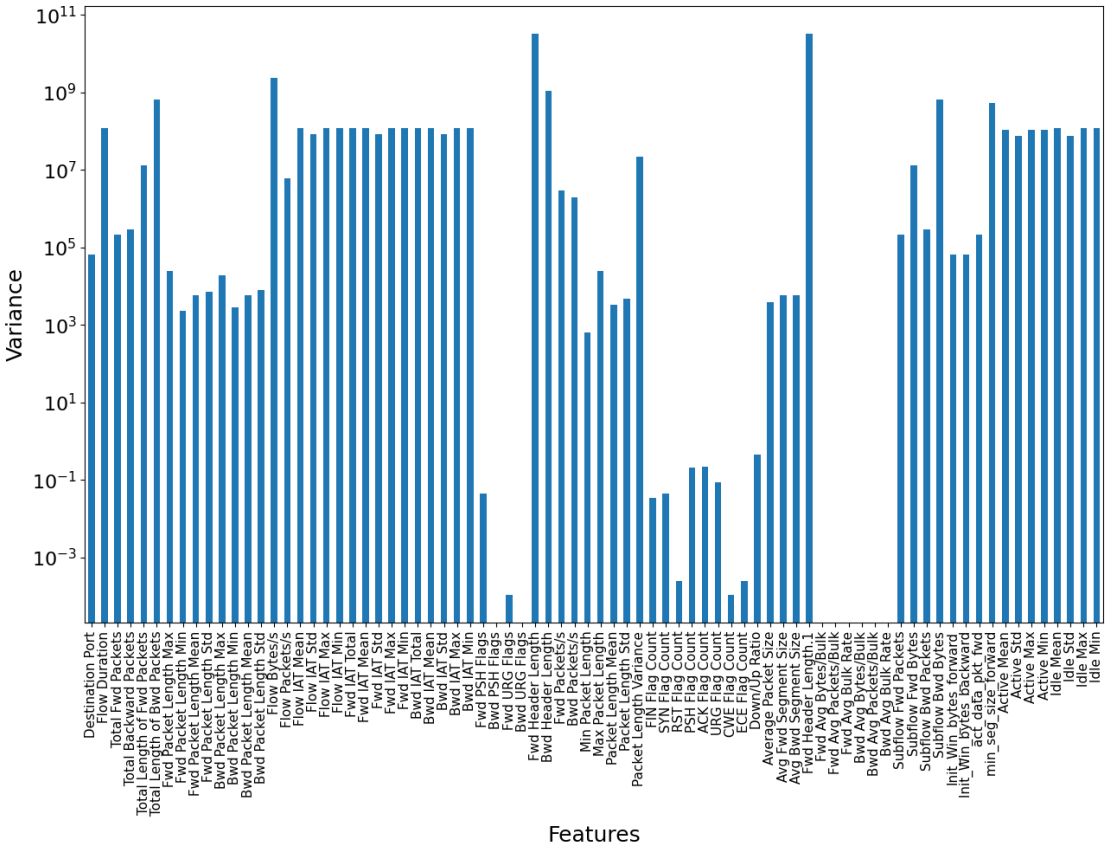


Figure 4.1: Features variance (on logarithmic scale)

4.2 Splitting

Under splitting I mean dividing the rows into three fully separable parts used for training, validation and test. Train and validation sets are assumed to be in our hands and test is the set which might have zero-days and in general attacks that are not necessarily

contained by the former two sets. In my thesis, I did not make steps in order to select attacks, those will only be contained by the test set because the attack pool is not big enough for this purpose. It models the previously unseen behaviours, which come up in real life scenarios. It is crucial to have a training set which is as close to perfect as possible because it is used to teach the models. Validation set is a part of the previously known data, however, it is used to get a feedback on how well the trained and fine-tuned model will classify new attacks, also meaning that it is used to choose the best hyperparameter setting (see Section 5.2).

I have tried several splitting measures i.e. $50 - 20 - 30$, $65 - 15 - 20$ and $70 - 15 - 15$, but I found that $60 - 20 - 20$ is the most viable ratio in this scenario, where the dataset has more than 2.8 million rows. I mainly chose this splitting ratio because it is broadly used in IDS related papers. With this split, train set has enough from each attack type to train the models, while it does not make the training phase extremely time consuming. Table 4.4 shows the number of samples for each category in the train set. Validation and test sets have both 565566 samples and around the same occurrences within categories because of the previously mentioned random shuffle.

Category	Number of occurence
BENIGN	1362936
DoS	151023
PortScan	95110
DDoS	76940
Bruteforce	8238
Web-based	1278
Bot	1172

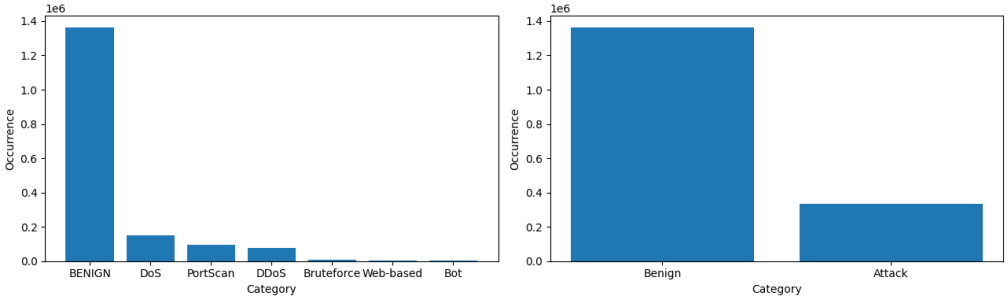
Table 4.4: The training set after splitting

4.3 Sampling

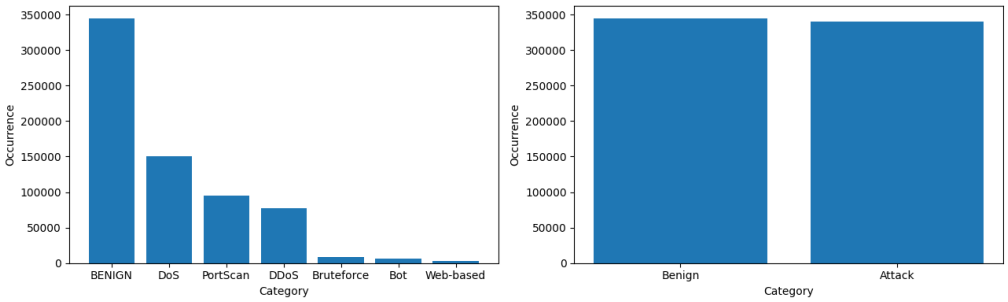
In this section I will detail the techniques I used for sampling the data. Sampling is an important step when the dataset is imbalanced. A dataset is imbalanced when the its labels are not roughly uniformly distributed. In Table 4.4, we can see that **BENIGN** samples are the majority by far, while **Web-based** and **Bot** categories have almost unrecognizably few occurrences. It is also notable that the attacks to benign ratio is around $1 : 4$, which is not ideal when we want to predict minority classes accurately. As a basic hypothesis, I wanted to sample down the majority class (the **BENIGN** samples) to the point of attacks. This proved not to be good enough, because the overall prediction error did not meet my expectations. Then I recognized that some kind of oversampling is also needed for the 2 least common attacks. However, making the CICIDS2017 dataset balanced is not as easy as it might seem because we do not want to decrease the expressiveness of the classes and we want to remove as many rows as possible at the same time. Oversampling on **Web-based** and **Bot** categories had to be tried several times with continuously changing the destined number of samples for the groups. These approaches were validated each time on the validation set to see whether the amount of oversampling used for these categories was beneficial.

The resulting class ratios after sampling can be seen in Figure 4.2. It can be seen on these ratios, that the dataset got balanced only by attacks to **BENIGN** ratio – which stands for binary classification – not by altering all categories to the same level. I tried

the latter method too, but it resulted in overfitting for several attack classes. The figure shows that **Bot** and **Web-based** attacks changed place in the number of occurrences as it is not required by oversampling to conserve the order in this sense. One important note: their labels will remain the same – which means 5 for **Web-based** and 6 for **Bot** category – because the sampling does not change the initial occurrence of the attacks. This method only tries to make the category more recognizable and learnable for the used model. Sampling was only used on the training set.



(a) before sampling



(b) after sampling

Figure 4.2: Occurrences before and after sampling the training data

4.3.1 Random Undersampling

For undersampling the **BENIGN** category, I used the `RandomUndersampling` [42] function from the imbalanced-learn package. The approach is quite simplistic: it takes random rows of the whole training dataset in a number previously specified but does not make any kind of exception on the samples. In other words, it is sampling with replacement. Thus, the method is time efficient in exchange for not taking in account the importance or uniqueness of rows. I used `RandomUndersampling` to randomly pick 345000 samples from **BENIGN** class in the training set, which is slightly higher than the total number of attack occurrences. This technique does not have any noticeable downsides for this class. This means that the amount of chosen **BENIGN** samples represent the whole class well enough. Another benefit of using this method is that we can reduce run time with it, which makes experimenting easier.

4.3.2 SMOTE

When a class is underrepresented in our training set, we can use various oversampling techniques to make duplicate or synthetic samples from our data. Synthetic Minority Oversampling Technique (SMOTE) [11] is a technique for the latter purpose. It aims to be a better alternative for random oversampling. Random oversampling takes random samples from the existing training set and adds it again to the dataset. This means that the chosen models will learn the oversampled rows multiple times, which might result in overfitting.

For this thesis, I chose `SMOTE` from the imbalanced-learn package and I oversampled **Web-based** attacks to 3000 and **Bot** attacks to 6000. SMOTE uses the k-Nearest Neighbors algorithm as the basis of its functionality. It has a `k_neighbors` parameter (defaults to 5), which tells the k-NN algorithm how many neighbors should be considered around each target sample (i.e. **Web-based** and **Bot** samples in this case). New points can be created between a target sample and its neighbors resulting in a maximum of $\frac{\texttt{k\_neighbors}*(\texttt{k\_neighbors-1})}{2}$ samples. The algorithm selects a random number $r \in (0, 1]$ for each synthetic sample. They will be created on the line between the target sample and one of its neighbors with the following equation:

$$S_i = T + r * \overrightarrow{TN_i}, \tag{4.2}$$

where $i = 0, \ldots, \texttt{k\_neighbors} - 1$, S_i is the synthetic sample, T is the target sample and $\overrightarrow{TN_i}$ is the vector between the target sample and its ith neighbor.

4.4 Feature elimination

Feature elimination can be applied to datasets, where the amount of features is found to be overwhelming. In order to eliminate features, first the selection of important features should be done. This method can be used in almost every training scenario because sometimes there are columns which are not important enough for the final estimator.

In my case, I had 70 features (plus label) after removing the zero variance features. Using this set for training the model still results in too high computational times on my hardware (see Chapter 5). To reduce this, we can eliminate features which might also be useful to remove less expressive features. In order to select features I applied a widely used method: the ANOVA F-test.

Analysis of variance (ANOVA) [64] is a collection of statistical hypothesis tests used for analyzing the differences among means. One of these tests is the F-test, which calculates the ratio of the variance between and within features.

In order to use the output of the F-test, I used `RandomForestClassifier` (see Section 5.4) to evaluate the difference between the selection of incrementally smaller sets from the best features. ANOVA F-test tests each feature separately and it checks if the distribution of the feature is the same for the different target categories. For the feature selection – by the values ANOVA F-test assigned to columns – I used `SelectKBest` from the scikit-learn package, which has a parameter `k` to set the number of top features to select. Then I masked the train and validation subsets with the output. In each iteration, I fitted the classifier on the train set with a smaller feature set every time. Then I evaluated the model on the also modified validation subset with F1 macro (see Section 5.1.6 and 5.1.2.1) as the metric. The output is shown in Figure 4.3.

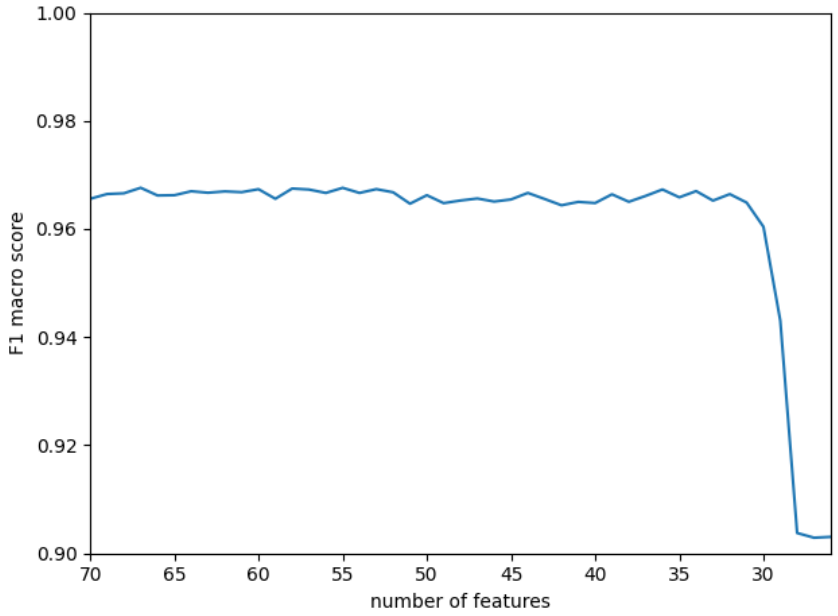


Figure 4.3: F-test iterations

As it can be seen in the figure, there are many features that do not make predictions better on the validation set. The selected number of features should be chosen by taking the run time reduction-data expressiveness trade-off into consideration, which might change in different scenarios. In this case, the rule of thumb was to choose the threshold before the formation of a knee. The last decent score is 0.9664 with 32 features. I wanted to sum up these features with their descriptions, however, the descriptions provided by the authors of the dataset are grammatically incorrect in many cases meaning that they are not always easy to interpret[1]. I chose to omit this table because I could not get a deep inspection of their CICFlowMeter tool during my thesis, which could lead to the correct descriptions.

4.5 Preprocessing summary

As I have mentioned previously, one of the key motivations behind my preprocessing approach is to reduce training time of the used models. To see how a preprocessing stage affects the training set, I fitted a model with the evolved training set each time and validated on the validation set. The model I used was `RandomForestClassifier` which will be detailed in Section 5.4. I chose this model because it is an ensemble learning method, which has the lowest time complexity of the ensemble learning models used in this thesis. For the validation, I used the following metrics: F1 score with macro averaging, recall with macro averaging, and precision with macro averaging (see Section 5.1). The results of each preprocessing step can be seen in Figure 4.4.

The figure contains 3 thresholds: *Default*, *Sampling*, and *Feature elimination*. The *Default* contains scores from the evaluation made after the elimination of zero variance features. It can be seen in the figure that in each stage the scores stay almost the same, which is the sign of success because we could reduce run time significantly (see Figure 4.5) while

[1]The feature names with descriptions `https://github.com/CanadianInstituteForCybersecurity/CICFlowMeter/blob/acaf8bea8611fb4b996b4d33964dfd9155d9efdf/ReadMe.txt#L79` (Accessed: 2022-12-04)

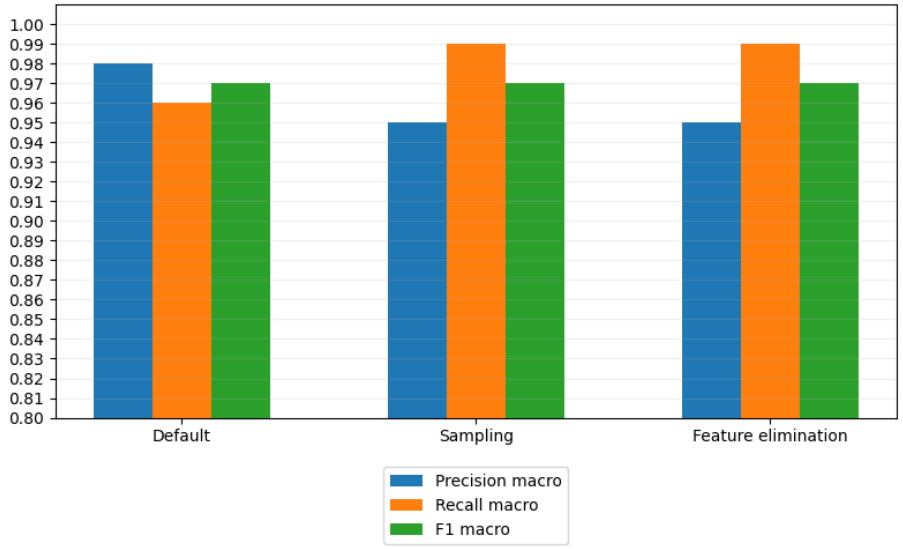


Figure 4.4: Results of preprocessing stages

keeping the predictive quality high. Precision-recall trade-off can also be seen in the figure, which means that if we boost precision up recall gets lower.

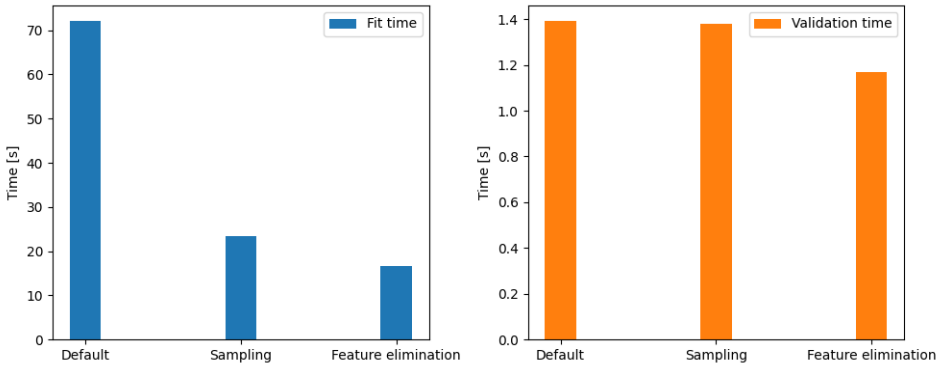


Figure 4.5: Times passed during fitting and validation

In Figure 4.5, we can see that the time of fitting the model is radically reduced with the sampling step and after the feature elimination stage the fit time reduced by 55.3 seconds. The second part of the figure shows that only *Feature elimination* changed the time complexity of validation, which is because validation (and test) sets were only altered in that process.

Chapter 5

Evaluation

After the preprocessing of the dataset was done I could test how different models perform on it. As it was previously mentioned, the main goal of preprocessing is to reduce the run time of the models which is crucial when working in a home environment where resources are limited. The specifications of my computer can be seen in Table 5.1. In some cases, when working with larger datasets the importance of page files gets bigger. I used a maximum of 12 GB on my SSD and 24 GB on my HDD for this purpose.

Hardware	Manufacturer	Type	Specifications
CPU	AMD	Ryzen 5 5600X	6 core, 12 thread, 3.7 GHz
RAM	Kingston	HyperX Fury	32 GB (4*8 GB), 3200 MHz
GPU	ASUS	GeForce RTX 2060 OC	6 GB GDDR6

Table 5.1: The used computer resources

I set up 2 different environments as I have mentioned in Chapter 2. One of them is directly on a Windows 10 host and the other one works under WSL 2. The latter is practical because I used GPU acceleration for one of the models (see Section 5.6) and WSL is capable of accessing it. It is important to note that my GPU is not optimal for this kind of evaluation as it is made for a more general use-case. I wanted to try GPU evaluation anyway because of its expected faster performance. With that said, time complexity is not the first priority, when looking for the most optimal models as my setup is not built for this usage and there is no target architecture to optimize the models for.

Arguably the most important phase of the intrusion detection topic is evaluation. In the sections below I would like to present a clear result which should be able to make my work comparable to other methods proposed in other papers. Because of the complexity of the evaluation process, in the followings I will present the base knowledge behind evaluating models. Section 5.1 contains the metrics that can be used to compare the results of the learning and classifying performance of the models. These metrics will be presented through mathematical explanations and also with figures when it is sufficient. There is no model that can be used for any data out-of-the-box and will provide the perfect solution. Moreover – to this day – it would not be a realistic intention. Therefore, the models should be customizable with parameters constraining their training methods. Some of these parameters, known as hyperparameters, cannot be perfectly found by only considering the model and dataset used. To find their presumably best value, hyperparameter fine-tuning is the key which will be detailed in Section 5.2.

After that, each model will be detailed with a high-level approach, sometimes writing about their mathematical background when it is crucial. These sections will try to focus on the features of the models, which are useful for beginner researchers and for introducing the concepts behind them. Later, for each model there will be a section where I write about their fine-tuning and evaluation procedures. Fine-tuning will be constrained to just $2 - 3$ parameters each time – because of the high computational time on my setup – for finding suitable parameters. There will be repeated figures throughout the description of models. The meaning of these figures will only be described in Section 5.3.

The section for each model will continue with a short evaluation part, where I summarize the classification with results broken down by categories. I used Decision Tree (Section 5.3), Random Forest (Section 5.4), AdaBoost (Section 5.5), and XGBoost (Section 5.6), which are famous machine learning models. The latter three are ensemble learning methods. Usually these methods share the base idea of using some variations of Decision Trees as their base estimator and this is why I have included it too.

After the separate descriptions of the evaluation of the models, the evaluation summary (Section 5.7) will contain the comparison of the models. This includes, e.g. the analysis of their time complexity while training, validation, and testing, their overall classification success measured by some of the metrics detailed in Section 5.1, and the analysis of the false negative samples (see Section 5.1) which could not be classified by the majority of these models. I will also compare some of the results with other papers. It is hard to find perfectly comparable papers – but naturally it is not necessary for them to be perfectly comparable – as the preprocessing phase differs in many cases. The most notable difference is made by the dropped attack types, which are (**Hearthbleed** and **Infiltration**). Although, the lack of these attacks are not causing highly unfair comparisons, I will describe the main differences between the to be compared approaches in each case.

5.1 Metrics

There are two main methods when evaluating IDS datasets. One of them is binary classification, which is when there are just two possible labels in the dataset meaning, e.g. one category for attacks and one category for benign behaviour. The other one is multiclass classification, when the attacks are distinguished too. For this thesis I used multiclass classification as I not only want to separate attacks as deviation from normal behaviour but to categorize them as a more real life approach. The following metrics are used to measure the performance of the anomaly detection methods in my thesis. They also make the proposed methodologies and algorithms comparable with other methods described in other papers.

5.1.1 Confusion matrix

The confusion matrix is a specific table for supervised learning, which reveals the performance of classification algorithms [47]. The two axes of the matrix represent the predicted and the actual values. Although I used multiclass classification, the binary confusion matrix will be presented first as it is easier to interpret.

In the binary approach the previously mentioned two axes contain only true and false values. This makes up 4 possible outcomes, which are true positive (TP), false positive (FP), true negative (TN), and false negative (FN). An ideal confusion matrix would contain a high number of samples in true positive and true negative states and around zero in false positive and false negative states. Figure 5.1 shows an example of a binary confusion matrix.

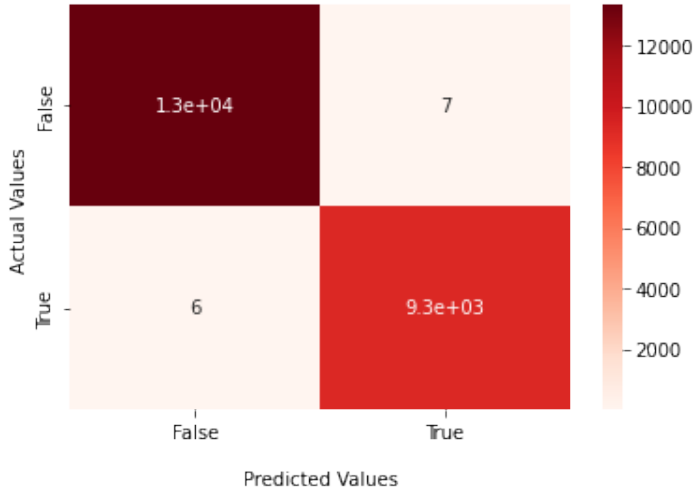


Figure 5.1: Example of an ideal binary confusion matrix

The axes of the multiclass version have as many values as the number of categories in the dataset. Thus, in the preprocessed CICIDS2017 dataset there will be 7 values on each axis representing: **BENIGN**, **DoS**, **PortScan**, **DDoS**, **Bruteforce**, **Web-based**, and **Bot**. To have a simpler layout, I used numbers instead of words for the figures. The numeric replacements can be seen in Table 4.3 and an example of a multiclass confusion matrix is shown in Figure 5.2.

Let CM be an $n \times n$ matrix, i be the row index, j the column index, and $CM_{i,j}$ an element of the matrix, where $i, j \in [0, n-1]$. Attacks are in those cells, where $i, j > 0$. The values,

where $i = j$ and $i, j > 0$ are TP, leaving other cells in this area to be miscategorized attacks. TN samples are in the cell, where $i, j = 0$. FP samples are represented in the cells, where $i = 0$ and $j > 0$. $CM_{i,j}$ is FN, when $i > 0$ and $j = 0$. To summarize: if most of the values stay in the diagonal, then it is generally an indicator of good classification. Further improvements can be issued with bringing FN classifications as close to 0 as possible if needed.

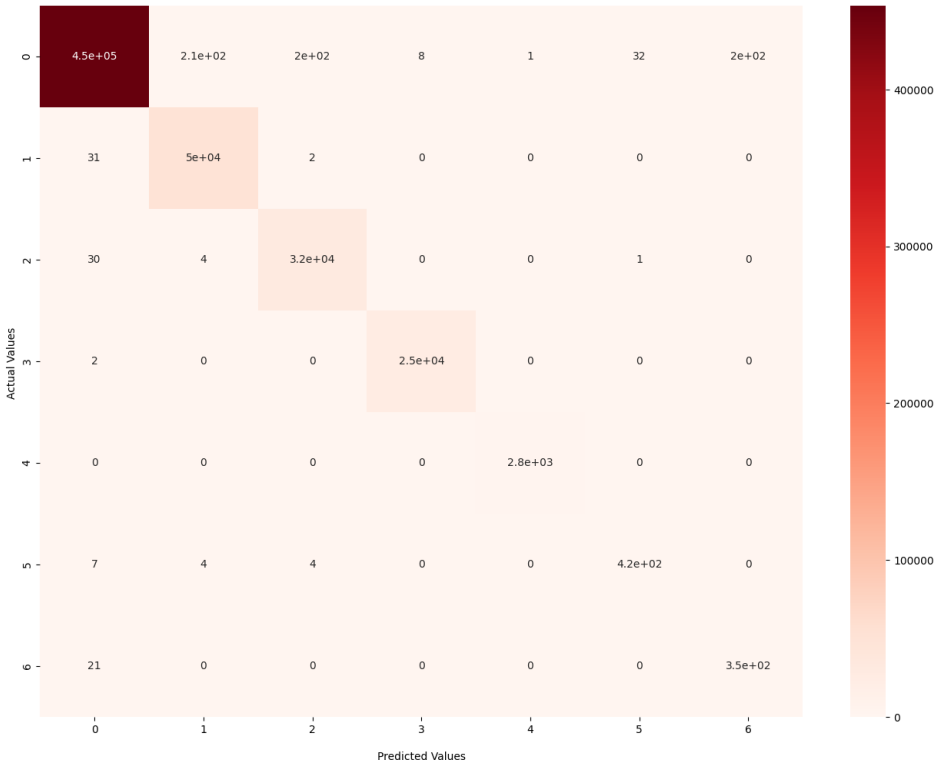


Figure 5.2: Example of a multiclass confusion matrix, which represents an almost perfect outcome

For metrics to be calculated in a multiclass scenario this confusion matrix is not suitable. Consequently, we need to break it down into multiple binary matrices. This can be done with making a matrix for every category where the label of the actual target class will be replaced with 1 and the labels of other classes with 0.

5.1.2 Averaging techniques for multiclass classification

When measuring the performance of a multiclass classification, the goal is to reduce the problem to binary classifications. These binary cases will be calculated with the alteration of the actual and predicted labels as it was formerly mentioned (see Section 5.1.1). This way we can calculate the metrics mentioned in the followings. If the case requires only one output value, then several averaging techniques can be performed in order to do so. Micro averaging will not be detailed because it is basically calculating accuracy [19].

5.1.2.1 Macro averaging

The macro average – or otherwise known as unweighted mean – can be calculated by taking the mean of the result of each class by the target metric [19, 54].

$$metric_{macro} = \frac{\sum_{i=0}^{n-1} metric_i}{n}, \tag{5.1}$$

where n is the number of classes and $metric_i$ is the result of the target metric for class i.

5.1.2.2 Weighted averaging

Weighted average takes into account the supporting samples behind the classifications for each category by multiplying each score with the presence of the categories in the used dataset [54].

$$metric_{weighted} = \sum_{i=0}^{n-1} metric_i * \frac{support_i}{m}, \tag{5.2}$$

where n is the number of classes, m is the number of samples in the whole data used and $support_i$ is the number of samples in class i.

I will use only macro averaging in my work because weighted averaging is too lenient towards the misclassification of low represented classes in imbalanced datasets.

5.1.3 Accuracy

Accuracy measures the ratio of correctly classified samples. If the predicted labels (y') and actual labels (y) are given, the following equation can be written for binary classification [54]:

$$accuracy(y, y') = \frac{1}{n} * \sum_{i=0}^{n-1} I(y'_i = y_i), \tag{5.3}$$

where y_i is the ith true value and y'_i is the ith predicted value and $I(x)$ is the indicator function. Parameter n is the length of the y column. The simplified expression of this – which can also be used for multiclass classification – is [19]:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \tag{5.4}$$

5.1.4 Average precision

To introduce the average precision, there are two metrics that have to be defined first. One of them is precision, which is the ratio of the true positive values and the predicted positive samples [19],

$$precision = \frac{TP}{TP + FP}. \tag{5.5}$$

The other one is recall, which is the ratio of the true positive samples and all positive samples [19].

$$recall = \frac{TP}{TP + FN}. \tag{5.6}$$

The average precision (AP) score is calculated by summarizing a precision-recall curve as the weighted mean of precisions at each threshold. To calculate this score – to one target class – the following equation can be written [46]:

$$AP = \sum_n (R_n - R_{n-1}) * P_n, \tag{5.7}$$

where R_n is the recall and P_n is the precision at the nth threshold. Figure 5.3 shows an example for an AP curve.

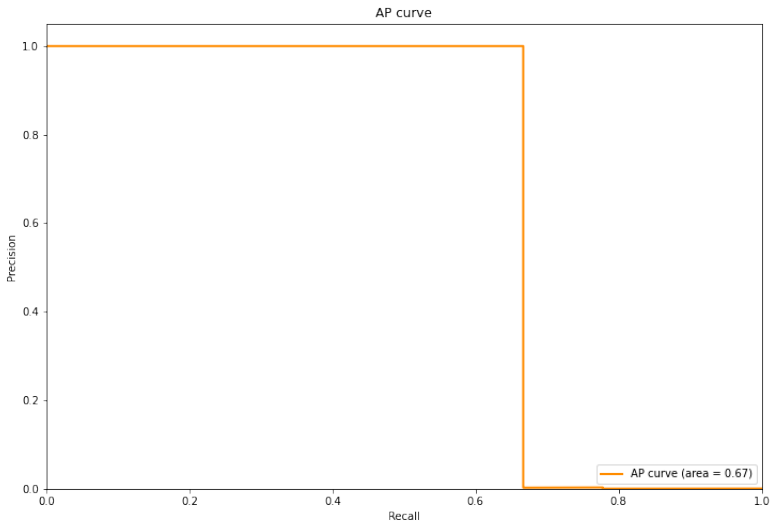


Figure 5.3: An AP curve with the score of 0.67

5.1.5 ROC AUC

Receiver operating characteristic (ROC) is a probability curve. It is plotted with false positive rate (FPR) on the x-axis against the recall – otherwise known as true positive rate (TPR) – on the y-axis, where [66]:

$$FPR = \frac{FP}{FP + TN}. \tag{5.8}$$

The area under the ROC curve (ROC AUC) [66] gives a scalar value in the $[0.5, 1.0]$ range. If the ROC AUC score is 1.0, then the model is considered a perfect classifier. On the other hand, if the score is 0.5, then it means that the model chooses labels randomly for the sample, i.e. it works like a random classifier. Figure 5.4 shows an example for a ROC curve.

I used the one-vs-rest (OVR) version for this metric with macro averaging. The one-vs-rest method means that the AUC of each class is computed against the rest [54].

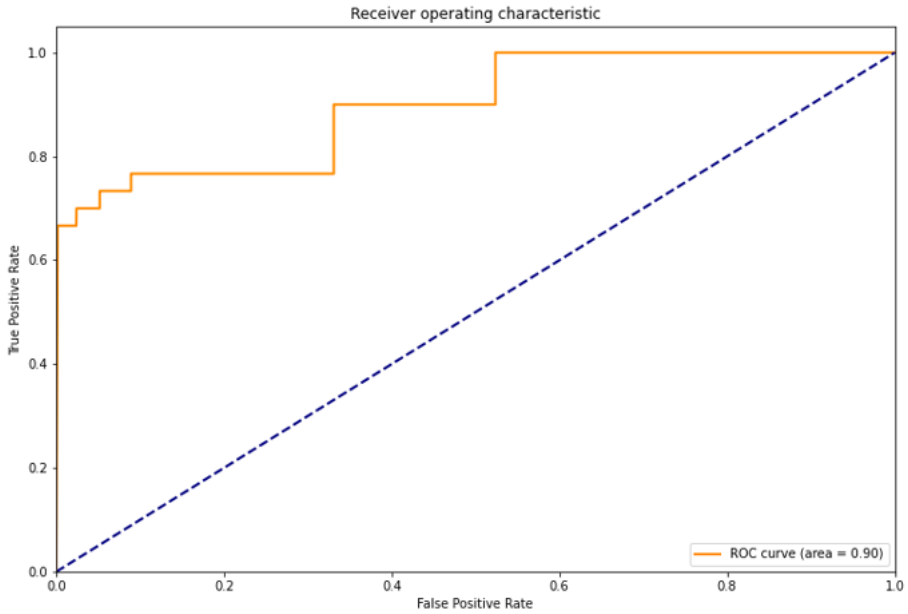


Figure 5.4: A plotted ROC curve with ROC AUC being 0.9

5.1.6 F1 score

ROC AUC score is not as sensitive to class imbalance as it might be needed for some scenarios. For those cases F1 score should be used. With this metric low false predictions will result in a high score as the relative contribution of precision and recall are equal [51]. F1 score can be calculated for each class with the following equation:

$$F1 = 2 * \frac{precision * recall}{precision + recall}, \tag{5.9}$$

where precision and recall are the metrics that are formerly discussed in Section 5.1.4.

5.2 Hyperparameter tuning

ML models have different parameters that can be set. This customization is required for the models to be able to provide appropriate results for various datasets. Some of these parameters can be easily chosen to fit the actual scenario but there are also some which can almost only be guessed. These are called hyperparameters.

In order to maximize the performance of the chosen model, these hyperparameters have to be tuned. After the iterations on different values, we can pick the one with the best performance. The performance might be measured by different means depending on the actual use-case. In machine learning this process plays a big role, thus I will detail it further in Section 5.2.1. However, there are different circumstances which can make this maximization task hard or even impossible with my current resources (see Chapter 5).

One of them is selecting the value set of the hyperparameters correctly. E.g. preliminary data inspection and evaluation with basic parameters can be done to get a basic idea of them. However, in some cases these are not sufficient and also some manual evaluation is needed where the parameters are changed by hand every time in order to find possible boundaries for their values. Also, this set should not be too big nor small because bigger sets result in high computational time and smaller ones in low coverage. Another problem with hyperparameters is that even if the perfect values are selected, how to know that it will classify also new data correctly, i.e. will it be robust enough for the use-case. The only assumption we can make on this is that the training set is reliable and large enough for preparing the classifier to be evaluated on the test set.

As the rule of thumb, I chose to tune all of the methods with randomly selected 100000 samples from the train set. I will refer to these samples as fine-tuning set. This sampling measure had to be done because the size of the training set was too large for my computational resources [4]. Also, it ensures that each classifier gets tuned on the same subset; the used machine learning methods have some kind of sampling built-in, however, I could not control them the same way. The results of this process might be biased because training data used for tuning was smaller, so the model training might not reach the optimum. However, this alteration is needed because of the limited resources the home environment has. Also, optimizing every method to reach its maximum effectiveness of classifying samples is not in the scope of this thesis. The validation set has a big emphasis in the tuning process because the evaluation of the best hyperparameter sets will be done on it. If the best estimator of a fine-tuning procedure is achieving poor scores on this set, then the value ranges provided for the fine-tuning procedure are not accurate and need altering. The process of fine-tuning can be seen in Figure 5.5.

5.2.1 Grid search with cross-validation

Before writing about the grid search mechanism, I will detail one of the machine learning fundamentals, which is cross-validation (CV). It is a widely used strategy to test and compare the performance of ML models and has several variants which differ in execution.

K-fold CV is one of them and it can be found in the scikit-learn package under the name `KFold`. This cross-validation technique randomizes the data and then splits it into K pieces [6]. Then the algorithm fits the model each time with different $K-1$ piece of the whole dataset and evaluates on the last 1 split through K iterations. The $K-1$ piece is referred to as train, leaving 1 for testing purposes. Important to note that in the fine-tuning part of each classifier I will use these train and test splits – or train and test folds

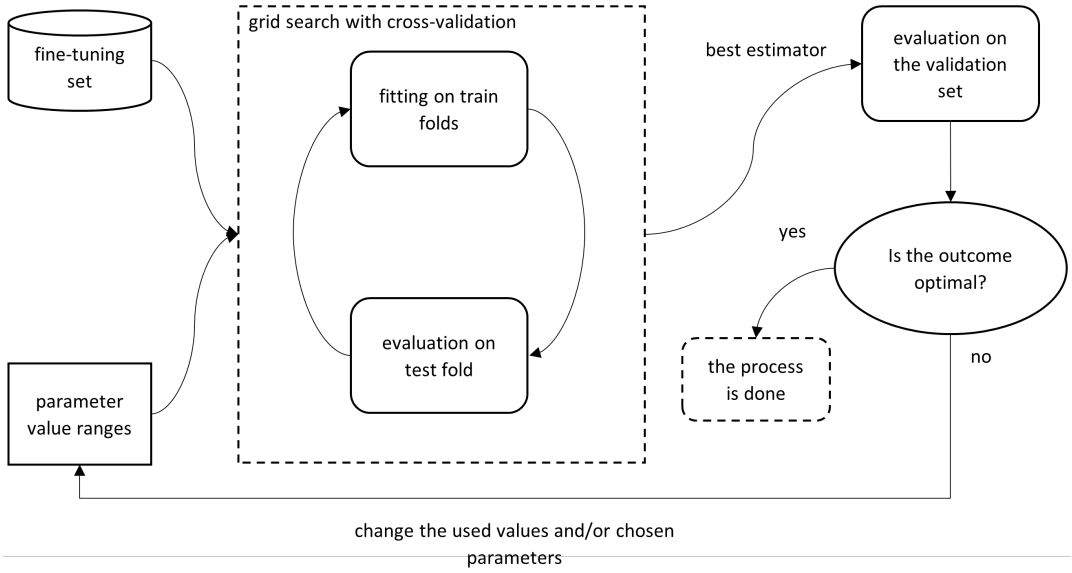


Figure 5.5: The process of fine-tuning

– not the global ones. One common parameter for K is 5. The splitting of a dataset is shown in Figure 5.6.

Figure 5.6: Example for the data splitting of 5-fold cross-validation

The K-fold CV then takes the average of the scores of each iteration: $CV = \frac{\sum_{i=0}^{K-1} CV_i}{K}$, where CV is the final score and CV_i is the ith. There are several metrics to choose from for cross-validation(see Section 5.1). I used F1 macro, ROC AUC OVR and precision macro for the tuning part.

There is an improved version of the K-fold CV technique, which is called Stratified K-fold. This method can be found as `StratifiedKFold` [56] in scikit-learn. The modification takes place in the splitting part of the K-fold, where this method considers the imbalance of the dataset, i.e. it preserves the ratio of resemblance of the dataset within folds [6]. I used mainly this method in my thesis because the dataset is balanced only in the benign-attack aspect, but it is not by categories along which the evaluation will take place.

Grid search is a popular method for fine-tuning with a rather simplistic approach. It takes a value set for each hyperparameter and exhaustively searches in order to find the best permutation. The best permutation of values will be determined by cross-validating each resulting model by the target metric(s) (see Section 5.1). The grid search is a fully parallelized process as each model will be fitted independently of each other. For grid search, I used `GridSearchCV` [53] from the scikit-learn package. It requires an estimator which needs to be tuned. It can be set with the `estimator` parameter. Also a `param_grid` have to be provided for the method in which parameter sets are given. We can specify metrics to use for validation with setting the `scoring` parameter. The used metrics were F1 score with macro averaging, precision with macro averaging, and ROC AUC OVR. However, when using multiple metrics the `refit` parameter should be set to one of the scorers by which the grid search can chose the best parameter value set. The grid search utilizes ranking in order to find the best value set. If the ranks for more configurations are the same then the first (i.e. the value set with the lowest index) will be chosen. I chose the F1 macro score for `refit` as I found it the most fitting metric to judge by. The ROC AUC OVR tended to be around 1.0 most of the time and the precision macro score proved to be too strict, which is not suitable in this scenario. Its strictness comes from Equation (5.5). It assigns high scores for classifications which produce low amount of FP samples. Having low amount of FP samples is good, however, it does not mean that the FN samples will also be low. Reducing the occurrence of FN classifications has a higher priority in my thesis. The cross-validation strategy can be set with the `cv` parameter. It defaults to 5-fold CV but I used the previously mentioned `StratifiedKFold` with `n_splits` (same as K) equal to 5. The method can also assure an evaluation on the training folds which can be used to see whether the parameter used resulted in underfitting, overfitting, or it made a good fit. This can be done with `return_train_score=True`. As I have mentioned earlier, the method can safely run parallelly, which can be controlled with setting the `n_jobs` parameter. To utilize all CPU threads -1 should be used for this parameter.

After the tuning process of each estimators, they will be trained on the whole training set instead of the fine-tuning set. The goal with the fine-tuning set was only to reduce the time complexity of hyperparameter fine-tuning.

5.3 Decision Tree

Decision trees are tree-like models, which are capable of forming expressive decision-making knowledge. They consist of two kinds of entities: decision nodes and leaf nodes. The former usually contains a parameter along which the decision has to be made and has multiple branches depending on the amount of possible outcomes. The latter is the bottom of a branch and represents the output of the decision set made on the branch. In other words, branches are the rules for classification. One leaf node cannot be contained by multiple branches. This provides the divisibility of the data. To build a Decision Tree, there are multiple algorithms such as C4.5, CART, and ID3. I used `DecisionTreeClassifier` from the scikit-learn package [49], which implements an optimized version of CART [48], thus I will detail this specific algorithm.

The Classification And Regression Trees (CART) [67] model is a binary tree. Each decision node represents a statement; if it is true for the data, it will continue on the true branch, otherwise on the false branch. At the bottom of the tree there are leaves, which contain the classification types based on the outcome of prior statements. To illustrate its structure, Figure 5.7 represents a CART model. This implementation does not support categorical features, however, the original version does. To get over this, we have to use feature encoding, which will not be detailed in this thesis as the used dataset does not contain features of this kind.

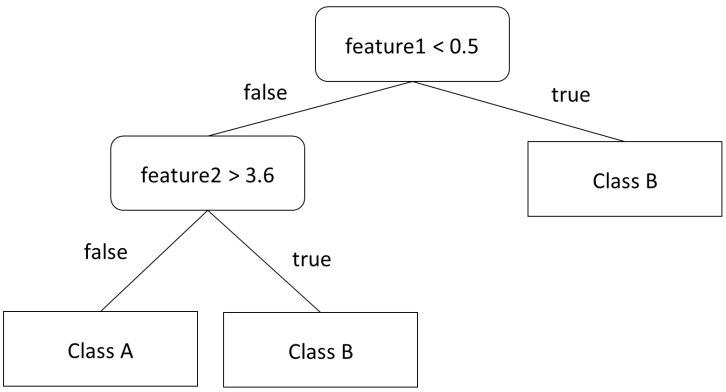


Figure 5.7: The structure of a Decision Tree

One fundamental parameter of the model is `criterion`. The model will use this parameter to find the best splits on a decision node. It can have 3 states: information gain, gini impurity, and log loss. In my thesis, I sticked to using information gain because their difference with gini impurity is negligible and log loss returns probabilities, which is not suitable for my evaluation concepts. Information gain is based on entropy. Entropy measures the average information in an n bit long sequence, when the bits have the given distribution. If the probability that bit i takes the value v_i for a statement is $P(v_i)$ then the entropy [30], denoted as $H(X)$, can be calculated with

$$H(X) = -\sum_{i=0}^{n-1} P(v_i) * \log_2 P(v_i), \tag{5.10}$$

The information gain $G(D|A)$ can be calculated with the following equation [30]:

$$G(D|A) = H(D) - H(D|A), \tag{5.11}$$

where D is the dataset, A is a feature of the dataset and $H(Y|X)$ is the conditional entropy, which is the amount of information, that describes the outcome of random variable Y, given that the value of X is known. Conditional entropy can be calculated with the expression below [30]:

$$H(Y|X) = \sum_{x \in X} P(x) * H(Y|X = x). \tag{5.12}$$

The algorithm constructs statements on the decision nodes with a feature and a threshold which is – in this case – chosen by the largest information gain. This makes my case different from the original CART model because it uses gini impurity as `criterion`.

The model also has several hyperparameters. I will discuss 3 of them. Firstly, there is `max_depth`, which constrains the maximum length of a branch with an exact number. This parameter is useful when we want more generic rules extracted from the model. It can also prevent overfitting but it is important to note that choosing a small `max_depth` results in underfitting. An essential parameter is `min_samples_leaf`, which defines the minimum number of samples needed to form a leaf. The impact of this is that other hyperparameters are considered only if they produce leaf nodes passing this norm. It helps making more generic rules because otherwise the leaves can contain very few samples. The next parameter is `min_samples_split`, which defines how many samples are needed to split a decision node. In this case, splitting means that the samples on the decision node can be divided into two further branches. Higher numbers prevent the model from learning detailed relations. Figure 5.8 shows an example of these parameters in practice.

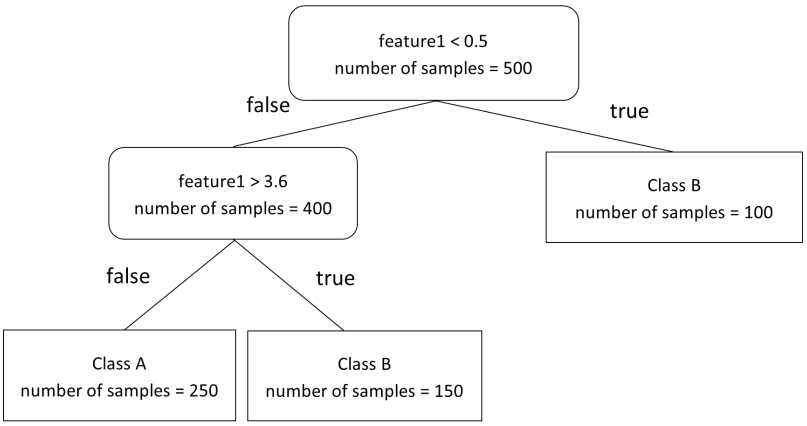


Figure 5.8: A Decision Tree parametrized with `max_depth=2`, `min_samples_leaf=100` and `min_samples_split=300`

The last parameter I would like to introduce is `class_weight`. It is immensely helpful if the categories are imbalanced. As mentioned earlier in Section 4.3, imbalance is present in the dataset. With this parameter, the significance of each class can be set. I used `class_weight='balanced'`, which assigns higher importance to lower represented categories and vice versa.

5.3.1 Fine-tuning

The tuning of a Decision Tree is the least time consuming from the ML methods that I will detail in the following sections, thus I tried to search for as many parameter combinations as possible. I chose to tune the above mentioned 3 hyperparameters because of their characteristics: `max_depth`, `min_samples_leaf`, and `min_samples_split`. The value ranges corresponding to them are in Table 5.2. The following parameters were set for each iteration: `criterion='entropy'`, `class_weight='balanced'`, and `random_state=42`. The F1 macro score of the model with the tuned parameters is 0.95 on the validation set after fitting on the whole training set.

Parameter	Value range
min_samples_leaf	$1, 2, 4, 6, 8, 10, 12, 14, 16$
min_samples_split	$2, 4, 8, 12, 16, 20, 24, 28, 32$
max_depth	$10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, \text{None}$

Table 5.2: The used parameter ranges

As the `min_samples_leaf` and `min_samples_split` parameters are highly correlated, I chose their range correspondingly. I started by adding their default values which are `min_samples_leaf=1`, `min_samples_split=2` and `max_depth=None`.

By looking at the value ranges it is recognizable that grid search will make some unnecessary steps. Let P be a parameter set made by the permutation of the value ranges. Let us assume that $P = \{\texttt{min\_samples\_leaf=10}, \texttt{min\_samples\_split=4}, \texttt{max\_depth=16}\}$. These parameter values will make contradiction in the model. The splitting parameter tells the model, that it can split a decision node until it reaches the minimum of 4, however, the minimum number of samples in leaf nodes is constrained to 10. This is not possible at the same time, thus the `min_samples_split` will be automatically overrided because the minimum number of leaf nodes is more constraining. I accepted this trade-off because other than this, the grid search is the simplest way of choosing the best parameter set and also, thanks to the low computational cost, this approach was feasible. The `max_depth` value range was chosen by intuition, as I assumed that the Decision Tree will be underfitted below 10. It turned out that this range will be below 16 as my results show it in Figure 5.9. The `max_depth` can be also set to *None* – which is also the default value for it – and it tells the model that there is no limit for this parameter.

After running the grid search, the chosen set of parameters is `min_samples_leaf=1`, `min_samples_split=4`, `max_depth=None`. The `max_depth` of this Decision Tree is 48 in my environment, which is generally considered deep but given that it is a single tree classifier, it does not boost time complexity to extreme levels. The total run time of the fine-tuning was ≈ 10 minutes which is a rather high computation time based on the previously mentioned hardware (see Chapter 5). This measurements had to be done multiple times until I have found the perfect pool of parameters which gives another time complexity layer to this process.

Figure 5.9 shows the mean of the evaluation metrics grouped by each parameter i.e. the means of the scores by the actual metric for each permutation are taken. These metrics are represented with their test and train values too. The 'x' symbols mark the maximum scores by each metric. The symbols connected to the mean F1 scores are especially important because the grid search is refitted with F1 macro (see Section 5.2). These marks do not necessarily represent the parameters which will be used by getting the best estimator from grid search. It uses a different approach (ranking) for choosing the best estimator. The

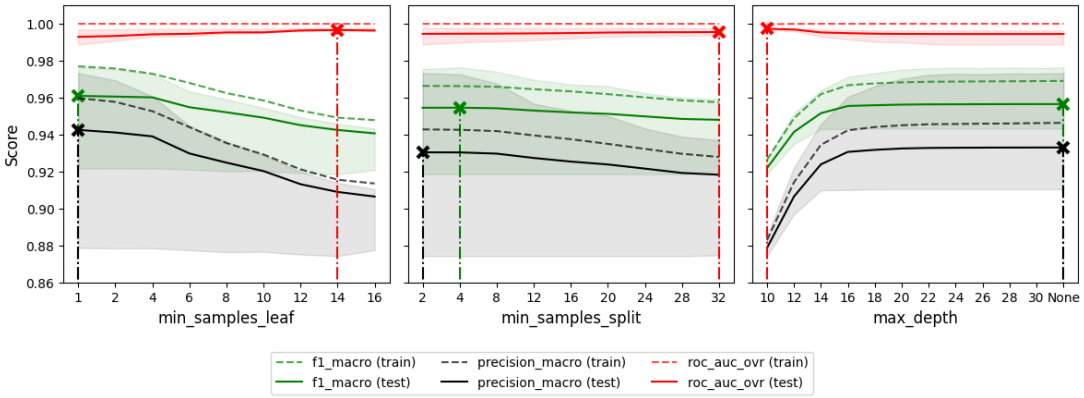


Figure 5.9: The result of grid search broken down to the chosen parameters

areas with fainter colour indicate the range of obtained mean test scores minimum to maximum.

It can be seen by the unfolding of the mean train and test scores, that the lower values for `min_samples_leaf` result in slight overfitting. However, as it is not a major difference when talking about the deviation between test and train scores, I accepted the output of grid search as optimal. Choosing the minimum leaf size higher would lead to a more generic model, however the results show that such endeavor would decrease the mean test F1 macro score by 1% and the mean test precision macro even more ($\approx 1.5\%$). As it can be seen, changing `min_samples_split` does not make that big of an impact on the outcome because it is almost plateauing in terms of the mean test F1 macro score. On the other hand, the choice of `max_depth` has high importance. Choosing it below 16 makes the model underfitted, however, the values are almost constant after it for all three of the scores. It can also be seen that a higher `min_samples_leaf` and `min_samples_split` value would make the mean test scores less uncertain. Although, having higher `max_depth` decreases the certainty. Mean test precision macro score tends to be more sensitive to these changes, which means that the number of FP samples were very different thorough validation splits.

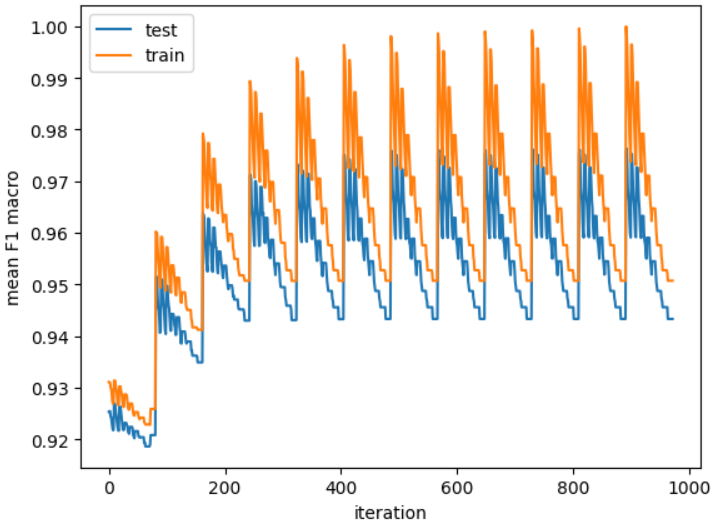


Figure 5.10: The mean test F1 macro scores of every iteration

The previous observation is also demonstrated by Figure 5.10, where the mean test F1 macro scores are plotted in each iteration. An iteration means an evaluation of a permutation of values from the hyperparameter set. Each spike corresponds to a Decision Tree with a new maximum depth from 10 to 'None'. This figure shows a gradual increase in scores. Also, the deviation between train and test scores gets bigger. The test scores reach a limit after ≈ 500 iterations, which corresponds to a depth of 16. Figure 5.11 is a 3 dimensional plot about the different parameter permutations, which are represented with the colored points. The color of the points correspond to their mean test F1 macro scores mapped on the color bar. This figure tells us that a high maximum depth combined with both low minimum samples per leaf and split result in a higher score. This means that the estimator cannot be overfitted on the dataset, which is an interesting scenario. For better scores, a more specific tree is needed which can be done by leaving this Decision Tree implementation in scikit-learn with its default parameters. Only the `min_samples_split` was fine-tuned because it changed from 2 to 4.

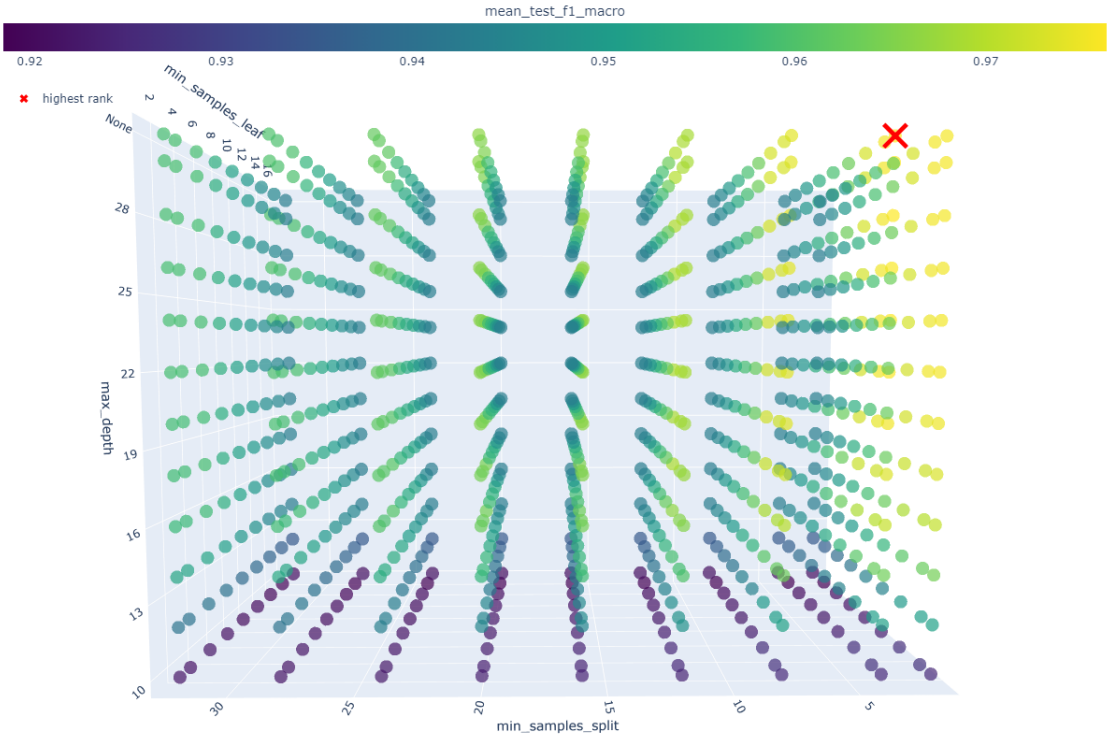


Figure 5.11: The result of `GridSearchCV`

5.3.2 Results

After the fine-tuning phase I validated and tested the model. The confusion matrix of this process is shown in Figure 5.12. It can be seen that the number of FN samples is good, as the goal is to reduce them as much as it is possible. There are also low numbers of attack misclassifications, which means that the classifier can almost perfectly separate different attack types. The number of FP samples is high for **DoS** (1), **PortScan** (2), and **Web-based** (5) attacks. The most significant from these three is **Web-based**, because the model finds $\approx 40\%$ more of this attack than it would be optimal. The validation and test scores generally do not have high deviation based on the characteristics on the dataset.

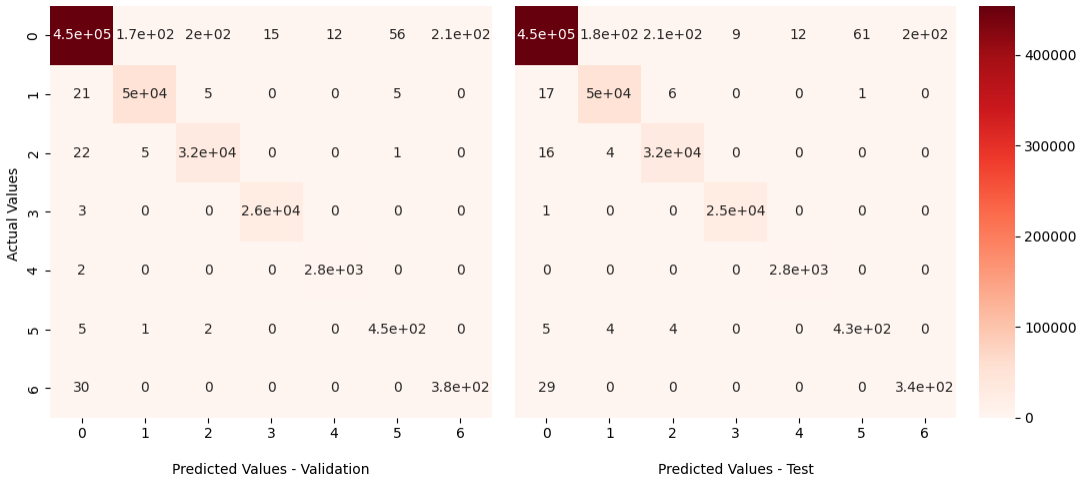


Figure 5.12: The confusion matrix of validation and test phases

Figure 5.13 shows the test classification success by ROC AUC OVR macro and average precision macro scores broken down to categories. It can be seen that the estimator scores are high by both of the metrics for the 5 most represented categories. For the other two categories, the scores start decreasing. The average precision score gets lower by a serious amount. A single Decision Tree cannot handle these categories but the main goal is to use the other estimators – which will be detailed later – to make classification better for **Web-based** (5) and **Bot** (6) categories.

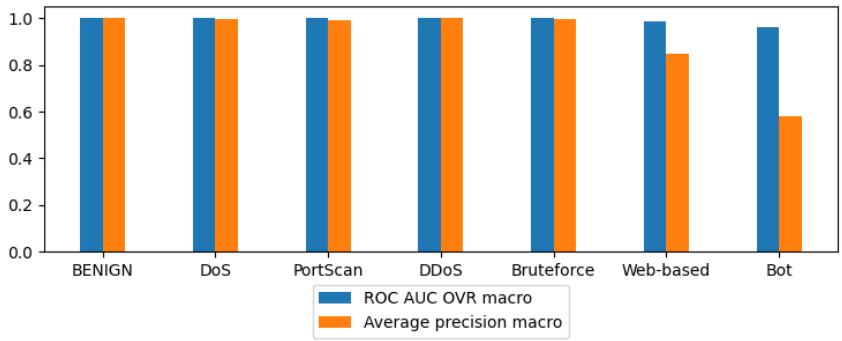


Figure 5.13: The ROC AUC OVR macro and average precision macro scores by categories on the test data

5.4 Random Forest

Random Forest [10] is an ensemble algorithm which becomes more accurate, than single Decision Trees by putting together the outcome of several – in many cases 100 or even more – trees. An ensemble method is a classifier constructed by combining multiple classifiers with the aim of having better performance, than a single one would have, i.e. scoring better by a chosen metric (see Section 5.1) because the time complexity, CPU, and memory usage will be significantly higher during the training stage [13]. In addition, the validation and evaluation will be slower but this gained complexity is, in general, a worthy trade-off because classifiers might not be trained too frequently in real life scenarios for resources to be an issue. However, when high speed processing is required, higher testing time can also be an issue, so complexity is not completely irrelevant. This issue might be solved, e.g. by the power of cloud computing. The idea behind the combination of multiple Decision Trees is that a single tree is not flexible enough when it comes to classifying new samples. I used `RandomForestClassifier` [55] from the scikit-learn package and I will use the parameter names from this implementation.

The Random Forest algorithm is a sophisticated way of evaluating multiple Decision Trees, meaning that its concept contains several ideas to make the trees more adaptable. The `bootstrap` parameter – when it is set to true – makes the classifier train each tree with a resampled training data with replacement [10]. This means that these bootstrapped datasets might have the same size as the original training data but they can contain each row multiple times as they are randomly selected. The samples not selected by bootstrapping are called out-of-bag (OOB) samples. These samples can be used to estimate generalization score, which can be enabled with `oob_score=True`. Also the maximum number of samples to build a bootstrapped dataset can be limited with the `max_samples` parameter. It is set to the same size as the whole training data by default. This parameter can lead to much lower execution times. The process of bootstrapping is shown in Figure 5.14. As it can be seen in the figure, Patrick and Gabriel got selected twice, while Juliet and George was not selected to the bootstrap data.

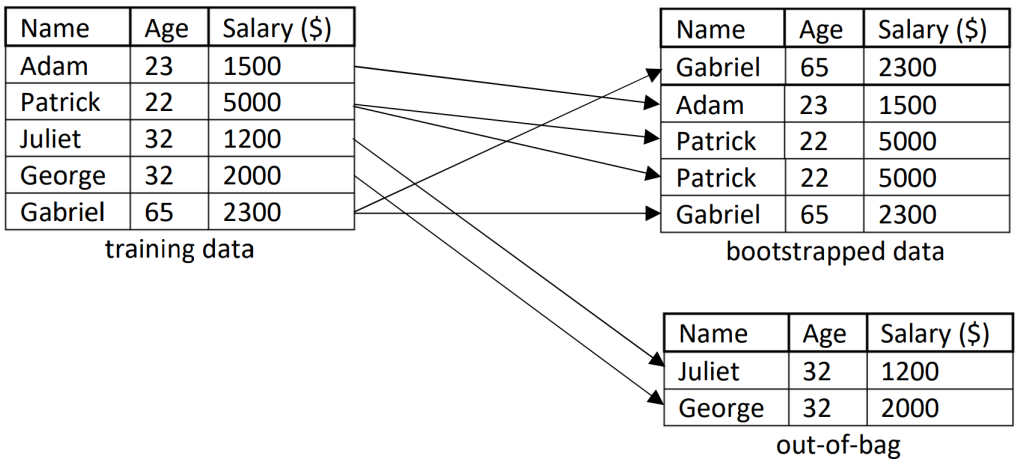


Figure 5.14: Example of bootstrapping

Other crucial parameters are `criterion`, `max_depth`, `min_samples_split`, and `min_samples_leaf`. These are inherited from the base estimator; their purpose is detailed in Section 5.3. A unique parameter of the `RandomForestClassifier` is `max_features`, which constraints the maximum number of columns used for searching the best split

on the nodes while training the trees. It is automatically set to $\sqrt{\texttt{n\_features}}$, where `n_features` is the number of columns. The selection of the feature subset is random for each splitting step on each tree. Another parameter is `n_estimators`, which tells the algorithm, how many trees should be constructed during training. Its base value is 100, which means that 100 trees will be formed. The effectiveness of increasing this parameter is limited because the performance usually plateaus. So in most cases, it is enough to look for the plateau in the fine-tuning phase. Using the `bootstrap=True`, `max_feature=`$\sqrt{\texttt{n\_features}}$ and `n_estimators=100` parameters results in highly diverse trees. This diversity can be observed, e.g. by the trees having different depths. Let $tree_0(x), tree_1(x), \ldots, tree_{\texttt{n\_estimators-1}}(x)$ be the predictions of the trees on sample x in the Random Forest, which is a sample to be classified. The following formula can be written to get the aggregated prediction of the trees proposed by Breiman [10]:

$$trees(x) = \frac{1}{\texttt{n\_estimators}} \sum_{i=0}^{\texttt{n\_estimators-1}} tree_i(x). \tag{5.13}$$

It is important to note that this implementation uses another technique for averaging, which will be detailed later.

The process, where the method uses bootstrapped data and aggregates their outcome, is called bootstrap aggregating or bagging [9]. The process is visualized in Figure 5.15. Bagging makes this method more flexible to new data. Another benefit of bagging is that it is less likely to overfit.

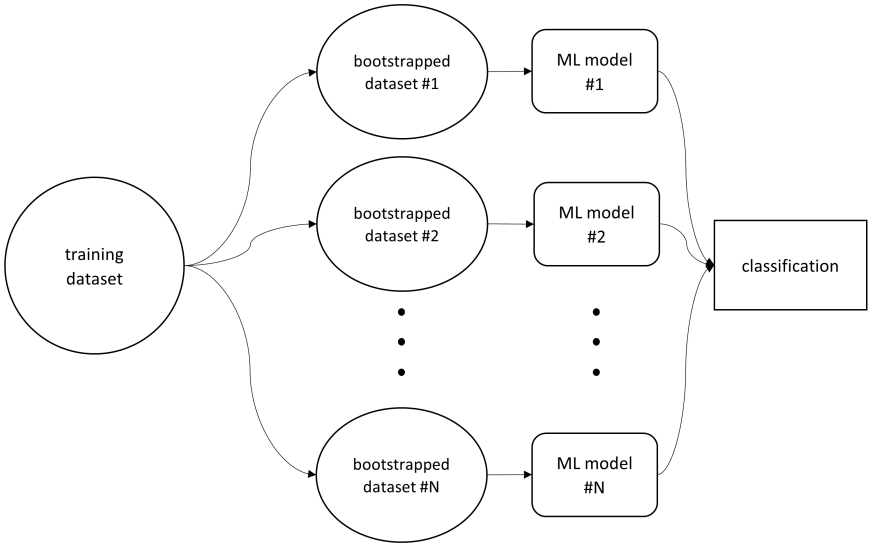


Figure 5.15: The process of bagging

The prediction of this implementation does not work the same way as Random Forest was originally proposed. It uses the average of the probabilistic prediction of each tree for classification [52]. This difference can also be noted while reading through the source code. The following snippet is from the original scikit-learn repository on GitHub[1].

[1] scikit-learn GitHub repository - ForestClassifier predict function `https://github.com/scikit-learn/scikit-learn/blob/3e6a39a73c2ca39e073e4b58117f59e92b3b2313/sklearn/ensemble/_forest.py#L824` (Accessed: 2022-12-05)

```
def predict(self, X):
    ...
    proba = self.predict_proba(X)

    if self.n_outputs_ == 1:
        return self.classes_.take(np.argmax(proba, axis=1), axis=0)
    ...
```

This function takes the prediction probability of each tree, which is an `n_samples` × `n_classes_` array consisting of probabilities indicating whether a sample is more or less likely to be a class based on the prediction of the estimator. These probabilities are calculated by the mean class probabilities of each tree. The `RandomForestClassifier` takes the class with the highest probability for the classification.

Slightly less important parameters made for customization are weights. One of them is `class_weight`, which is to make classification biased by weighting classes more or less. In this implementation, the `balanced_subsample` value is used to calculate weights automatically. These weights assigned to samples are assigned to classes using the formula `n_samples_bs / (n_classes_ * np.bincount(y_bs))` – as it is stated in the documentation of this classifier – where `n_samples_bs` is the number of samples in the bootstrapped dataset, `n_classes_` is the number of classes, `y_bs` is the labels for the bootstrapped data, and `np.bincount(y_bs)` is a 1D array containing the frequency of each class in `y_bs`. The other weight type is `sample_weight`, which can be added in the fitting stage. It requires a 1D array with the length of the training labels with a weight for each row.

`RandomForestClassifier` is parallelized, meaning that it can train multiple trees at the same time on different threads. This can be done because the construction of each tree is fully independent from the others. This functionality can be used by setting `n_jobs` to the number of destined threads to use. It is practical to use -1 as the value because then the model will execute on as many threads as it is possible on the given hardware.

5.4.1 Fine-tuning

As the previous section describes, Random Forest combines multiple Decision Trees in order to make a better prediction of the to-be-classified samples. Using more trees to find the results make ensemble learning more computationally complex. However, single trees in most of the cases cannot perform well enough. To tune the `RandomForestClassifier` I chose to use 2 of the formerly detailed parameters which are the following: `n_estimators` and `max_depth`. The chosen values are shown in Table 5.3. The tuning of `min_samples_leaf` and `min_samples_split` is also possible in `RandomForestClassifier`, however, I did not want to constrain the model this way because it creates lots of different trees (`n_estimators`) during training. To optimize their value is not feasible given the limited hardware resources. Also constraining them causes decrease in the outcome, which might be a worthy trade-off in real-life scenarios (for more generalized classification), but my goal was to make as good predictions as it is computationally possible. Every fine-tuning step included a base parameter set, which is the following: `criterion='entropy'`, `bootstrap=True`, `oob_score=False`, `n_jobs=-1`, and `random_state=42`. The fine-tuning phase lasted $\approx$ 8 minutes. It is recognizable that a new parameter for fine-tuning would radically increase the computational time, which is not small even with only 2 parameters considering that all threads were utilized for this task. The F1 macro score of the model with the tuned parameters is 0.97 on the validation set after fitting on the whole training set.

Parameter	Value range
n_estimators	5, 10, 25, 50, 100, 150, 200
max_depth	10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, None

Table 5.3: The used parameter ranges

I chose the range of `max_depth` according to the used value set in Section 5.3.1 because I wanted to see the differences between the trees made by Random Forest and a single Decision Tree. It can be seen in Figure 5.16 (for details on what the figures illustrate, see Section 5.3.1) that constraining the forest with `max_depth=10` already makes an improvement of $\approx 2\%$ regarding mean test F1 macro score.

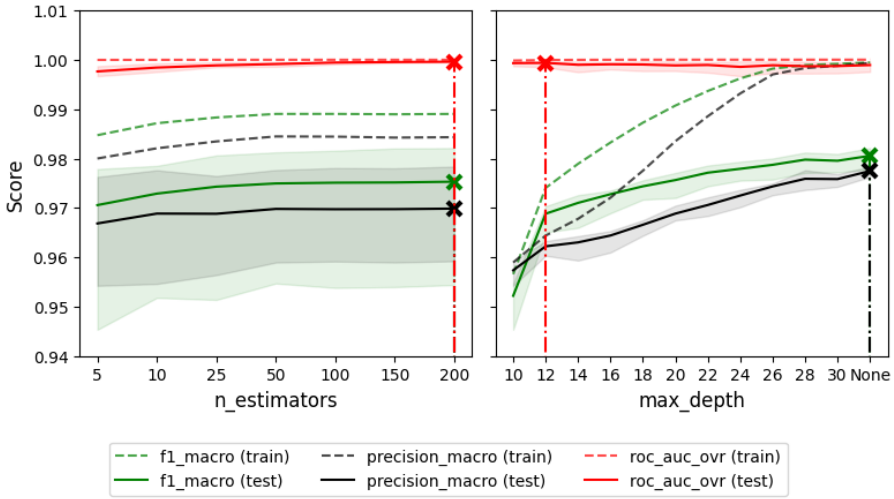


Figure 5.16: The result of grid search broken down to the chosen parameters

Random Forest works better with shallower trees than a Decision Tree, which is because there are obvious differences between the creation of a tree. E.g. one of them is that the Decision Tree uses all features for building a tree but Random Forest uses only the square root of the number of features. It can also be seen in this figure that there is a notable difference ($\approx 0.9\%$) by mean test F1 macro between `max_depth=10` and `max_depth=12`. After this threshold the score by this metric is slowly increasing until it reaches around 0.98. Mean test precision macro scores are also slowly increasing and it gains $\approx 1\%$ from 10 to `None`. Towards the end for both of the former metrics, the train and test mean scores are gaining distance, which means that overfitting is present. It depends on the actual scenario but in a real-life environment – where the testing phase might have more variance – it might not be suitable to use an overfitted estimator as the final estimator. In this case, the goal is to make as good predictions as it is possible and this deviation ($\approx 2\%$), when using maximum depth, is acceptable. The `max_depth=None` parameter was chosen by grid search. `DecisionTreeClassifier` made a tree with the maximum depth of 48 by the `max_depth=None`. In Figure 5.17, the trees within this forest are counted based on their depth. It can be seen in the figure that the majority of the trees are deeper than the single Decision Tree presented in Section 5.3.1 which is because of the previously mentioned random feature selection. It can also be seen that there is a high deviation between the shallowest and deepest trees (35 levels).

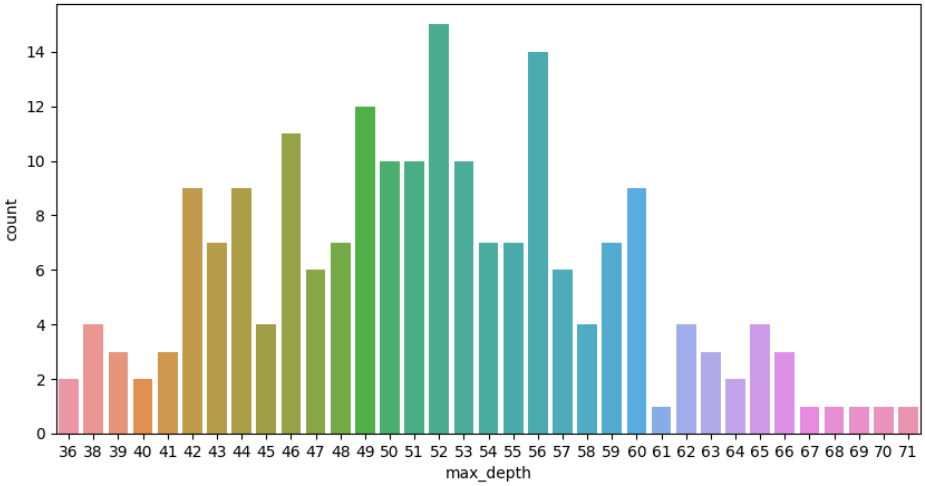


Figure 5.17: The occurrence of depths within trees constructed by `RandomForestClassifier`

The `n_estimators` parameter proved to be not as influential to the outcome. As it can be seen in Figure 5.16, the values by mean F1 macro and mean precision (both train and test) scores are almost constant. The only difference can be seen when we take `n_estimators=5`, which is generally a low setting for Random Forest but it has also great scores. There is always a trade-off when increasing the number of estimators because more estimators cannot ruin prediction capabilities but it surely increases training and testing time to the extent that it might not worth increasing it any longer. It can also be seen that mean ROC AUC OVR (both train and test) scores are almost perfect by both of the formerly mentioned hyperparameters. The final values for the fine-tuned parameters are `max_depth=None` and `n_estimators=200`.

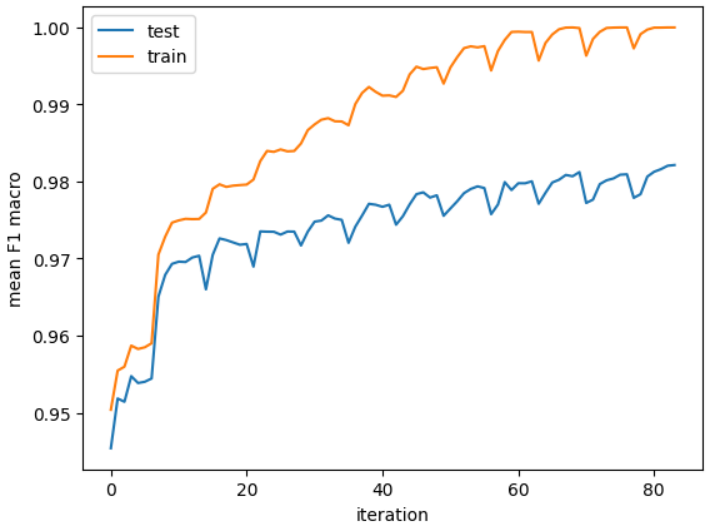


Figure 5.18: The mean test F1 macro scores of every iteration

The previously mentioned slow increase can also be seen on Figure 5.18. It is also more recognizable in this figure how the deviation between train and test gets bigger and bigger

converging to the end. The spikes represent the different `n_estimators` values for each `max_depth`. For details on what the figure illustrates see Section 5.3.1.

Figure 5.19 also shows that `max_depth` is more important of these two hyperparameters because if the depth is not too constraining then the classifier will get decent scores even with `n_estimators=5`.

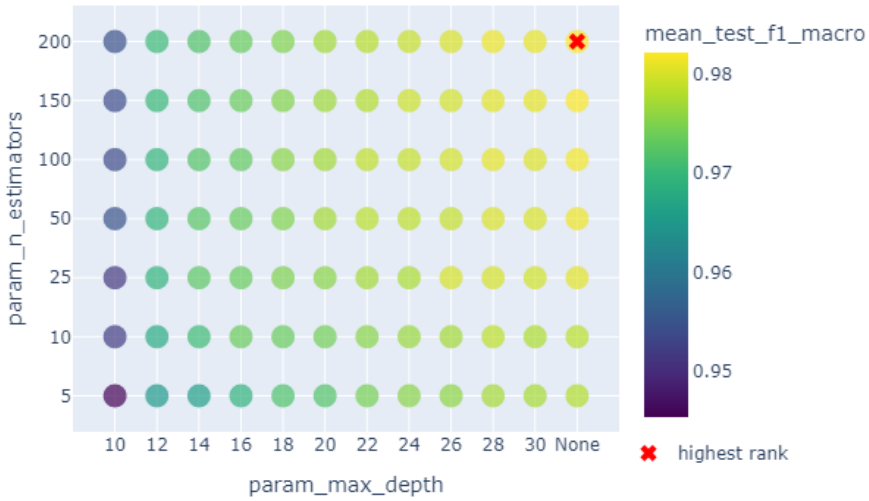


Figure 5.19: The result of `GridSearchCV`

5.4.2 Results

The validation and test classifications can be seen in Figure 5.20 with the help of a multiclass confusion matrix. The number of FP samples notably reduced compared to the classification by Decision Tree, shown in Section 5.3.1. However, the number of FN samples stayed around the same. **Bruteforce** (4) attacks could be detected almost perfectly with this method in the test dataset. The good separation is probably because of the characteristics given by brute force in general.

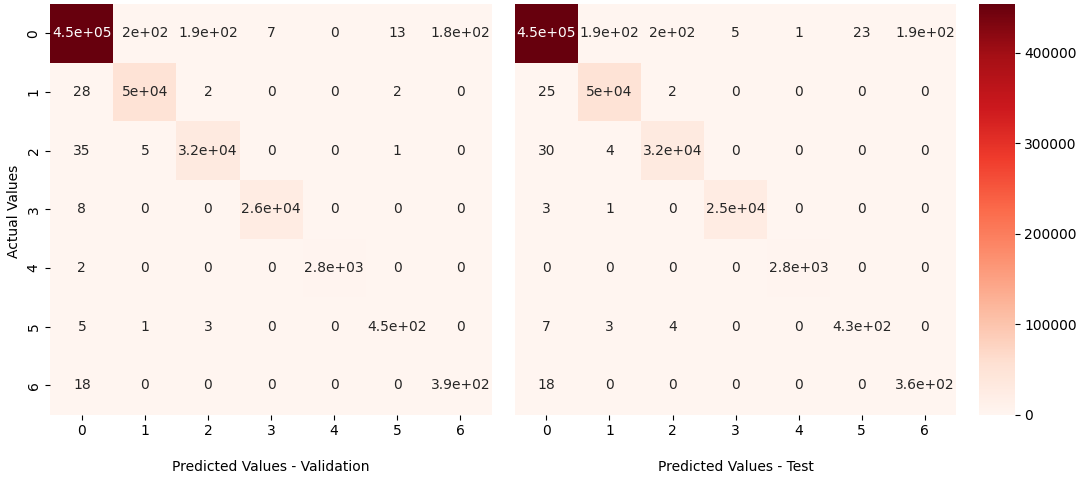


Figure 5.20: The confusion matrix of validation and test phases

It can be seen in Figure 5.21, that the average precision macro scores for **Web-based** (5) and **Bot** (6) categories increased compared to the single Decision Tree approach (see Figure 5.13). It is important to note that these classes are represented by low number of samples, meaning that their slightest misclassification will decrease the scores by a large amount.

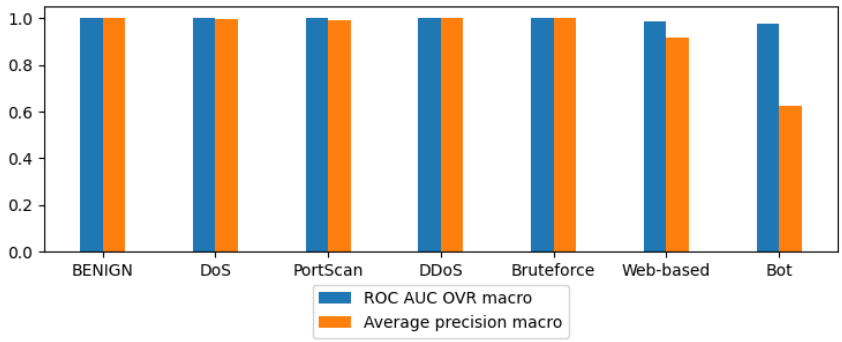


Figure 5.21: The ROC AUC OVR macro and average preciosion macro scores by categories on the test data

5.5 AdaBoost

Adaptive Boosting (AdaBoost) [21] is an ensemble learning method, which combines the prediction of multiple weak learners with boosting to make classifications [17]. The main idea of boosting is a sequential improvement by adding new weak learners that primarily target misclassified samples. Weak learners are models, which make predictions close to guessing, but a low degree of learning takes place. An example of a weak learner – which is frequently represented alongside AdaBoost – is a decision stump. A decision stump is a Decision Tree (see Section 5.3) which has only one level (i.e. `max_depth=1`). This means that in each one of these stumps only 1 feature is used to split the data into 2 parts. As I have mentioned, the learners are not trained independently of the others. Boosting is based on their sequential training. Each one of the trees are correcting the weaknesses of the previous models. This characteristics makes, it impossible to parallelize AdaBoost. The main concept behind the algorithm is to use weak learners as estimators. I want to test the capability of this model boosting strong learners to make their predictions more accurate. This will be detailed in the following sections. The simplified version of boosting in AdaBoost is shown in Figure 5.22.

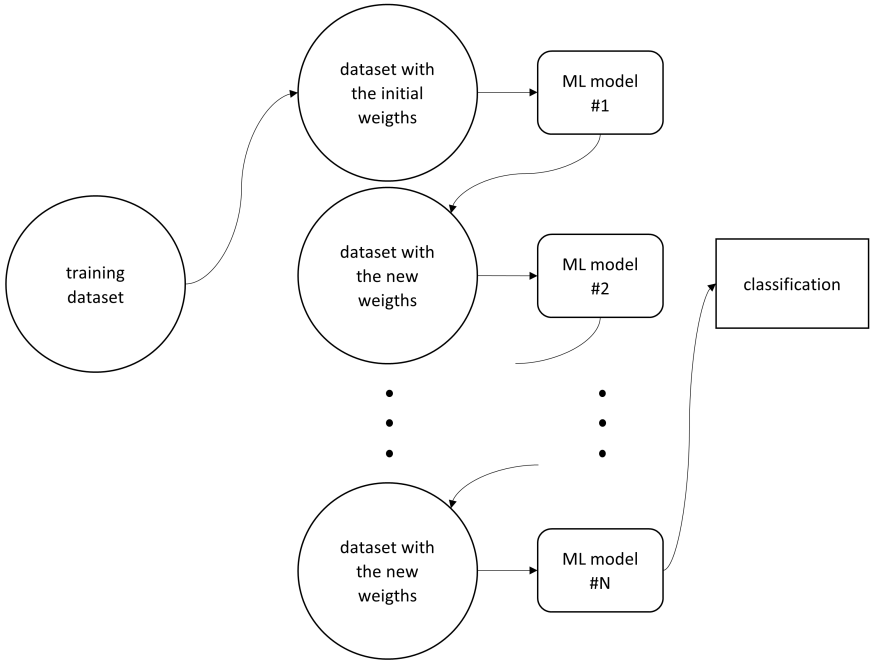


Figure 5.22: The process of boosting

I used `AdaBoostClassifier` from scikit-learn [45], which implements two variations different from the originally proposed AdaBoost algorithm: SAMME (Stagewise Additive Modeling using a Multi-class Exponential loss function), which is discrete and SAMME.R, which is real AdaBoost. The main difference is that these variants support multiclass classification. They were proposed by Hastie et al. [21]. The default algorithm for this classifier is SAMME.R and it can be changed through the `algorithm` parameter. In my work, I sticked to the default algorithm because it is claimed that it typically converges faster than SAMME with lower test error [50].

If SAMME.R is used, it is required to have a base estimator which can predict classes with probabilities. I used `DecisionTreeClassifier` (see Section 5.3), which is suitable because it has a method called `predict_proba`. This method returns an `n_samples`

× n_classes matrix, where each row will contain values indicating the probabilities of the given sample belonging to the different classes. Here n_samples is the number of samples and the n_classes is the number of categories in the training dataset. These probabilities will be used to calculate the weighted probabilities – by which the data will be classified – after fitting in each boosting iteration. This makes a key difference between SAMME and SAMME.R because in SAMME.R, estimators will not have dedicated weights – other than 1 –, only the formerly mentioned probabilities will be used. The used base estimator can be set with the base_estimator parameter. It defaults to a stump of DecisionTreeClassifier, which is not ideal for every case. Further exploration of this parameter will be shown in Section 5.5.1. Another useful parameter is the learning_rate $\in (0, 1]$. It defines how much impact should a boosting iteration have. With the n_estimators parameter, we can set the number of destined iterations within the algorithm. This parameter makes predictions more stable. Higher n_estimators values are not recommended because the model cannot be parallelized. Thus, the model does not have an n_jobs parameter which would utilize more threads for the algorithm.

In contrast with the previous models, the weights of AdaBoost are adaptively changed in each step [21]. The samples start with equal weights and after each boosting step, weights of correctly classified samples decrease multiplicatively, while weights of incorrectly classified samples increase multiplicatively. Let w be the weight for each sample in the training dataset. Its initialized value will be $w = \frac{1}{\texttt{n\_samples}}$. The update of weights can be controlled via changing the value assigned for learning_rate. More formal and thorough description of the weight update mechanism is out of the scope of this thesis.

A fundamental part of the scikit-learn implementation is to get an error score of the fitted estimator at the start of every boosting step. If estimator_error <= 0 – as it is in the source code – is True, then early stopping happens[2]. Early stopping can be executed immediately when an algorithm achieves perfect scores within the destined iterations (i.e. n_estimators). This error score can be calculated with the following equation:

$$\texttt{estimator\_error} = \frac{\sum_{i=0}^{f-1} w_i}{\sum_{j=0}^{\texttt{n\_samples-1}} w_j}, \tag{5.14}$$

where f is the length of the falsely predicted samples. Since weights are strictly positive, the lowest estimator_error is 0, when everything is correctly predicted.

5.5.1 Fine-tuning

As I have mentioned previously, AdaBoost is not parallelizable resulting in a higher time complexity. This is even more true during the fine-tuning process, when a lot of AdaBoost instances has to be fitted and evaluated. One advantage is that AdaBoostClassifier does not have too many parameters to tune, although its base estimator, which is a DecisionTreeClassifier, has several hyperparamters itself. Differently from the fine-tuning phase of other classifiers, I will include 3 fine-tuning approaches which center around DecisionTreeClassifier mainly. The one with the highest scores – combined with a reasonable time complexity – will be further detailed in Section 5.5.2. In this section I will tune learning_rate and n_estimators, while using the same values of max_depth, min_samples_split and min_samples_leaf that were obtained during the

[2]scikit-learn - early stopping in real AdaBoost https://github.com/scikit-learn/scikit-learn/blob/b01f018c9b4e963d3b897906fd90a5de42c1a6b8/sklearn/ensemble/_weight_boosting.py#L595 (Accessed: 2022-12-05)

fine-tuning of `DecisionTreeClassifier`. The used parameter value ranges for each approach – considering only the `AdaBoostClassifier` parameters – is presented in Table 5.4. Also the used default parameters for each AdaBoost model were `base_estimator` which was set to the actual `DecisionTreeClassifier` and `random_state=42`. For each Decision Tree, I chose `criterion='entropy'`, `class_weight='balanced'`, and `random_state=42` as base parameters. The meaning of the used Decision Tree parameters were described in Section 5.3.

Parameter	Value range
n_estimators	$5, 10, 25, 50, 100, 150, 200$
learning_rate	$0.1, 0.3, 0.5, 0.7, 0.9, 1.0$

Table 5.4: The used parameter ranges.

The following approaches are not ordered based on their contribution, instead I listed them chronologically as I have experimented with them.

5.5.1.1 First approach

This approach implements the main idea behind AdaBoost, which is stump learning. Stump learning is when the predictions are made with trees having only one level [17]. This means that only one statement divides the dataset in each tree. Constructing stumps is the least time consuming method compared to the approaches, which will be described later in Section 5.5.1.2 and Section 5.5.1.3. However, there are serious downsides for multiclass classification as the following figures will show. The fine-tuning phase lasted ≈ 9 minutes. The tuned parameter values for this approach are the following: `learning_rate=0.1` and `n_estimators=10`.

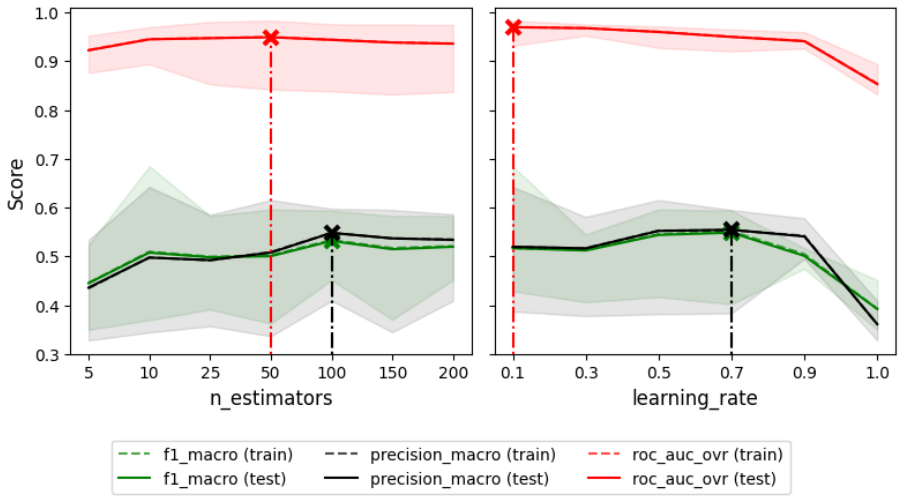


Figure 5.23: The result of grid search broken down to the chosen parameters

In Figure 5.23 it can be seen that both mean F1 macro and mean precision macro are low, which means that the model has bad classification potential. These metrics on `n_estimators` both have high deviation between minimum and maximum mean scores through the entire value ranges, which means that the model is completely biased by how the `StratifiedKFold` cross-validation (see Section 5.2) splits the dataset. Also, it is too

sensitive to little changes, which is not ideal because in real world scenarios the continuity of predictions is important to guarantee the performance of the used method. The `learning_rate` parameter has also high deviation except for 0.9 and 1.0. Mean ROC AUC OVR scores were lower than for the methods discussed in earlier sections for both parameters. Train and test scores by all three mean metrics are the same and are also low, which indicates underfitting.

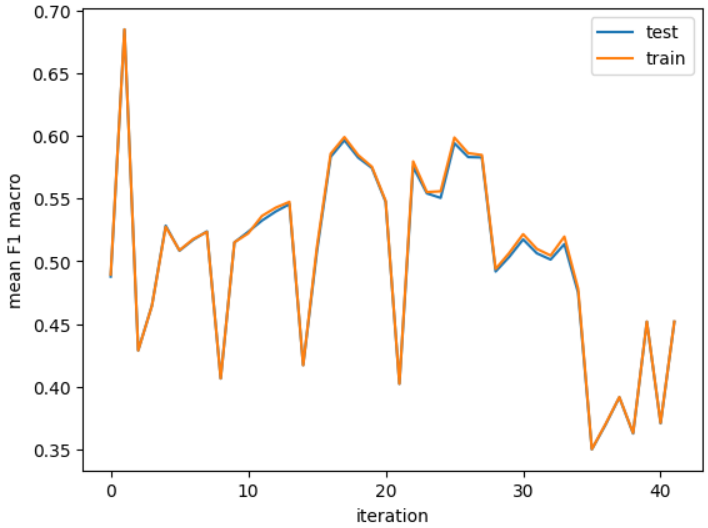


Figure 5.24: The mean test F1 macro scores of every iteration

Both train and test mean F1 macro scores are low which can be seen through grid search iterations on Figure 5.24. It can be deduced from this, that at the end of the fine-tuning phase even the model with the best scores will be underfitted.

5.5.1.2 Second approach

To find a more usable AdaBoost model, I first made experiments to see that how the `max_depth` of the base estimator changes the outcome of the `AdaBoostClassifier`. This usage of AdaBoost might be unconventional based on the properties proposed earlier in this section as the base estimators will not remain to be stump learners. The main idea behind this approach is that I should be able to specify a Decision Tree with a just appropriate depth. I started exploring this parameter which resulted in a to-be-expected outcome: the scores and also the computational time increased. The fine-tuning phase lasted $\approx$ 30 minutes, which is around 3 times longer than the first approach described in Section 5.5.1.1. The resulting values for the parameters are the following: `learning_rate=1.0` and `n_estimators=200`. The `max_depth` of the `DecisionTreeClassifier` changed to 7.

Firstly, I explored the `max_depth` of `DecisionTreeClassifier` in the range of 1 to 15. The exploration metric was F1 macro score. The outcomes of this can be seen in Figure 5.25. I chose 7 as `max_depth` because it is the first good score after the knee point of the graph. The evaluations were made with the same base parameters as described earlier but with the additional `max_depth` modification.

After selecting the base estimator with the ideal `max_depth`, the same fine-tuning process was used for tuning `n_estimators` and the `learning_rate`. As Figure 5.23 shows, we can achieve radically better results by all three metrics with this new maximum depth. Using 5 and 10 for `n_estimators` causes the predictions to be slightly uncertain as it can be seen

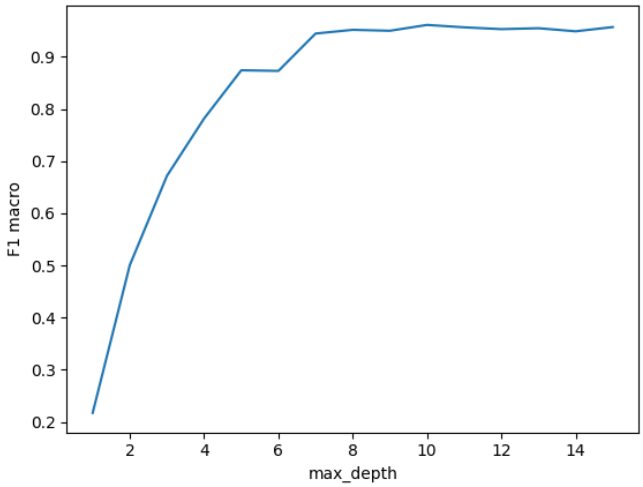


Figure 5.25: The exploration of `max_depth` within the base estimator of `AdaBoostClassifier`

on the difference of minimum and maximum mean scores. The `n_estimators` parameter reaches a plateau after 100, however, the ranking will consider 200 as the best value for the parameter. It can also be seen that with increasing `n_estimators` the model becomes more reliable. The `learning_rate` does not have reasonable contribution to the result.

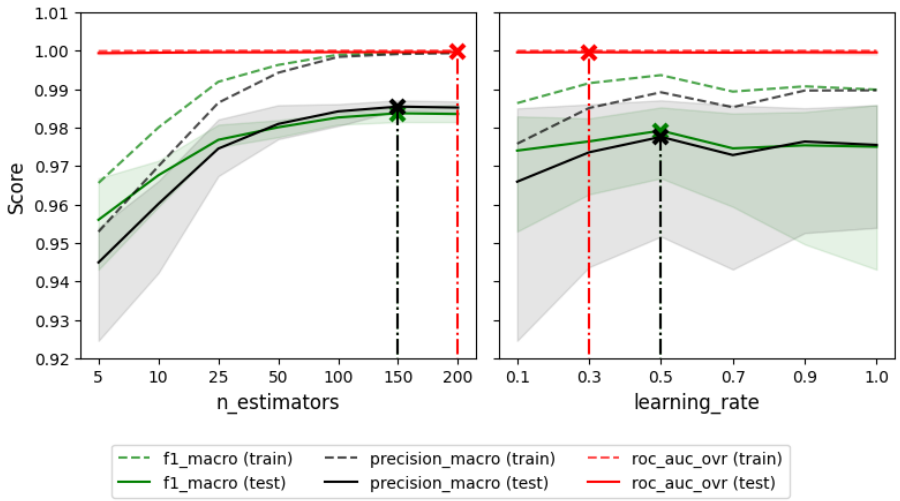


Figure 5.26: The result of grid search broken down to the chosen parameters.

As it can be seen in Figure 5.24 the `n_estimators` will be better at changing the prediction success out of these 2 parameters. It can also be deduced that `learning_rate` does not have a significant contribution to the outcome because the graph will stay almost identical until the end. Though, the predictions tend to be lower when higher numbers of `n_estimators` are combined with lower `learning_rate`.

The chosen parameters for the model were `n_estimators=200` with the `learning_rate` being 1.0, which is arguable in the light of Figure 5.26, however, I chose to stick with the outcome of grid search because I would like to make comparable results.

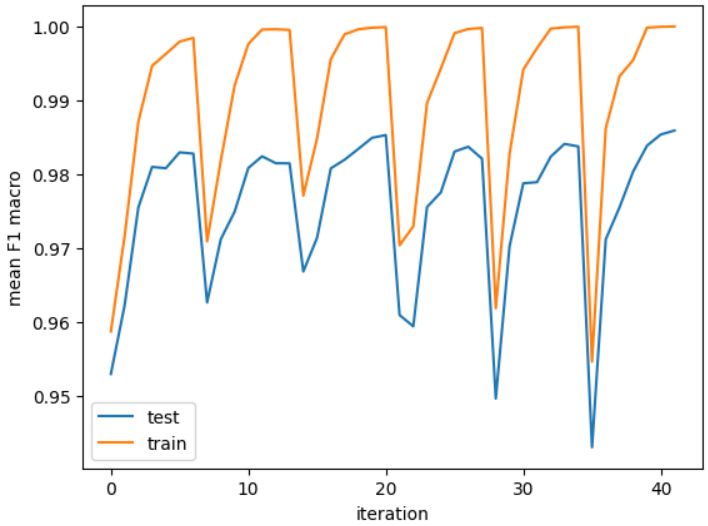


Figure 5.27: The mean test F1 macro scores of every iteration

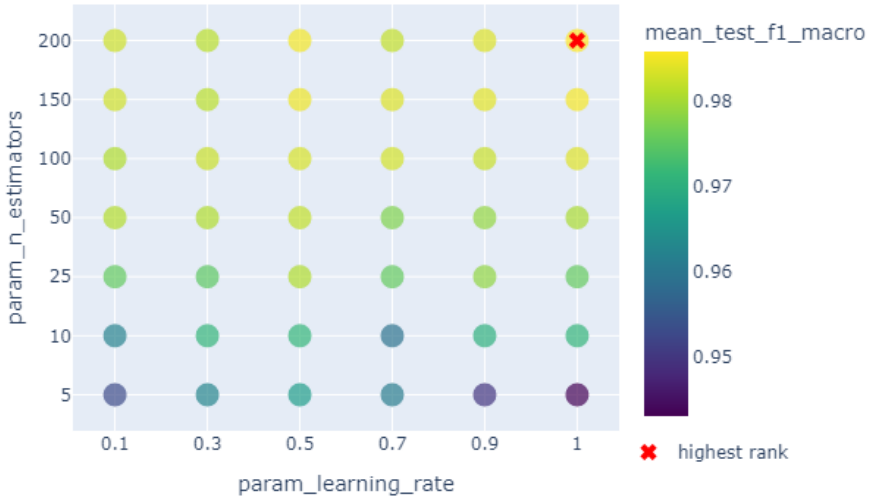


Figure 5.28: The result of `GridSearchCV`

5.5.1.3 Third approach

The main concept behind this approach is that I have fine-tuned a single `DecisionTreeClassifier` in Section 5.3.1. I thought that it would be interesting to see how `AdaBoostClassifier` can affect the predictive success of an ideal fully developed Decision Tree. This `DecisionTreeClassifier` had `max_depth=16`, `min_samples_leaf=1`, and `min_samples_split=4`. The fine-tuning of this AdaBoost variant required a lot of computation time: $\approx$ 53 minutes. The tuned parameter values are the following: `learning_rate=0.9` and `n_estimators=200`.

It can be seen in Figure 5.29 that for all three of the metrics the train and test scores are identical which means the model is too specific. I expected this result because the base estimator alone is too specific for boosting.

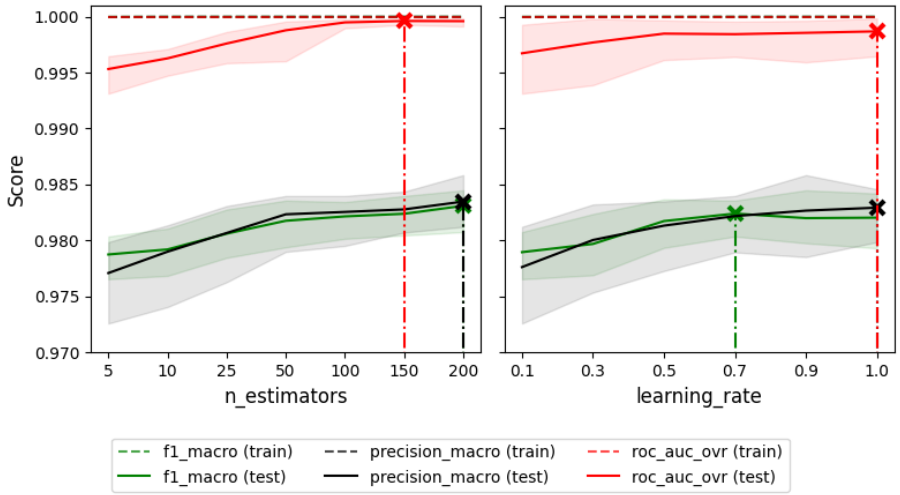


Figure 5.29: The result of grid search broken down to the chosen parameters

The specificity of the model can be seen clearly in Figure 5.30, which shows that train scores are constantly 1.0. Although, test scores are affected by just a notch by the used parameters.

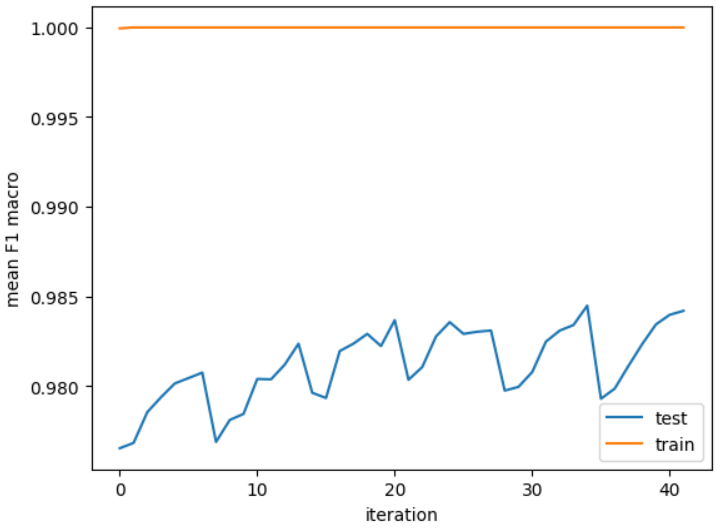


Figure 5.30: The mean test F1 macro scores of every iteration

This model would be perfectly suitable for this dataset, however, the goal was to make more generic estimators which might give a more predictable classification when a slight change occurs in the traffic. Its run time also makes it very unlikely to be deployed in a real life environment.

5.5.1.4 Summary

Thus, the first approach showed underfitting and the third approach granted too specific classifications, I chose the best estimator from the second approach. It will be evaluated on the test set in Section 5.5.2. The parameters of the resulting model with their values are the following: `learning_rate=1.0`, `n_estimators=200`, `random_state=42`, and the

`base_estimator` a `DecisionTreeClassifier` with the following parameters and values: `criterion='entropy', max_depth=7, class_weight='balanced', random_state=42`. The F1 macro score of the model with the tuned parameters is 0.95 on the validation set after fitting on the whole training set.

5.5.2 Results

After the fine-tuning, I decided to use the second approach as it provided the most reasonable estimator. This section will detail the validation and test phase of that model. It can be seen in Figure 5.31 that the number of FN samples increased compared to single Decision Trees which is unfortunate and makes this model not suitable for this scenario. The increment is present in **DoS** (1) and **PortScan** (2) categories.

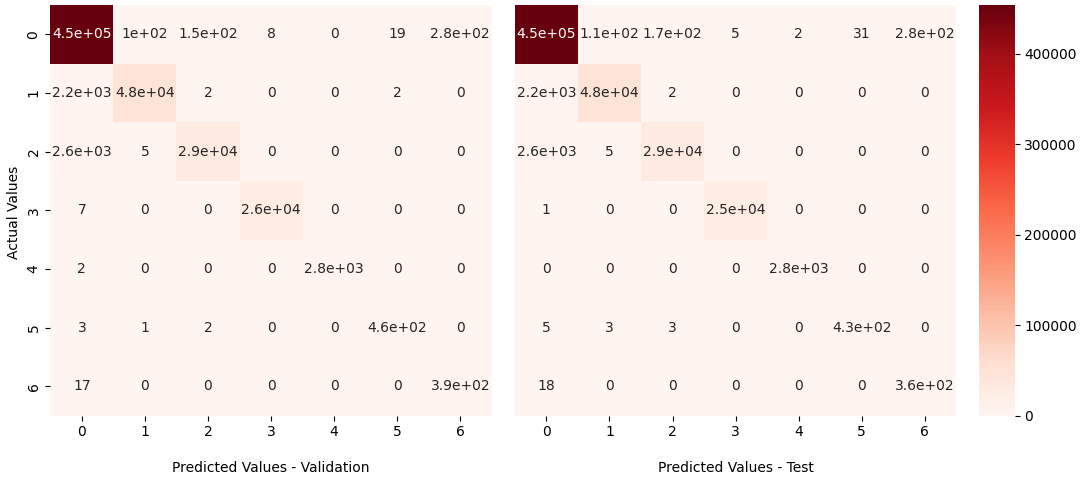


Figure 5.31: The confusion matrix of validation and test phases

Based on Figure 5.32 only the **DDoS** (3) and **Bruteforce** (4) categories could be optimally classified. The first three categories got worse compared to the Decision Tree approach and also the classification problems in **Web-based** (5) and **Bot** (6) categories did not get better – in terms of average precision macro score – in the process. A possible solution would be to increase the `max_depth` parameter in order to make better predictions, however, the time complexity is high even in this case which will be further detailed in Section 5.7.

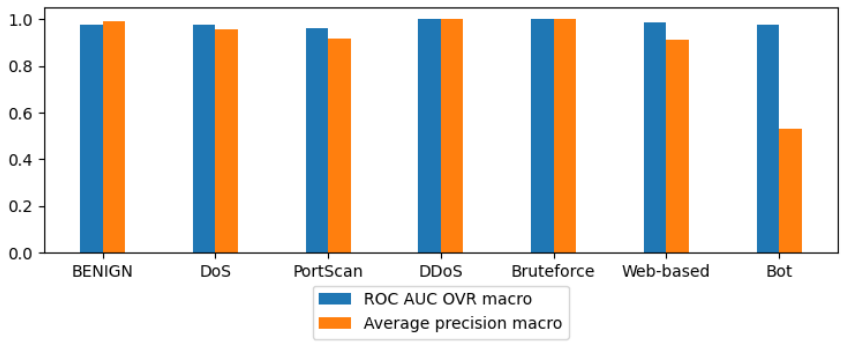


Figure 5.32: The ROC AUC OVR macro and average precision macro scores by categories on the test data

I would like to mention that I have evaluated the third approach too and it made similar output as the Random Forest method mentioned in Section 5.4.2 but the time complexity got out of hands because the fit time of one model was ≈ 32 minutes on the whole training data.

5.6 XGBoost

The eXtreme Gradient Boosting (XGBoost) [12] model is the most complex approach used in this thesis. In this section, I will summarize the top-level properties of the model with some details. The further exploration of the model is not in the scope of this thesis. I used `XGBClassifier` from the xgboost package [70]. The reason why I picked this classifier is because I wanted to include a model based on gradient boosting as the last classifier in the comparison. The first model I tried was `GradientBoostingClassifier` from the scikit-learn package, however, I quickly noticed that it is not as optimized as I wanted it to be. In order to provide a better result, I tried this `XGBClassifier` as it is a more recent model. Another criterion was to have a model which can utilize multiple CPU cores and also the possibility of GPU usage was important. The power of the GPU computation is that it can usually make run times much lower. It is important to note that not every segment of a machine learning model can be successfully converted to be executed on GPU. Although most of the algorithms used in XGBoost can be accelerated by GPU. This feature is rather helpful in the fine-tuning phase as I will demonstrate in Section 5.6.1. This ML model has also a very good reputation in the machine learning community based on its performance and the wide variety of customization possibilities required by a classification or regression task.

Similar to AdaBoost, XGBoost is based on boosting (see Section 5.5), however, it builds different kinds of weak learners. It builds gradient boosting trees which are also shallow Decision Trees but they are regression trees and built in a different manner [12]. First an initial score has to be assigned to XGBoost which can be set with `base_score` that defaults to 0.5. The key component of building gradient boosted trees is calculating residuals for each row by getting the differences between the observed values (i.e. the label of each row) and the predicted values (which is initially the `base_score`). The residuals will be used to calculate the so-called similarity score for each leaf that is present in the actual tree. To calculate similarity scores the following equation can be written [63]:

$$similarity = \frac{(\sum_{i=0}^{n} r_i)^2}{n + \texttt{lambda}}, \tag{5.15}$$

where n is the number of residuals present on the leaf, `lambda` is a regularization parameter, and r_i is the ith residual on the leaf. This formula is only correct if the loss function is the squared error (see Equation (6) in the work of Chen et al. [12] for the more general formula). In this section, I will use this assumption, as it is the most common loss function.

Regularization is used to make the algorithm more conservative to change. While doing so, the variance of the model decreases and the bias increases, i.e. it is used to prevent overfitting. XGBoost contains L1 (Lasso) and L2 (Ridge) regularization for this purpose. The parameter for L1 regularization is `alpha`, which defaults to 0 and the parameter for L2 regularization is the previously mentioned `lambda` and its default value is 1. Further detailing of regularization is not in the scope of this thesis as the regularization parameters will be left on their defaults, while tuning and evaluation.

For each value of each parameter, XGBoost could make a stump and the model could calculate the gain from the similarity scores of each stump – to see, which feature has the best splitting potential –, but it would be extremely time consuming. To avoid this XGBoost implements the Approximate Greedy Algorithm. Greedy algorithms in general are searching locally for the best possible solution, meaning that they are not taking into account its effect in the long run. Firstly, the amount of splitting points to be considered is reduced by this algorithm, which is achieved by using only quantiles of the values.

Secondly, the algorithm picks the highest gain each time while constructing the trees. The gain is calculated by the following equation [63].

$$gain = similarity_{left} + similarity_{right} - similarity_{root}, \tag{5.16}$$

where $similarity_{root}$ is the similarity score for the root node of the tree, $similarity_{left}$ is the left child of it and $similarity_{right}$ is the right one. Roughly speaking, $similarity_{left} + similarity_{right}$ is the highest when the residuals are homogeneous inside the left and right sets. XGBoost chooses the best split according to the gain. The model can use multiple cores for this computation by splitting the dataset into multiple parts and executing them in parallel. It summarizes the outcomes with an approximate histogram and defines weighted quantiles in it, which can be used in the previously described greedy algorithm.

It is important that these trees can be pruned. Pruning of a tree is used for making the classification more generic by limiting the formation of a new leaf. It can be set with the `gamma` parameter – which defaults to 0 – and it controls the minimum amount of gain an additional leaf node should have to be added to the tree.

For calculating the output value for a sample the following equation can be written [63].

$$output = \texttt{base\_score} + \sum_{j=0}^{\texttt{n\_estimators}} output_j * \texttt{eta}, \tag{5.17}$$

where `n_estimators` is the number of trees created in the learning phase of XGBoost, `base_score` is the initial score – which is 0.5 by default –, `eta` is the learning rate and $output_j$ is the output value in the jth tree, which can be calculated with the following equation [63].

$$output_j = \frac{\sum_{i=0}^{n} r_i}{n + lambda}, \tag{5.18}$$

where n is the number of residuals in the leaf node, `lambda` is the L2 regularization parameter and r_i is the ith residual in the leaf. Without regularization and learning rate, we would simply add the average of the residuals successively from each tree. `lambda` and `eta` make it so that we only take a step toward this local optimum.

XGBoost implements several techniques to make execution faster when the used data is huge. Formerly I have mentioned some of them, e.g. Approximate Greedy Algorithm and parallelization. It also uses Cache Aware Access, which optimizes the cache memory of the CPU. The main benefit of this is that the CPU can interact with its cache memory with the lowest latency compared to memory or page files. Another important asset of the model is the support for Out-of-Core computation when the data is too large to fit into the cache and the RAM. It compresses the dataset to be used from the page file on the storage. This makes communication faster and with a good CPU the decompression-latency trade-off is favourable because the CPU can decompress faster than the data could be read. Also, it can compress multiple blocks of the dataset and store them on separate drives. There are several parameters which can directly reduce computation time as well. One of them is `subsample` (defaults to 1, meaning that it will use the whole dataset for training), which is to control the amount of data used for training. Also, there are parameters to control the number of features used while building the trees. The `colsample_bytree` parameter constrains each tree, `colsample_bylevel` constrains each level and `colsample_bynode` constrains each node with a fraction of the whole feature set to be used.

XGBoost is widely customizable, which results in a high number of parameters. I have written about some of them in this section, however, there are some more to be added. One of them is `max_depth`, which constrains the number of levels a gradient boosting tree can have and it defaults to 6. It controls the fit of the model on the dataset. It is notable that higher numbers might result in high memory usage. Another parameter is `min_child_weight`, which sets the minimum sum of instance weight in a child node. It constrains the splitting of the child nodes. If the squared error is the loss function, then it is the minimum number of samples on a leaf node. To use the histogram based approach detailed earlier the `tree_method` should be set to `'hist'`; to use it with GPU it should be set to `'gpu_hist'`. To use the exact greedy tree construction approach `tree_method='exact'` should be used. In this thesis I want to explore both CPU and GPU computation so I will include `'exact'` and `'gpu_hist'` in Section 5.6.2. With `predictor` we can set the prediction architecture. For CPU prediction it should be set to `'cpu_predictor'` and for using GPU computation it should be set to `'gpu_predictor'`.

5.6.1 Fine-tuning

I chose to tune `n_estimators`, `max_depth` and `eta` which is more than the used hyperparameters for Random Forest and AdaBoost, however, considering the number of hyperparameters the `XGBClassifier` has it is not sufficient. To reduce time complexity of fine-tuning, I chose to ignore L1 and L2 regularization and `min_child_weight` also. The value ranges used for tuning the classifier can be seen in Table 5.5. The base parameters were the following for each iteration: `tree_method='gpu_hist'`, `gpu_id=0`, `objective='multi:softmax'`, `random_state=42`, `seed=42`, `n_jobs=-1` and `predictor='gpu_predictor'`. As it can be seen, I chose the `n_estimators` values low compared to other ensemble methods detailed before. Choosing them is based on a preliminary evaluation with this estimator. Higher values did not result in considerably better results and also they were far more time consuming. I tuned the selected hyperparameters of `XGBClassifier` with GPU acceleration as it is a less time consuming approach. However, it is important to note that this classifier cannot run only on GPU meaning that CPU will also be used. The total fine-tuning time was ≈ 11 minutes. After fine-tuning, the optimal parameter values chosen by grid search were `n_estimators=30`, `max_depth=11`, and `eta=0.75`. The F1 macro score of the model with the tuned parameters is 0.99 on the validation set after fitting on the whole training set.

Parameter	Value range
n_estimators	$10, 15, 20, 25, 30$
max_depth	$4, 5, 6, 7, 8, 9, 10, 11, 12$
eta	$0.1, 0.25, 0.5, 0.75, 1.0$

Table 5.5: The used parameter value ranges

As it can be seen in Figure 5.33, the increment in `n_estimators` does not make a huge difference in the outcome. All three metrics stay high from 10 to 30 but around 1% improvement can be noticed from the lowest to the highest parameter setting in terms of mean precision macro and mean F1 macro scores. Also, it can be seen that the deviation between minimum and maximum scores is higher than usual which means that the model might not be perfectly certain. This unfolding tends to close while increasing the value, which is reasonable because relying on more estimators (i.e. gradient boosting trees) in the classification process will always lead to a more stable solution. As the figure shows, increasing `max_depth` will make a bigger contribution in the fine-tuning process as it boosts

mean F1 macro scores by $\approx 2\%$ from its minimum value to its maximum. The minimum and maximum scores also get closer towards the end, which is also reasonable as a more developed tree will be more accurate in classification even with gradient boosting trees. The `eta` parameter has around the same amount of significance in the tuning process as `max_depth`, however, a bigger `eta` means a more stable F1 macro score. The ROC AUC OVR for all 3 of the parameters gives a different result than for the estimators described before (see Section 5.3.1, Section 5.4.1, and Section 5.6.1). The classification by this score tends to be less stable for these parameters when their values are smaller and more stable towards the end.

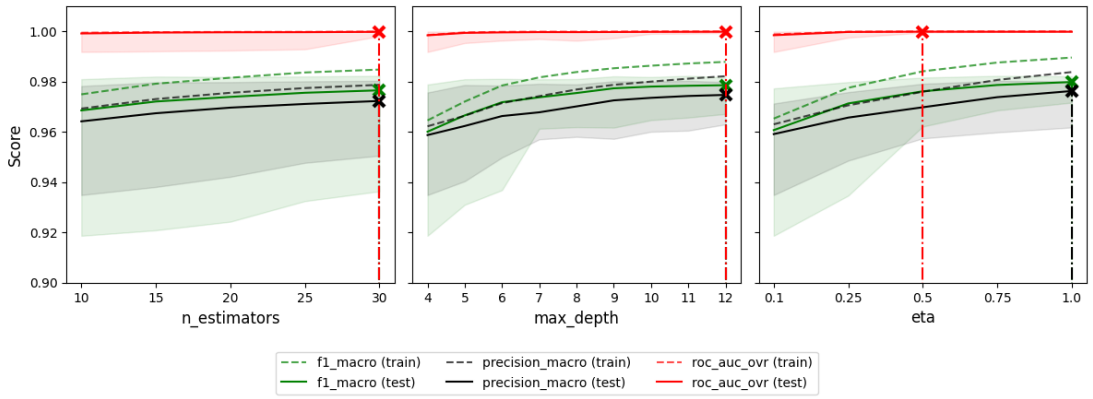


Figure 5.33: The result of grid search broken down to the chosen parameters

In Figure 5.34, we can see that the difference between training and testing increases with higher train performance in general. As it is not a high deviation, I will consider the outcome as an optimal parameter set given the constraints of my environment. The wider peaks in the figure represent the value changes within the `n_estimators` parameter range. The importance of `max_depth` and `learning_rate` tends to be higher when the `n_estimators` value is higher.

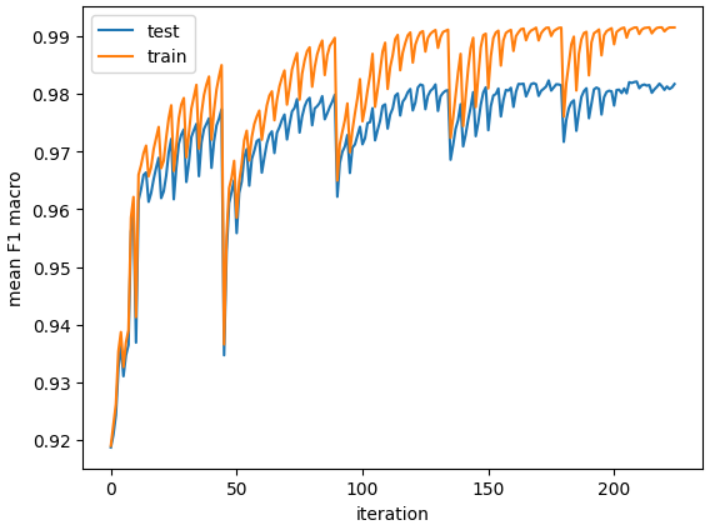


Figure 5.34: The mean test F1 macro scores of every iteration

Figure 5.35 shows that a higher `eta` and `max_depth` will result in better scores each time but `n_estimators` is not that effective in optimizing the results. A high `n_estimators`

combined with both high `learning_rate` and `max_depth` performs the best based on my measurements. Tuning only the `n_estimators` parameter would make the training and testing procedures longer with almost no positive consequences.

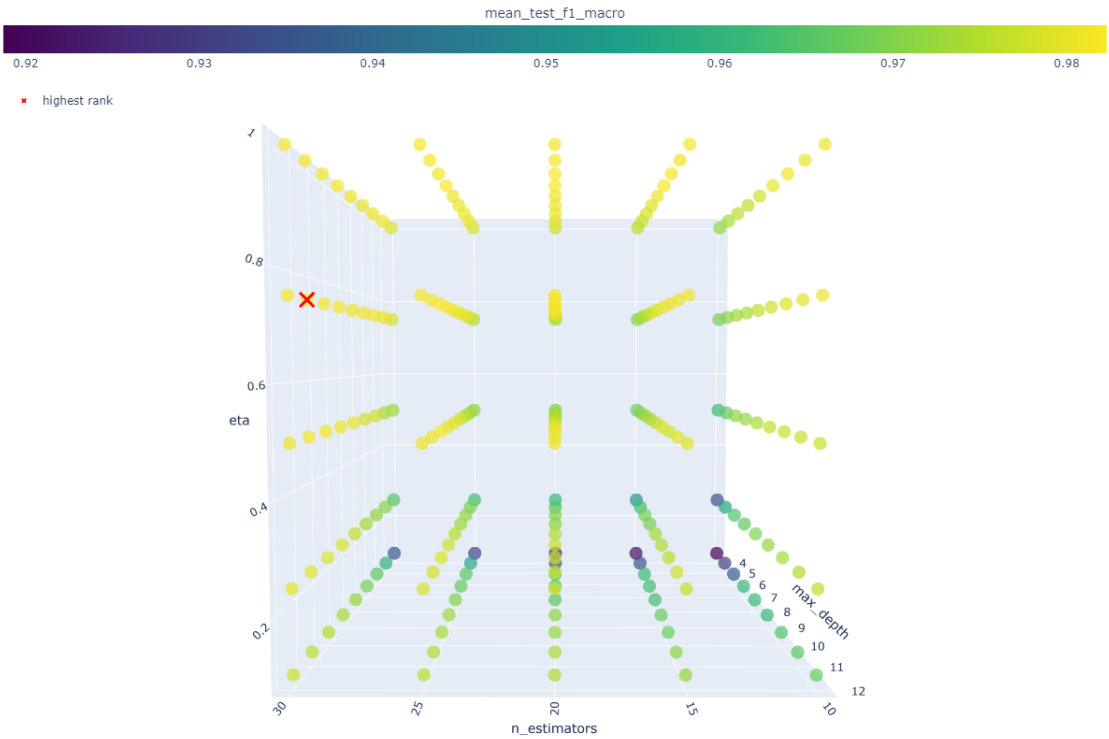


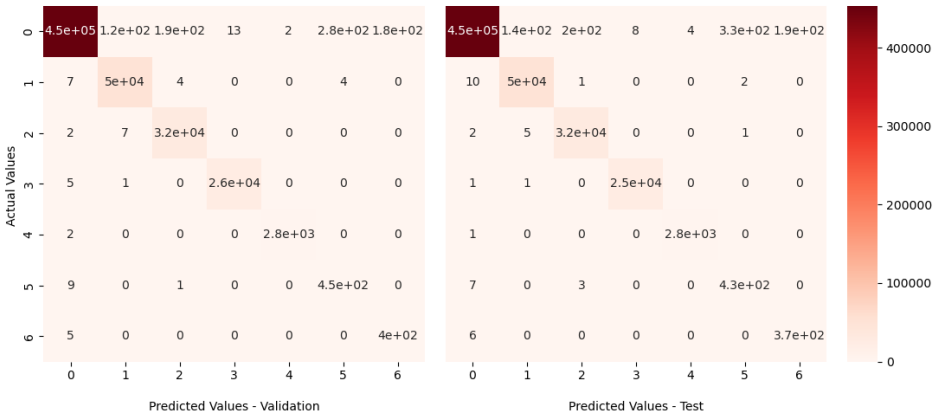
Figure 5.35: The result of `GridSearchCV`

5.6.2 Results

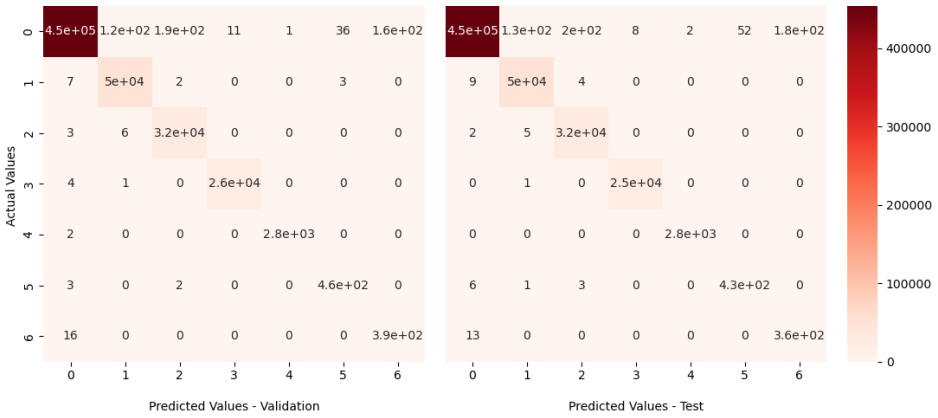
After fine-tuning, I validated and tested the model. As previously mentioned, I used both `'exact'` and `'gpu_hist'` approaches for the `tree_method` parameter. The `'gpu_hist'` uses a histogram-based estimation calculated on GPU which aims to be faster and suitable for large datasets. The used dataset is not huge considering the sizes used in real life scenarios but I would like to present the differences that these two values make. The model was tuned with `'gpu_hist'` but I assume that the tuned parameter values will be appropriate for `'exact'` too. To summarize, the only difference between the two model is the value of `tree_method` parameter.

As it can be seen in Figure 5.36, the number of FN samples is low for each of the categories in both scenarios. The main difference can be spotted when we look at the FP samples. The histogram based approach makes more FP classifications than the exact approach. This is because the former classification is based on a kind of estimation. This phenomenon is mostly present within the misclassification of **BENIGN** (0) samples to **Web-based** (5) attacks which gets fixed with the `'exact'` approach. Neither of these tree construction algorithms can assess the FP cases within the **Web** category.

Figure 5.37 shows that the difference between the two approaches is that the **Web-based** (5) category has better average precision macro score. This is a worthy trade-off as it will be detailed in Section 5.7.

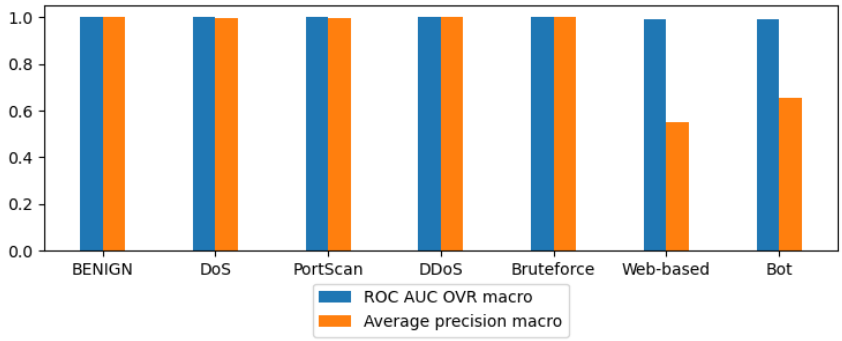


(a) GPU histogram-based tree construction

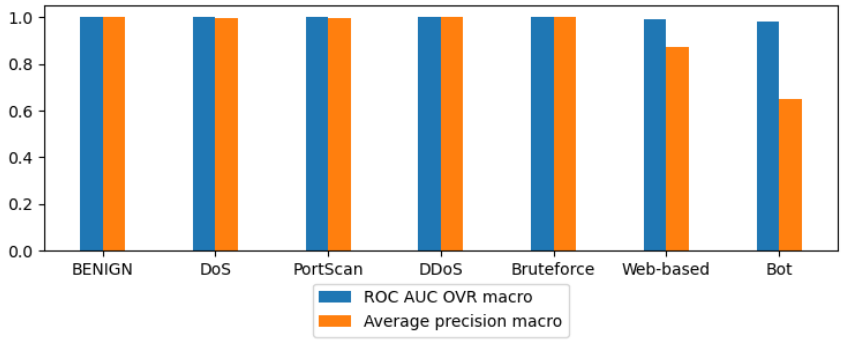


(b) exact greedy tree construction

Figure 5.36: Results represented with confusion matrices



(a) GPU histogram-based tree construction



(b) exact greedy tree construction

Figure 5.37: The ROC AUC OVR macro and average precision macro scores by categories on the test data

5.7 Summary

In this section, I would like to summarize the performance of the classifiers by two measures: time complexity and classification success. An additional plot will also be granted at the end of this section which shows the effectiveness of the classifiers used in this thesis by the detection of TN samples. After that, I would like to compare my results to some results published in other papers.

As it can be seen in Figure 5.38, the different models resulted in different train and test times. The plot for `DecisionTreeClassifier` (**DT**) shows a relatively small training time and it can predict within the least time. This is because it uses only one fully developed tree. `XGBClassifier` with `'gpu_hist'` (**XGB:gpu_hist**) took the least time to be trained which is the result of GPU acceleration. This model also grants a low test time. One difference of the `'exact'` parameter (**XGB:exact**) is that it takes 38.6 seconds more to train, however, the trade-off is worth it as it will be presented later. The `AdaBoostClassifier` with the second approach (**ABC**) takes way too much time to train (1400 seconds) and also, the test time is much higher than for the other models which is the result of trees being too deep in the model and also the lack of parallelization. This makes **ABC** unusable for this dataset considering a real life scenario. It is important to note that the hardware capabilities of my computer are not optimized for IDS purposes, however, I assume that these values for time complexity are too large for even architectures built for this purpose. `RandomForestClassifier` (**RF**) – as it is referenced in many cases – is an all-around method suitable in various scenarios. It has reasonable train and test times despite the fact that it constructs 200 trees.

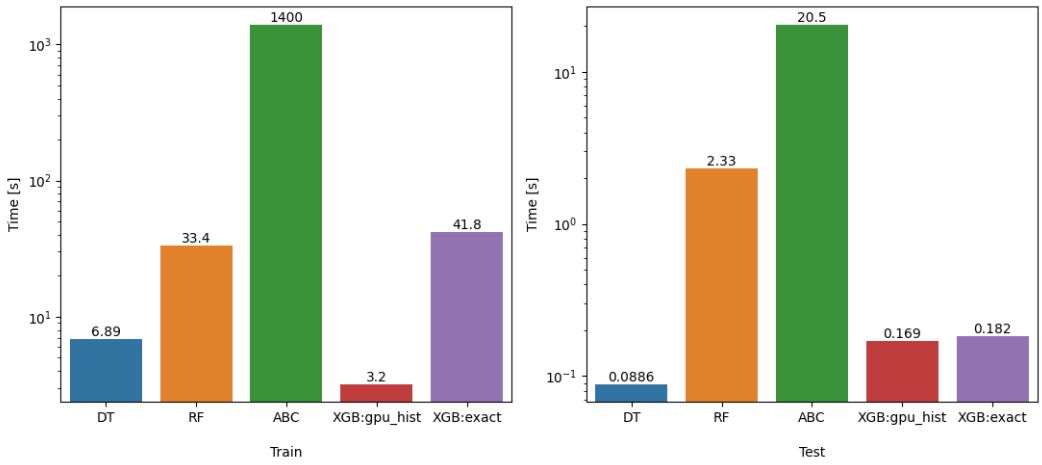


Figure 5.38: The train and test times of the models

Figure 5.39 shows the scores achieved with each estimator by the metrics used previously: precision with macro averaging, recall with macro averaging, and F1 score with macro averaging. It can be seen by the scores that **XGB:gpu_hist** produces the lowest precision macro and F1 macro, which is because it makes more FP classifications than the other models but the train time of the model is outstanding. As it was stated before, it is generally the decision of the developer whether the model is applicable for the use-case; the anomaly-based approaches usually have a relatively high amount of FP samples. **DT** provides good base scores with only one tree, however, **RF** and **XGB:exact** boosts all metrics up by 0.01 which can be useful in some cases. The difference between the latter two approaches are the train and test times which make RF the better. `AdaBoostClassifier` with the second approach gives lower scores than a single Decision Tree which makes this

model not suitable for this dataset. Also, as I have mentioned earlier when I evaluated the third approach (see Section 5.5.1.3), it gave similar results to **RF** and **XGB:exact**, however, the used time for training was even higher.

If time would be the only criterion, I would choose **XGB:gpu_hist** for my main model, however, if all train and test times and scores are important, then **RF** is the most suitable. If the versatile customization of the model is also needed, then **XGB:exact** should be used because this model is built for the complex constraining of its parameters and performed well on the dataset. I would not use **DT** because the more trees are used, the more certain the classification of a model is.

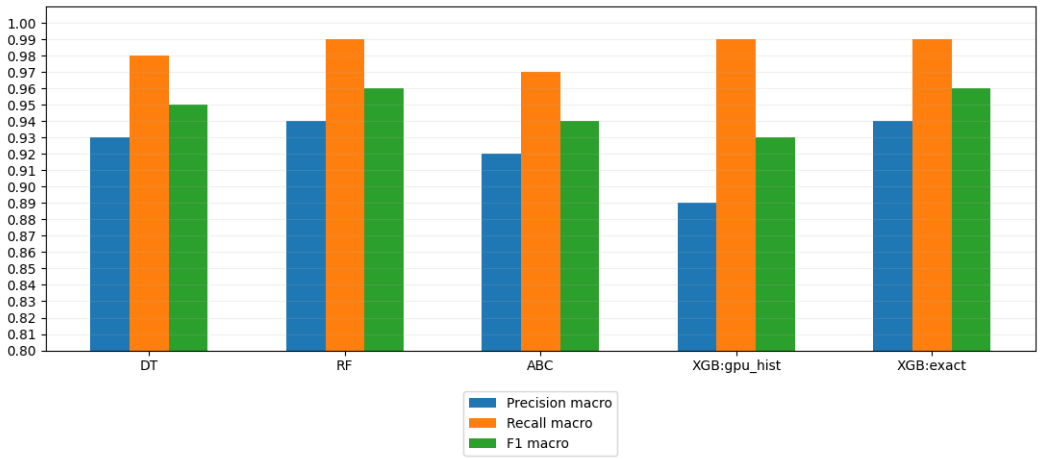


Figure 5.39: The scores achieved on test dataset for each model

Lastly, I would like to show the detection counts of false negative samples in Figure 5.40. After every evaluation step, I stored the FN samples of the classification. Then I could count that how many times a sample could be found as false negatively classified. The number of FN samples ideally would be 0 but it is not a realistic expectation. To give a fair representation of the models, I chose to use only **XGB:exact** from the two XGBoost approaches alongside **DT**, **RF** and **ABC**.

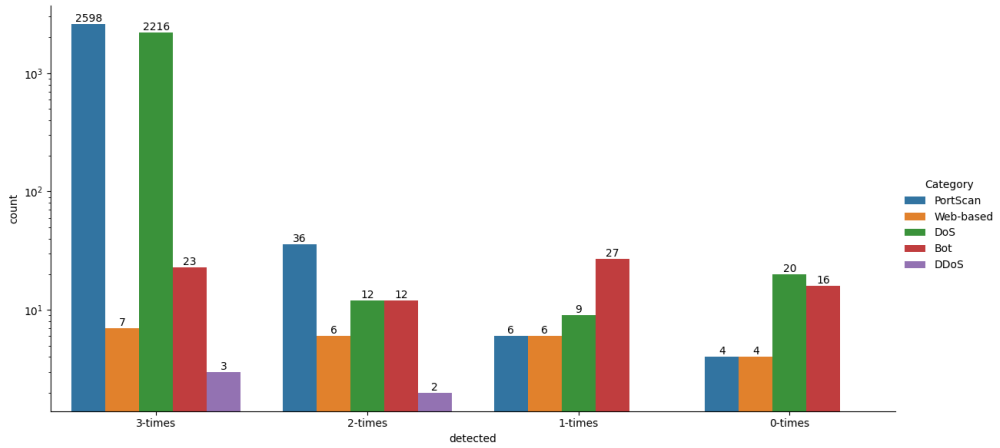


Figure 5.40: The grouping of false negative samples by the times that they could be detected

If the sample was present 4 times in the resulting dataset, then I plotted it as 0-times detected. This means that none of the estimators could classify those samples, which

means that these attacks would get through every anomaly-based intrusion detection system composed by the models. This is fortunately a relatively small set of data. The figure shows that **Bruteforce** samples could always be classified as attacks, which is a good result. Also, only a few port scan and web attacks are getting through. Larger numbers of Denial of Service and bot attacks could not be classified as attacks. This is a good result even with **DoS**, because a **DoS** attack might not be usually successful within 20 flows. Also, there were more **DoS** attacks than other attacks.

5.7.1 Comparison

Bhattacharya et al. [7] used a different approach on feature elimination. Their strategy was to use PCA and PCA-firefly algorithms. In their work, they proposed several models combined with this preprocessing method but I chose only the two best performing models. These are *XGB-PCA and Firefly* and *RF-PCA and Firefly*. Figure 5.41 presents their accuracy and recall macro scores next to mine which are **RF** and **XGB:gpu_hist**. I chose the histogram-based XGBoost because the paper states that GPU was used for the models. The comparison is not perfect as I chose to drop 2 attack types – as it was mentioned previously in Chapter 4 – but their representation does not make a significant difference. It is because these attack types are not represented enough well in the dataset. The differences can be spotted by the recall macro metric which shows that my models are scoring significantly better. The accuracy score does not have that high deviation in this case. I found that lot of researchers are showing the classification success of their models by accuracy on this dataset. This measure is not powerful enough in my opinion because accuracy does not take into consideration the imbalance of the dataset. It should be combined with precision for a better representation or using F1 score instead.

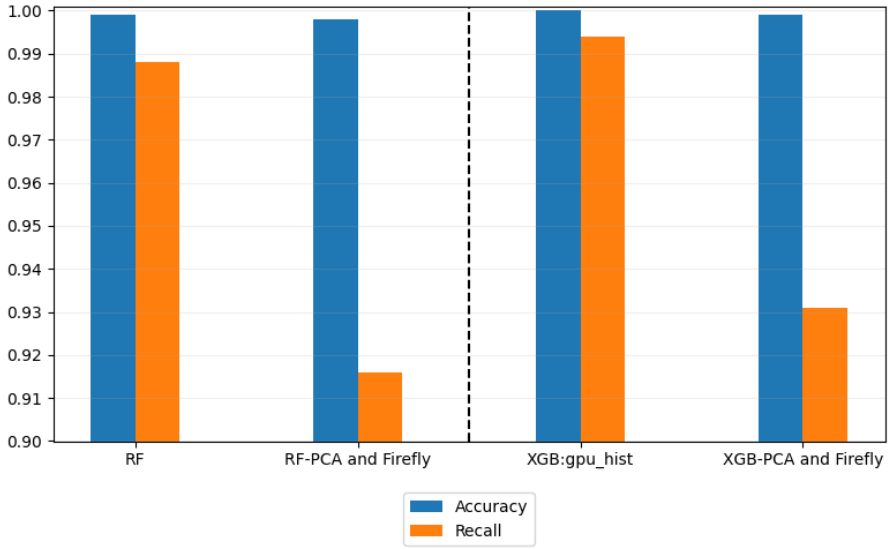


Figure 5.41: The comparison between the proposed models of Bhattacharya et al. and my models

Panwar et al. [36] also used a specific technique in their work to reduce the number of features. Their goal was to use as minimal number of features as it is possible. They used the Weka machine learning software for the models. I chose to include the two best performing models from their work, which are *J48 (CSE with Naive Bayes)* and *J48 (CSE with J48)*. I compared them with **RF** and **XGB:exact** as these models are the best by

F1 score on CICIDS2017 in my thesis (see Figure 5.42). The difference that makes the comparison not perfect is that they used separate models to detect **BENIGN** from each category, however, I have made single models for multiclass classification. This way they could reduce the number of features in each train set according to the needs of a category to be classified correctly, which makes their run times lower. Also, they could tune the hyperparameters of the models for each grouping, which improves the classification of the categories. These measures are not likely in a real life scenario as it is not a realistic task from an IDS to distinguish only one attack from the normal behaviour.

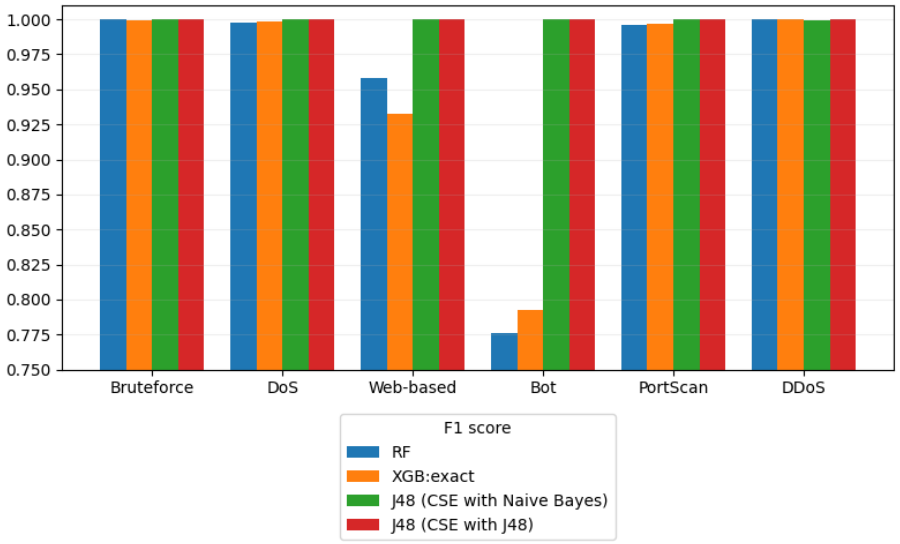


Figure 5.42: The comparison between Panwar et al. models and my models broken down by categories

The included models from the paper published by Panwar et al. made significantly better classifications on categories with lower representatives (i.e. **Bot** and **Web-based**) by F1 score. The **DDoS** category could be classified by my proposed method slightly better, which is a good result considering the differences in our classification strategies. Detecting DDoS attacks has a big emphasis in an anomaly-based IDS, especially today, when these attacks are occurring more frequently. For the other categories, there is only a low amount of deviation between our scores.

In Figure 5.43, another comparison is present, which is based on the classification of **DDoS** attacks. I used the work of Roopak et al. [43] for this comparison. They used neural networks for classification and proposed 4 different deep learning models in this context, which are the following: *1dcnn* (1D Convolutional Neural Network), *mlp* (Multi-layer Perceptron), *lstm* (Long Short-Term Memory), and *cnn+lstm* (Convolutional Neural Network combined with Long Short-Term Memory). The used models utilize `Keras` on `Tensorflow`. These estimators generally represent more complex models than the ones proposed in my thesis. I chose **XGB:exact** for this comparison as it was the best scoring estimator for this attack type. As it can be seen, all deep learning models scored significantly worse than my model which made almost perfect scores by both precision and recall. The *cnn+lstm* model was the best among them which scored outstandingly high by these metrics compared to the other deep learning models. The run time of these models was not detailed, thus I cannot make conclusions based on it, however, the complexity of correctly building a neural network can be high and requires in-depth knowledge of several

type of these networks. My approach with **XGB:exact** grants a more easily interpretable model which scores better on **DDoS** attacks with only a short fine-tuning phase.

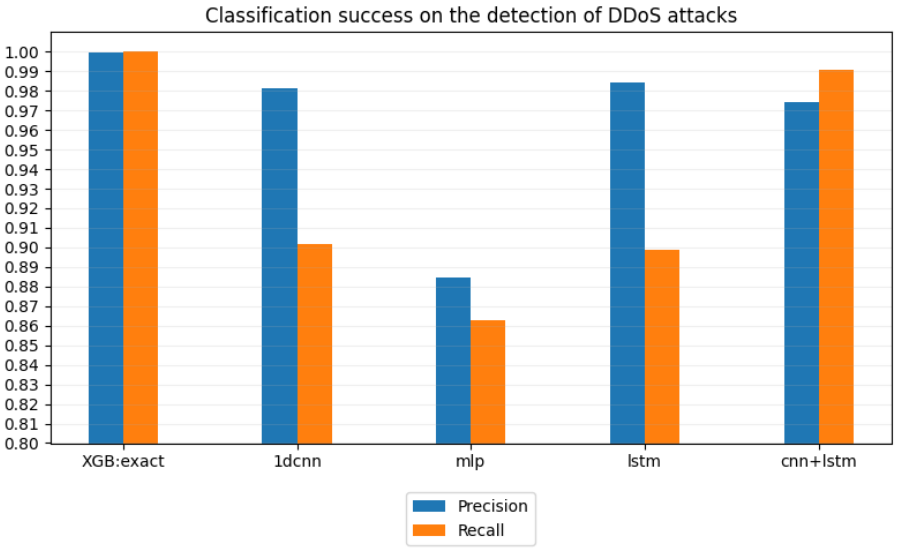


Figure 5.43: Comparison between the models proposed by Roopak et al. and my models based on **DDoS** classification

Chapter 6

Corrections in CICIDS2017

As I have mentioned in Section 3.4, the original CICFlowMeter contains several flaws which result in a less reliable dataset. In this chapter, I would like to demonstrate the difference between the original CICIDS2017 and the corrected **RP** dataset proposed by Lanvin et al. [29]. The procedures and figures will stay the same as the ones previously discussed in Chapter 5 during each step.

6.1 Preprocessing

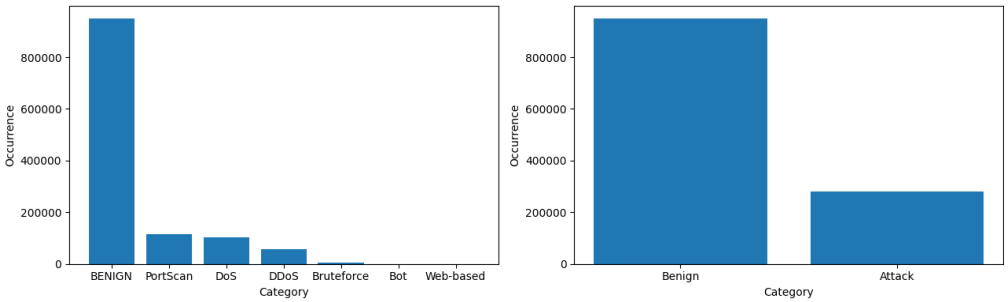
The corrected database consists of 5 CSV files representing the days of the workweek. The first notable difference after concatenating the corrected CSVs is that the resulting dataset contains less samples for each attack type (excluding **Heartbleed**) than the original dataset had and this is because of the corrections described in the work of Engelen et al. [15]. The corrected dataset also contains the additional port scans (i.e. *Port_scan*) described in Section 3.4. The representation of attack types can be seen in Table 6.1.

Category	Number of occurence
BENIGN	1597836
PortScan	159579
DoS Hulk	158470
DDoS	95144
Port_scan	64185
DoS GoldenEye	7567
DoS slowloris	4001
FTP-Patator	3973
SSH-Patator	2980
DoS Slowhttptest	1742
Bot	738
Web Attack - Brute Force	151
Infiltration	32
Web Attack - XSS	27
Web Attack - Sql Injection	12
Heartbleed	11

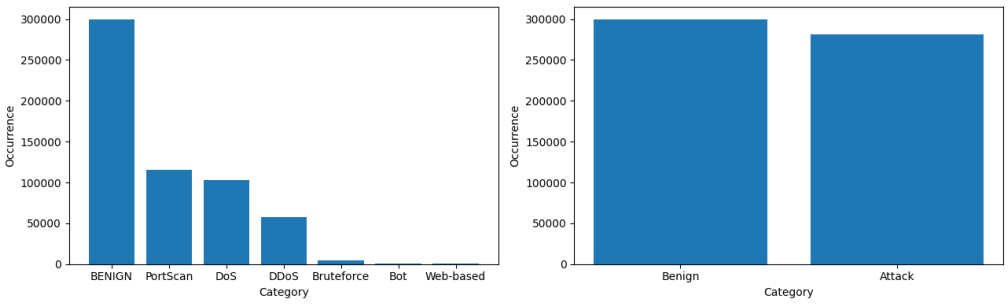
Table 6.1: The categories of the samples in the corrected dataset

The concatenated dataset also contains 5 additional columns, which are Flow ID, Src IP (i.e. source IP address), Dst IP (i.e. destination IP address), Timestamp, and Src Port (i.e. source port). These columns are categorical, thus the conversion to numerical features would be sufficient for the machine learning models. However, they are dropped because the Flow ID and Timestamp does not contain information for classification purposes, Src IP and Dst IP would make the models learn too specific relations (i.e. make it unable to be deployed in other networks) and Src Port is random for some of the attacks executed while the creation of the dataset which would bias the learning process. To follow the procedures in Chapter 4, I dropped **Heartbleed** and **Infiltration** as these attacks do not have enough samples in the dataset. I have tried to drop zero variance features too, but there was none which is already a big difference compared to the original dataset.

I decided to merge *Port_scan* attacks into **PortScan**, this makes the variety of port scans even higher. These port scans represent the same attack type but executed on different parts of the network. I have also combined Patator attacks into **Bruteforce**, Denial of Service attacks into **DoS** and web attacks into **Web-based** category as it was described in Chapter 4. In terms of labeling, I chose to use the numeric notation presented previously in Table 4.3. However, this scenario would result in changes in the numeric notation because some attacks changed place in their occurrence within the dataset. After cleaning the data, I have split it by the ratio of $60 - 20 - 20$ to train, validation and test sets according to the previously described procedure for the original dataset.



(a) before sampling



(b) after sampling

Figure 6.1: Occurrences before and after sampling

For sampling strategy, I chose to manipulate the categories that were also sampled in the original dataset which are the following: **BENIGN**, **Bot** and **Web-based**. First, I sampled down the **BENIGN** samples from 951446 to 345000 which is for reducing the time complexity of the training phase. I oversampled the **Bot** from 438 to 1000 and **Web-based** from 114 to 500. These procedures were made in order to make the attacks and benign samples to around the same level as it can be seen on Figure 6.1.

Feature elimination of the sampled dataset gave interesting results compared to the one done for the original dataset. I have used the same way of evaluating the dataset with an each time smaller feature set on `RandomForestClassifier` (combined with F-test) and it gave the results shown in Figure 6.2. This figure shows that even with having only 10 features, the `RandomForestClassifier` can achieve decent scores. I chose the most optimal threshold for features to keep, which is 35 columns as a small decrease happens after it. The remaining number of features were 32 in the case of the original dataset but with an $\approx 2\%$ worse score.

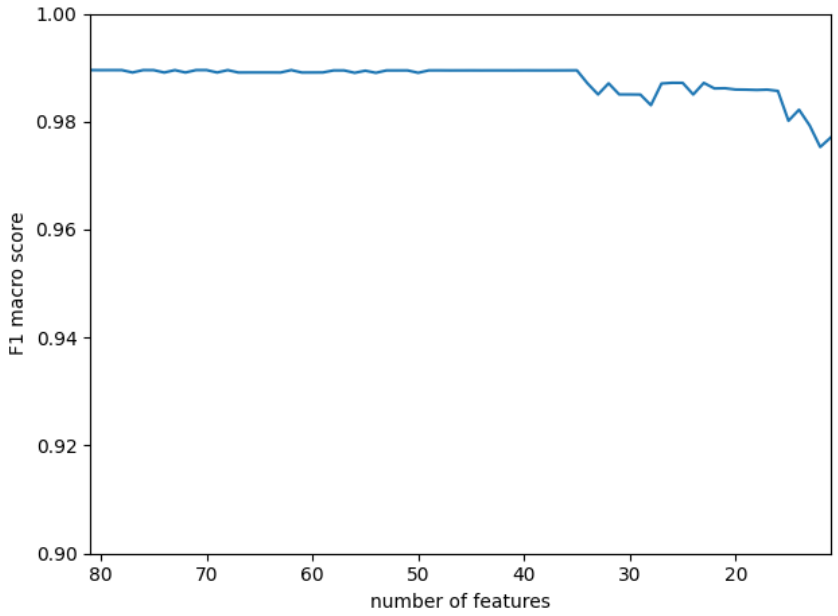


Figure 6.2: F-test iterations

After each preprocessing step, I have evaluated a `RandomForestClassifier`. The classification success can be seen in Figure 6.3. This figure shows that the scores are better than for the original dataset and stay around the same during each process.

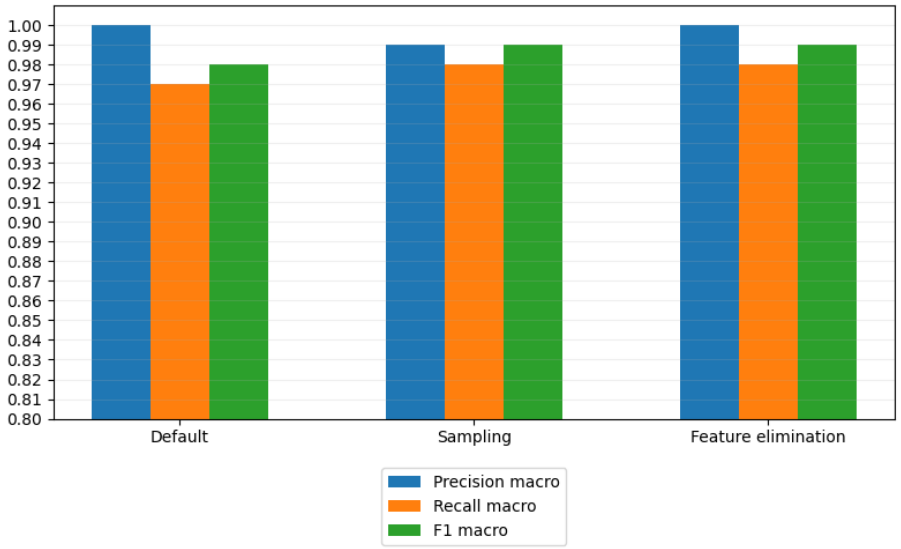


Figure 6.3: Results of preprocessing stages

The main goal of preprocessing in my case is the reduction of run time. The summary of the changes by run times can be seen in Figure 6.4. It shows an $\approx 40\%$ fit time reduction from the start of preprocessing to the end of feature elimination, which is a considerable amount. The validation times stay around the same during these three stages.

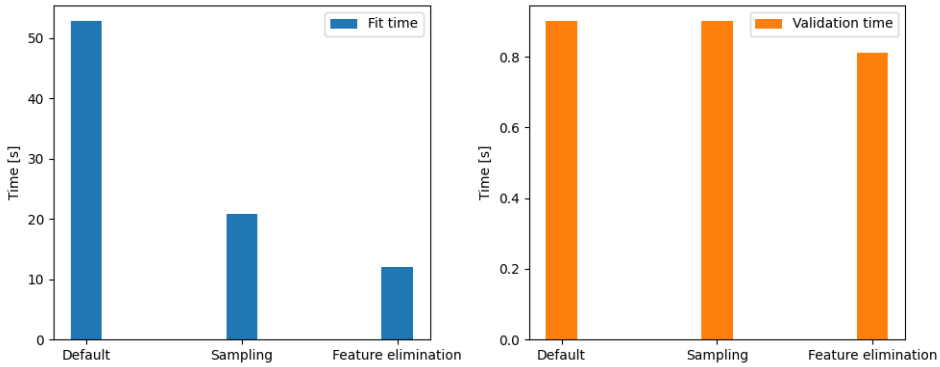


Figure 6.4: Times passed during fitting and validation

6.2 Fine-tuning summary

I used similar parameter ranges for hyperparameter fine-tuning as proposed in Chapter 5. I will not detail the fine-tuning process as the main idea behind it was captured in that chapter. Table 6.2 provides the hyperparameter sets for each model compared to the previous parameter sets of the models tuned on the original CICIDS2017 dataset. It can be stated based on these tables that the corrected dataset requires shallower trees which is good as it makes run time faster. It also means that the data is easier to separate. For the ensemble methods a lower amount of `n_estimators` is needed, which also reduces run time.

Classifier	Parameters
DT	`min_samples_leaf=1`, `min_samples_split=4`, `max_depth=None`
RF	`n_estimators=200`, `max_depth=None`
ABC	`n_estimators=200`, `learning_rate=1.0`
XGB	`n_estimators=30`, `max_depth=11`, `eta=0.75`

$\Downarrow$

Classifier	Parameters
DT	`min_samples_leaf=1`, `min_samples_split=4`, `max_depth=22`
RF	`n_estimators=25`, `max_depth=26`
ABC	`n_estimators=150`, `learning_rate=0.3`
XGB	`n_estimators=20`, `max_depth=10`, `eta=0.5`

Table 6.2: The used parameters for the classifiers – over their base parameter set – detailed in Chapter 5 compared to the novel parameters

The time complexity of fine-tuning is not comparable, because of the deviation between the datasets, i.e. they differ in the amount of training samples within categories and also in the number of features used.

6.3 Evaluation summary

As for the evaluation part, I chose to include only the comparison between models with the same methodology as it was detailed in Section 5.7. This includes, e.g. time complexity analysis, which can be examined in Figure 6.4. Both training and testing times reduced compared to the models used for the original CICIDS2017 dataset. This is because of the previously described model properties, i.e. less `max_depth` for all four models and less `n_estimators` for the three ensemble methods. E.g. `AdaBoostClassifier` with the second approach (see Section 5.5.1.2) uses only $\approx$ 58% of the training time of the `AdaBoostClassifier` benchmarked in Section 5.7. Also, the training time for **RF** decreased by 29.8 seconds and the training time of **XGB:exact** dropped by 23.9 seconds. The testing times also decreased by a lot resulting in the decrease of **RF** test time by $\approx$ 87%.

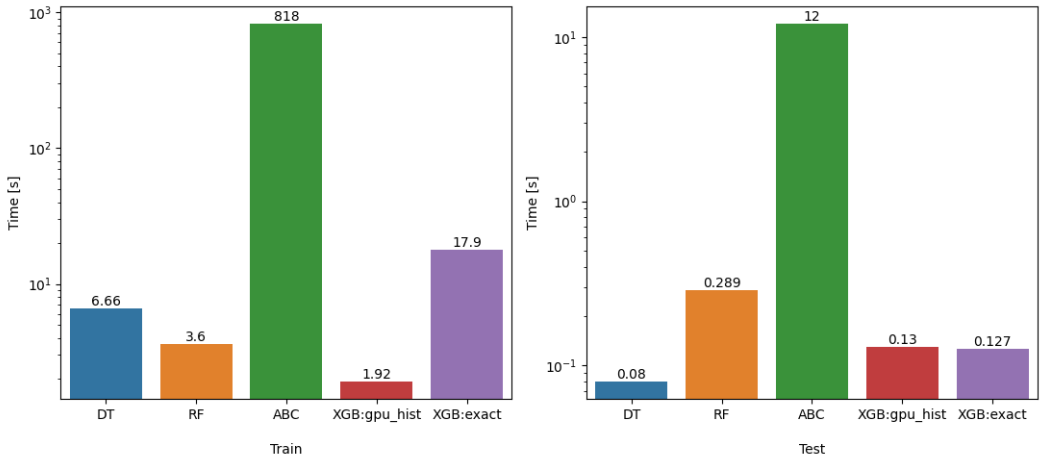


Figure 6.5: The train and test times of the models

Based on the scores shown in Figure 6.6, the ensemble methods all performed better than a single Decision Tree which was not the case with the previous evaluation on the original dataset. As it can be seen, all ensemble methods perform around the same. **ABC** produces the best scores overall and **XGB:gpu_hist** has the lowest scores.

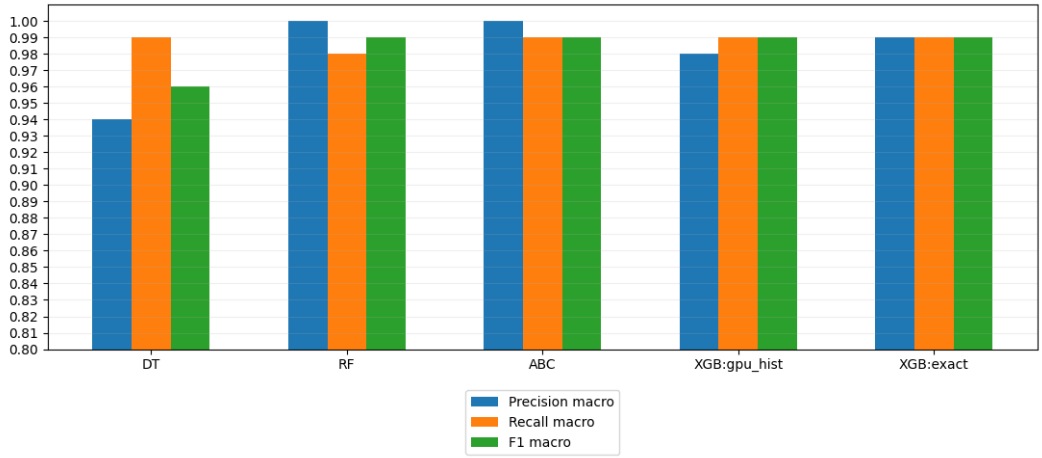


Figure 6.6: The scores achieved on test dataset for each model

Looking at its training and testing time, **ABC** performs well but it has higher time complexity than the others. For a use-case when the training and testing both should be fast then the **XGB:gpu_hist** should be used in my opinion. When both scores and run times are important, **RF** would be an ideal choice.

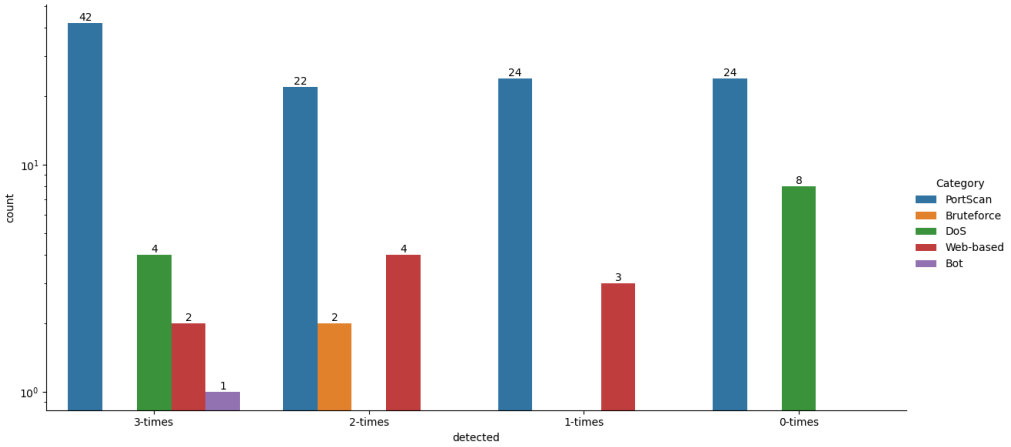


Figure 6.7: The grouping of false negative samples by the times that they could be detected

Figure 6.7 shows the same figure as presented in Section 5.7. The number of $0 - times$ detected attacks are low and narrow down to port scans and Denial of Service attacks. I would like to mention that port scans are not harmful for the systems, i.e. they are just for the reconnaissance of the infrastructure meaning that they have less weight in my eyes when looking at the results. As this figure shows, all **DDoS** attacks could be detected.

Chapter 7

Final thoughts

One of the main conclusions I have is that the corrected dataset should be used for IDS researching purposes. After seeing the difference between the original CICIDS2017 dataset and its corrected version, I can tell that the original gives an invalid picture of the proposed attack types. It was not well engineered because it contains several flaws and its documentation is not detailed enough to get a closer concept of it. However, as I have mentioned before, I have not read anything concerning about the dataset before I started using it. The corrected dataset is much more reliable in giving a proper picture about the presented attack types. However, I would not say that the preprocessing, fine-tuning and evaluation on CICIDS2017 was pointless because at least I could demonstrate the difference the correction makes, and the used methodologies are closely related for these datasets. Also, I could learn from these mistakes and now I know what to look for when I would like to validate the comprehensiveness of a dataset. Another conclusion is that a more complex dataset should have been used in which there are more attacking tools because it could give a more realistic result. It could lead to a more realistic separation of train, validation and test sets where there are zero-days – for the IDS – in the validation and test sets. Also, a more recent dataset would be better to use, however, I read about how promising CICIDS2017 is and I thought that an older, already proven dataset would be a safe way for my thesis. The attacks used within the dataset are mostly outdated by now, however, this aspect does not make my results less valuable as I wanted to demonstrate the anomaly-based attack detection.

In terms of the used models, it can be concluded that ensemble learning methods are usually better at making accurate predictions than a single Decision Tree, however, in some cases the time complexity makes the model not likely to be used. E.g. AdaBoost performs well on the corrected data, however, its run time is much longer than for other methods. Intuitively a single Decision Tree would be the most straightforward to use because it offers a smaller train time, however, as it was shown in Chapter 6 it was not the fastest model. XGBoost is the most flexible solution out of the 4 classifiers presented. It grants great possibilities to tune its parameters and it is also very fast – even more so when GPU-based histograms are used –, while performing well on the dataset. Random Forest is a model which usually performs better than a single Decision Tree while keeping its run time optimal for real life scenarios.

The fine-tuning of the models could be better because the used grid search with cross-validation is sometimes not good enough for my use-case. It is a very resource intensive task as it is an exhaustive search. A questionable substitution would be to use randomized search, which picks random parameters from a range in a predetermined number of itera-

tions. Another alternative would be to use genetic algorithm which tunes the parameters in a way that resembles biological evolution, which could be explored in a future work.

Based on my results, the used anomaly-based IDSs have great potential. The main drawback of IDSs based on machine learning is that they produce a relatively large amount of false positive alarms which would be overwhelming for the human factor in this procedure. These false positive cases can be mitigated with combining anomaly-based solutions hierarchically. Naturally, the time complexity will be multiplied for benign samples as the remaining attacks are dropped in each stage of the classification and the benign samples go through the whole process. A simplistic architecture for this use-case is presented in Figure 7.1.

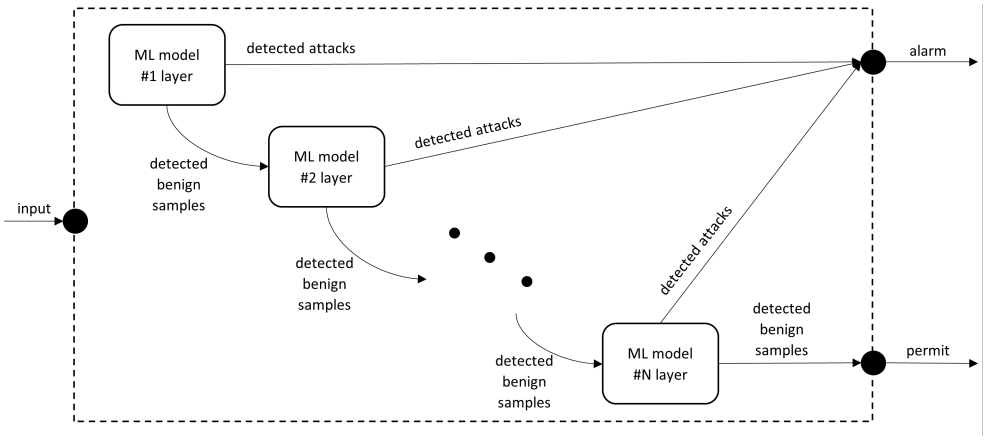


Figure 7.1: An example architecture for a hierarchical IDS

If the goal is to make the predictions as certain as possible, then another option would be to use the previously mentioned hybrid IDS technique (see Figure 7.2). It combines the anomaly-based approach with the signature-based one. The input goes through the signature-based IDS which is, e.g. a database of rules, and if the traffic does match one rule, then it will be dropped as it is an attack based on the predefined rules. After this the remaining data will continue its way through an anomaly-based solution, where the classification happens. From the attacks detected by the anomaly-based approach, it can make new signatures for the database. One of the problems with this method is keeping this database up-to-date but this is possible as the signature-based approaches are most commonly used today. In a future work both of these architectures would be interesting to experiment with.

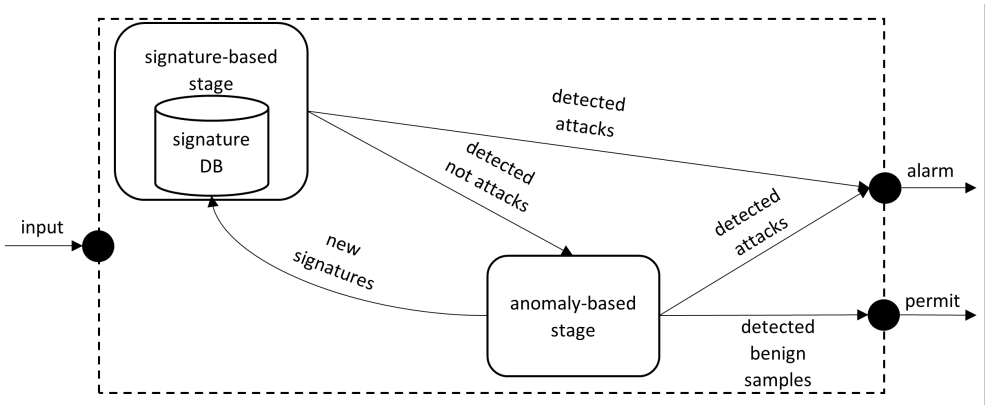


Figure 7.2: An example architecture for a hybrid IDS

Acknowledgements

I would like to thank my advisor, Dr. András Gergely Mészáros, who was always available to help me through the writing of this thesis and also during the work in previous semesters. Our frequent consultations were very progressive and helpful. His willingness to answer my questions even in his free time is greatly appreciated. I am grateful for his good insights and corrections, which helped to improve the overall quality of this work and influenced me to write a demanding paper on this topic.

I would like to thank my girlfriend, Noémi Ujlaki, for all her love and support and for her grammatical corrections in this thesis.

List of Figures

List of Tables

Bibliography

[1] About pandas. https://pandas.pydata.org/about/index.html. Accessed: 2022-12-03.

[2] Ossama B. Al-Khurafi and Mohammad A. Al-Ahmad. Survey of Web Application Vulnerability Attacks. In *2015 4th International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pages 154–158. IEEE, 2015.

[3] An introduction to seaborn. https://seaborn.pydata.org/tutorial/introduction. Accessed: 2022-12-03.

[4] Răzvan Andonie. Hyperparameter optimization in learning systems. *Journal of Membrane Computing*, 1(4):279–291, 2019.

[5] Rebecca Gurley Bace, Peter Mell, et al. Intrusion detection systems. 2001.

[6] Daniel Berrar. Cross-validation., 2019.

[7] Sweta Bhattacharya, Praveen Kumar Reddy Maddikunta, Rajesh Kaluri, Saurabh Singh, Thippa Reddy Gadekallu, Mamoun Alazab, and Usman Tariq. A Novel PCA-Firefly Based XGBoost Classification Model for Intrusion Detection in Networks Using GPU. *Electronics*, 9(2):219, 2020.

[8] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, 2016.

[9] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[10] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[11] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[12] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

[13] Thomas G Dietterich et al. Ensemble learning. *The Handbook of Brain Theory and Neural Networks*, 2(1):110–125, 2002.

[14] DVWA GitHub repository. https://github.com/digininja/DVWA. Accessed: 2022-12-07.

[15] Gints Engelen, Vera Rimmer, and Wouter Joosen. Troubleshooting an Intrusion Detection Dataset: the CICIDS2017 Case Study. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 7–12. IEEE, 2021.

[16] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. A Survey of Botnet and Botnet Detection. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 268–273. IEEE, 2009.

[17] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The Annals of Statistics*, 28(2):337–407, 2000.

[18] Amirhossein Gharib, Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. An Evaluation Framework for Intrusion Detection Dataset. In *2016 International Conference on Information Science and Security (ICISS)*, pages 1–6. IEEE, 2016.

[19] Margherita Grandini, Enrico Bagli, and Giorgio Visani. Metrics for Multi-Class Classification: an Overview. *arXiv preprint arXiv:2008.05756*, 2020.

[20] Oday A. Hassen and H. Ibrahim. Preventive Approach against HULK Attacks in Network Environment. *International Journal of Computing and Business Research*, 7(3), 2017.

[21] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. Multi-class AdaBoost. *Statistics and Its Interface*, 2(3):349–360, 2009.

[22] Heartbleed Official Website. `https://heartbleed.com/`. Accessed: 2022-12-03.

[23] HULK GitHub repository. `https://github.com/grafov/hulk`. Accessed: 2022-12-03.

[24] imbalanced-learn – Documentation. `https://imbalanced-learn.org/stable/index.html`. Accessed: 2022-12-03.

[25] Intrusion Detection Evaluation Dataset (CIC-IDS2017). `https://www.unb.ca/cic/datasets/ids-2017.html`. Accessed: 2022-12-03.

[26] Jupyter Lab – Overview. `https://jupyterlab.readthedocs.io/en/stable/getting_started/overview.html`. Accessed: 2022-12-03.

[27] KDD99 site. `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`. Accessed: 2022-12-07.

[28] Manju Khari, Parikshit Sangwan, et al. Web-application attacks: A survey. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 2187–2191. IEEE, 2016.

[29] Maxime Lanvin, Pierre-François Gimenez, Yufei Han, Frédéric Majorczyk, Ludovic Mé, and Eric Totel. Errors in the CICIDS2017 dataset and the significant differences in detection performances it makes. In *CRiSIS 2022-International Conference on Risks and Security of Internet and Systems*, 2022.

[30] Wenjuan Lian, Guoqing Nie, Bin Jia, Dandan Shi, Qi Fan, and Yongquan Liang. An Intrusion Detection Method Based on Decision Tree-Recursive Feature Elimination in Ensemble Learning. *Mathematical Problems in Engineering*, 2020, 2020.

[31] Matplotlib – Mission Statement. https://matplotlib.org/stable/users/project/mission.html. Accessed: 2022-12-03.

[32] Leland McInnes, John Healy, and James Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[33] Bharti Nagpal, Pratima Sharma, Naresh Chauhan, and Angel Panesar. DDoS tools: Classification, analysis and comparison. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 342–346. IEEE, 2015.

[34] NSL-KDD dataset. https://www.unb.ca/cic/datasets/nsl.html. Accessed: 2022-12-07.

[35] Numpy – What is Numpy? https://numpy.org/doc/stable/user/whatisnumpy.html. Accessed: 2022-12-03.

[36] Shailesh Singh Panwar, Pritam Singh Negi, Lokesh Singh Panwar, and Y. Raiwani. Implementation of Machine Learning Algorithms on CICIDS-2017 Dataset for Intrusion Detection Using WEKA. *International Journal of Recent Technology and Engineering Regular Issue*, 8(3):2195–2207, 2019.

[37] Patator GitHub repository. https://github.com/lanjelot/patator. Accessed: 2022-12-03.

[38] Nerijus Paulauskas and Juozas Auskalnis. Analysis of data pre-processing influence on intrusion detection using NSL-KDD dataset. In *2017 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–5. IEEE, 2017.

[39] Plotly Express – Arguments in Python. https://plotly.com/python/px-arguments/. Accessed: 2022-12-03.

[40] K. Munivara Prasad, A. Rama Mohan Reddy, and K. Venugopal Rao. DoS and DDoS Attacks: Defense, Detection and Traceback Mechanisms - A Survey. *Global Journal of Computer Science and Technology*, 2014.

[41] Ajjarapu Kusuma Priyanka and Siddemsetty Sai Smruthi. WebApplication Vulnerabilities: Exploitation and Prevention. In *2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA)*, pages 729–734. IEEE, 2020.

[42] Joseph Prusa, Taghi M. Khoshgoftaar, David J. Dittman, and Amri Napolitano. Using Random Undersampling to Alleviate Class Imbalance on Tweet Sentiment Data. In *2015 IEEE International Conference on Information Reuse and Integration*, pages 197–202. IEEE, 2015.

[43] Monika Roopak, Gui Yun Tian, and Jonathon Chambers. Deep Learning Models for Cyber Security in IoT Networks. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0452–0457. IEEE, 2019.

[44] Arnaud Rosay, Eloïse Cheval, Florent Carlier, and Pascal Leroux. Network intrusion detection: A comprehensive analysis of cic-ids2017. In *8th International Conference on Information Systems Security and Privacy*, pages 25–36. SCITEPRESS-Science and Technology Publications, 2022.

[45] scikit-learn – AdaBoostClassifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html. Accessed: 2022-12-06.

[46] scikit-learn – Average Precision Score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html. Accessed: 2022-12-03.

[47] scikit-learn – Confusion Matrix. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html. Accessed: 2022-12-04.

[48] scikit-learn – Decision Trees. https://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart. Accessed: 2022-12-05.

[49] scikit-learn – DecisionTreeClassifier. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html. Accessed: 2022-12-06.

[50] scikit-learn – Discrete versus Real AdaBoost. scikit-learn – discrete versus real adaboost. https://scikit-learn.org/1.1/auto_examples/ensemble/plot_adaboost_hastie_10_2.html. Accessed: 2022-12-05.

[51] scikit-learn – F1 score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html. Accessed: 2022-12-04.

[52] scikit-learn – Forests of randomized trees. scikit-learn – forests of randomized trees. https://scikit-learn.org/stable/modules/ensemble.html#forest. Accessed: 2022-12-05.

[53] scikit-learn – GridSearchCV. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. Accessed: 2022-12-07.

[54] scikit-learn – Model Evaluation. https://scikit-learn.org/stable/modules/model_evaluation.html. Accessed: 2022-12-04.

[55] scikit-learn – RandomForestClassifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. Accessed: 2022-12-06.

[56] scikit-learn – StratifiedKFold. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html. Accessed: 2022-12-05.

[57] scikit-learn - Getting Started. https://scikit-learn.org/stable/getting_started.html. Accessed: 2022-12-03.

[58] Iman Sharafaldin, Amirhossein Gharib, Arash Habibi Lashkari, and Ali A. Ghorbani. Towards a Reliable Intrusion Detection Benchmark Dataset. *Software Networking*, 2018(1):177–200, 2018.

[59] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. *ICISSP*, 1:108–116, 2018.

[60] Tanishka Shorey, Deepthi Subbaiah, Ashwin Goyal, Anuraag Sakxena, and Alekha Kumar Mishra. Performance Comparison and Analysis of Slowloris, GoldenEye and Xerxes DDoS Attack Tools. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 318–322. IEEE, 2018.

[61] slowhttptest GitHub repository. https://github.com/shekyan/slowhttptest. Accessed: 2022-12-03.

[62] slowloris GitHub repository. https://github.com/gkbrk/slowloris. Accessed: 2022-12-03.

[63] Joshua Starmer. XGBoost playlist. URL https://youtube.com/playlist?list=PLblh5JKOoLULU0irPgs1SnKO6wqVjKUsQ. Accessed: 2022-12-06.

[64] Lars Sthle, Svante Wold, et al. Analysis of variance (ANOVA). *Chemometrics and Intelligent Laboratory Systems*, 6(4):259–272, 1989.

[65] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A Ghorbani. A detailed analysis of the KDD CUP 99 data set. In *2009 IEEE Symposium on Computational Intelligence in Security and Defense Applications*, pages 1–6. IEEE, 2009.

[66] Alaa Tharwat. Classification assessment methods. *Applied Computing and Informatics*, 2020.

[67] Roman Timofeev. Classification and Regression Trees (CART) Theory and Applications. *Humboldt University, Berlin*, 54, 2004.

[68] UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. https://umap-learn.readthedocs.io/en/latest/index.html. Accessed: 2022-12-07.

[69] XGBoost – Documentation. https://xgboost.readthedocs.io/en/stable/. Accessed: 2022-12-03.

[70] XGBoost – parameters. https://xgboost.readthedocs.io/en/stable/parameter.html. Accessed: 2022-12-06.