

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství
Studijní program: Aplikace informatiky v přírodních vědách



Detekce operačního systému zařízení z dat síťového provozu

Operating System Detection from Network Traffic

DIPLOMOVÁ PRÁCE

Vypracovala: Bc. Anastasiia Kuznetsova
Vedoucí práce: RNDr. Tomáš Jirsík, Ph.D.
Rok: 2023

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství

Akademický rok 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Anastasiia Kuznetsova
Studijní program: Aplikace informatiky v přírodních vědách
Název práce česky: Detekce operačního systému zařízení z dat síťového provozu
Název práce anglicky: Operating System Detection from Network Traffic
Jazyk práce: Angličtina

Pokyny pro vypracování:

1. Nastudujte problematiku monitorování síťového provozu a detekce operačního systému zařízení z dat síťového provozu se zaměřením na detekci pomocí navštívených domén.
2. Na zvolené datové sadě proveďte explorativní analýzu a navrhnete vhodné proměnné pro detekci operačního systému.
3. Nastudujte a zvolte vhodné metody strojového učení pro detekci operačního systému.
4. Proveďte evaluaci modelů pro detekci operačního systému. Zaměřte se na analýzu vlivu zvolených proměnných na přesnost detekce operačního systému.

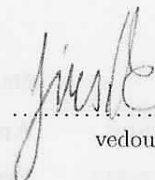


Doporučená literatura:

- [1] LAŠTOVIČKA, M., JIRSÍK T., ČELEDA P., ŠPAČEK S. and FILAKOVSKÝ D. Passive OS Fingerprinting Methods in the Jungle of Wireless Networks. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. Taipei, Taiwan: IEEE Xplore Digital Library, 2018. pp. nestránkováno, ISBN 978-1-5386-3416-5. doi:10.1109/NOMS.2018.8406262.
- [2] LAŠTOVIČKA, M., HUSÁK, M., VELAN, P., JIRSÍK, T. and ČELEDA, P. Passive Operating System Fingerprinting Revisited: Evaluation and Current Challenges. Dostupné z SSRN: <https://ssrn.com/abstract=4292623> or <http://dx.doi.org/10.2139/ssrn.4292623>
- [3] ALLEN, J.M., OS and Application Fingerprinting Techniques, SANS Institute InfoSec Reading Room, 2007. Dostupné z: <https://www.sans.org/white-papers/32923/>.

Jméno a pracoviště vedoucího práce:

RNDr. Tomáš Jirsík, Ph.D.
Cisco Systems (Czech Republic), s.r.o.
Karlovo nám. 2097/10, 120 00 Nové Město, Czech Republic


.....
vedoucí práce

Jména a pracoviště konzultantů:

Mgr. Jan Kohout, Ph.D.
TruU Czech Republic BOT s.r.o.
Praga Studios, Pernerova 697/35, 186 00 Prague 8, Czech Republic


.....
konzultant


Mgr. Dana Majerová, Ph.D.
Katedra softwarového inženýrství,
Fakulta jaderná a fyzikálně inženýrská, ČVUT v Praze


.....
konzultant

Datum zadání diplomové práce: 12. 10. 2022

Termín odevzdání diplomové práce: 3. 5. 2023

Doba platnosti zadání je dva roky od data zadání.


.....
garant oboru


.....
vedoucí katedry




.....
děkan

V Praze dne 12. 10. 2022

Statement of Originality

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during the elaboration of this work are properly cited and listed in complete reference to the due source.

In Prague on 2.05.2023



.....
Bc. Anastasiia Kuznetsova

Acknowledgment

I would like to thank all the people who have played a part in shaping me into the person I am today. I am grateful for the lessons I have learned, the challenges I have overcome, and the experiences that have made me who I am.

First and foremost, I would like to thank my supervisor RNDr. Tomáš Jirsík, Ph.D. for providing me with valuable guidance, support, and encouragement throughout the entire thesis process. His insightful feedback and constructive criticism have been instrumental in shaping the final outcome of this work.

I would also like to give special thanks to my consultant Mgr. Jan Kohout, Ph.D., who led me on my academic journey, who is the author of this topic, and who patiently guided me with his firm hand from my bachelor's thesis through research to my master's thesis.

I would also like to thank Cisco Systems s.r.o. for granting me access to their resources and tools for scientific research. I am also indebted to the consultant Mgr. Dan Majerová, Ph.D. for her invaluable advice and expertise.

I would like to dedicate this work to my beloved grandmother.

Bc. Anastasiia Kuznetsova

Název práce:

Detekce operačního systému zařízení z dat síťového provozu

Autor: Bc. Anastasiia Kuznetsova

Studijní program: Aplikace informatiky v přírodních vědách

Druh práce: Diplomová práce

Vedoucí práce: RNDr. Tomáš Jirsík, Ph.D.
Cisco Systems (Czech Republic), s.r.o.,
Karlovo nám. 2097/10,
120 00 Nové Město, Czech Republic

Konzultant: Mgr. Jan Kohout, Ph.D.
TruU Czech Republic BOT s.r.o.,
Praga Studios, Pernerova 697/35,
186 00 Prague 8, Czech Republic

Konzultant: Mgr. Dana Majerová, Ph.D.
Katedra softwarového inženýrství,
Fakulta jaderná a fyzikálně inženýrská,
České vysoké učení technické v Praze

Abstrakt: Tato práce se zaměřuje na pasivní identifikaci operačních systémů zařízení v síti na základě navštívených domén. Práce bude analyzovat data ze síťového provozu zařízení a na základě analýzy navrhne transformace pozorovaných dat v podobě parametrických filtrů a doporučí vhodné modely strojového učení pro detekci operačního systému. Vybrané parametry a modely budou otestovány a vzájemně porovnány, bude zvolena kombinace parametrů s nejvyšší přesností a bude provedena diskuze nad výsledky experimentů.

Klíčová slova: Detekce operačního systému, strojové učení, klasifikace, hostname, filtrování dat

Title:

Operating System Detection from Network Traffic

Author: Bc. Anastasiia Kuznetsova

Abstract: This thesis focuses on the passive identification of operating systems of devices on a network based on visited hostnames. The work will analyse the device's data from the network traffic and based on the analysis will propose transformations of the observed data in the form of parameter filters and recommend suitable machine-learning models for operating system detection. The selected parameters and models will be tested and compared with each other, the combination of parameters with the highest accuracy will be selected and the results of experiments will be discussed.

Key words: Operation system detection, machine learning, classification, hostname, data filtering

Contents

Introduction	11
1 Data Collection	13
1.1 Network Monitoring	14
1.2 Dataset Description	17
1.2.1 Flows	17
1.2.2 Hostnames	18
1.2.3 User Agents	21
2 Data preparation	25
2.1 Data Cleaning	26
2.2 Labeling	27
2.3 Vectorizers	29
2.3.1 CountVectorizer	31
2.3.2 TF-IDF Vectoriser	32
2.4 Related Works	32
3 Classifiers	35
3.1 Decision Tree	35
3.1.1 Description	35
3.1.2 Algorithm	37
3.1.3 Parameters	38
3.2 Random Forest	40
3.2.1 Ensemble Methods	40
3.2.2 Description	40
3.2.3 Algorithm	41
3.2.4 Parameters	42
3.3 Adaboost	43
3.3.1 Description	43
3.3.2 Algorithm	44
3.3.3 Multi-Class Modifications of the AdaBoost Algorithm	45
3.3.4 Parameters	46
3.4 Evaluation Metrics	46
4 Experiments	49
4.1 Input Parameters Tuning	49
4.1.1 Threshold on the Number of Flows	51

4.1.2	Threshold on the Number of Visited Hostnames	52
4.1.3	Hostname Format	53
4.1.4	Vectorization	55
4.2	Hyper-parameters of the classifiers	56
4.2.1	Decision Tree	57
4.2.2	Random Forest	58
4.2.3	AdaBoost	59
4.3	Discussion	61
Conclusion		63
Bibliography		64
Appendix		67
A Additions to Chapter 1		67
A.1	Top 30 Popular Hostnames. Devices	67
A.2	Top 30 Popular Hostnames. Flows	68
A.3	Ua-Parser Return Examples	69
B Additions to Chapter 2		71
B.1	Number of Flows per User Agent Label	71
B.2	Number of Devices per User Agent Label	72
C Attachment Information		73

Introduction

As computer networks continue to grow in size and complexity, the ability to accurately identify the operating systems (OS) of devices connected to the network is becoming increasingly important for effective network management and security. Passive identification of OS based on visited hostnames is a promising approach for achieving this goal, as it allows network administrators to identify the OS of devices on their network without requiring any active scanning, probing, or physical access to the device.

In recent years, machine learning algorithms have emerged as powerful tools for analyzing complex datasets and making accurate predictions. The currently used methods for the passive OS classification leverage mainly the user agents information, time-to-live, or TCP window size information. However, the distinctiveness of these methods is usually limited to the major OS families. The application of contemporary machine learning algorithms on the available network traffic data could significantly improve the effectiveness of OS detection as we can retrieve a lot more information from the network traffic, such as hostnames.

This thesis aims to explore the feasibility and effectiveness of using these machine learning algorithms for passive OS identification based on visited hostnames, with the goal of developing a practical and reliable method for identifying the OS of devices on a network. We describe the methods of data preprocessing, including data cleaning, data labeling, and feature selection, which are critical for producing accurate and meaningful results. By conducting a thorough analysis of the data generated by this method and comparing the performance of different machine learning algorithms, this thesis will contribute to the development of more effective network management and security strategies, particularly in the context of passive OS identification based on visited hostnames.

The purpose of this work is to analyze the input data set and propose suitable variables and methods for the detection of operating systems. The input data set is a proxy-server logs collected during 24 hours from 9 companies' networks. These data will be analysed both quantitatively and qualitatively to assess the information it contains. Next, a suitable method will be selected to determine the operating system of the devices based on the sites visited.

The first chapter will describe the input data, its sources of origin, and the information it contains. The second chapter will explore the methods of processing this data in order to prepare their transfer to the next stage—the process of training the classifier. The third chapter contains descriptions of the selected classifiers, their

algorithms, and the parameters for their tuning. The last chapter will present the results of experiments with the selected filtering parameters of the input file and the classifier hyper-parameters, the ones that give the most accurate results will be selected, and a discussion of the results will be held.

Chapter 1

Data Collection

The first step in machine learning is to analyse the input data, as this affects the whole model and each step individually. It is important to understand not only the structure and content but also the sources of the information. Understanding the source of information can show which parts of the dataset are reliable and available in sufficient quantity. The input directly affects the choice of model used.

A machine learning life cycle is demonstrated in Figure 1.1. The process begins with the preprocessing of the input data, which consists of error correction, feature selection, a transformation of data formats and their representation, labeling the dataset, balancing sizes of chosen classes, filtering training set elements, and other processes that improve quality of the data and its ability to be successfully used for training a machine-learned model. Some parts of the processing have been divided into separate steps, as we will refer to these steps separately later in the thesis.

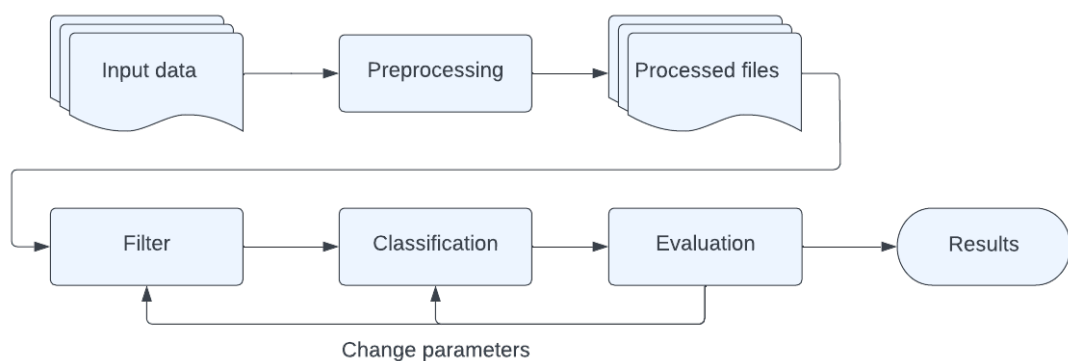


Figure 1.1: Machine learning life cycle

This chapter provides background information on the available data, its quantity, and the quality of the information presented in it. It will also describe in detail the source of information, its specificity, and the data that can be obtained from this source.

1.1 Network Monitoring

A computer network is a set of connections between computing devices(hosts) that allows the transmission and sharing of information. A traditional computer network includes personal computers, servers, and networking hardware. The purpose of a network is to allow computers to communicate with each other and share resources such as storage, computational power, software, sensors, and smart electronics.

The communication between devices can be described by the Open Systems Interconnection(OSI) model. The model partitions data flows into seven abstraction layers of communication between computing systems: Physical, Data Link, Network, Transport, Session, Presentation, and Application. Each layer of the OSI model handles a specific job, data units(PDU), and communicates with the layers above and below itself. In this work, we are explicitly talking only about Network layer and layers above it.

When data is sent over the packet-based network it is broken up into packets. A packet is a unit of data that is transmitted between devices on a network. Packets are routed through a series of interconnected devices, such as routers and switchers, before reaching their final destination. At their destination point packets are collected and reassembled into the original data. The process of breaking down data into packets is called packetisation. This process takes a portion of the user data and prepends the header with control information. The control information typically includes the source and destination addresses and error-checking data. User data is the transferred content, such as text, images, or audio/video files.

A set of rules that specify how to format, send and receive data over a network is called a protocol. Protocols enable different devices and software to communicate with each other, regardless of their differences. Protocols exist for different types of processes and functions on a network: routing, file transfer, error detection, encryption, etc. Some of the most commonly used protocols on the Internet are:

- IP: *Internet Protocol* responsible for routing data packets across networks;
- TCP: *Transmission Control Protocol* ensures reliable and ordered delivery of data;
- UDP: *User Datagram Protocol* provides fast and connectionless delivery of data;
- HTTP: *Hypertext Transfer Protocol* is used to access web pages and resources;
- HTTPS: *HTTP Secure* is an HTTP extension that uses encryption;
- DNS: *Domain Name System* translates domain names into IP addresses;

Different protocols operate at different layers of the OSI model and perform different functions. Network data collection can be performed by different methods and tools, depending on the type, format, and volume of data to be collected. Common methods and tools for data collecting are:

- Packet capture: This method collects all or some of the packets that pass through a network interface or device. Packet capture can provide detailed information about the content and behavior of network traffic, but it can also consume a lot of resources and storage space. Packet capture tools include `tcpdump`¹
- Flow analysis: This method involves collecting and aggregating statistics about network flows [20], which are sequences of packets that share common attributes, such as source and destination addresses, ports, and protocols. Flow analysis can provide an overview of network traffic patterns, volumes, and trends, but it cannot provide the content of packets [9]. An example of the flow protocol would be the NetFlow², and tools such as `nfdump`³ or YAF⁴
- SNMP polling: This method involves querying network devices using the Simple Network Management Protocol [12], which is a standard protocol for managing and monitoring network devices. SNMP polling can provide information about the status, performance, and configuration of network devices, such as routers, switchers, firewalls, and servers. SNMP tools include Cacti⁵
- Log analysis: This method involves collecting and analyzing log files generated by network devices, applications, and services. log files can provide information about the events, errors, and activities that occur on a network. The most popular tool for this method is Splunk⁶.

In our case, we use the flow analysis method for capturing logs as a combination of NetFlow logs and information captured from proxy servers. The captured dataset includes information including passive DNS, timestamps, usernames, and server and client IPs. Most of the pieces of information are almost useless for our purposes since the information in them is filled in a minority of cases and only indirectly relates to this topic. Our goal, among other things, is to develop a universal algorithm, which will not be tied to firm-specific data, so we will also exclude data related to them, such as usernames and server/client IPs.

A network flow is defined as a set of IP packets passing an observation point in the network during a specified time interval. Packets are captured at the observation point, which can be a separate device that does the probe function but can also be collected at internal network elements, such as proxy servers. The device-exporter then aggregates flow records and sends them to the collector. [9]

The flow is defined by its 5-tuple of data points extracted out of the IP header of a packet:

- The source and destination IP addresses;

¹<https://www.tcpdump.org>

²<https://www.cisco.com/c/en/us/products/ios-nx-os-software/netflow-version-9/index.html>

³<https://github.com/phaag/nfdump>

⁴<https://tools.netsa.cert.org/yaf/>

⁵<https://www.cacti.net/>

⁶<https://www.splunk.com/>

- The source and destination ports;
- The protocol

The following information from an IP packet can be added to the flow:

- IP version;
- Length of datagram header;
- Type of Service
- Total Length of the datagram including both header and the data measured octets;
- Identification of the datagram;
- Flags;
- Fragment Offset;
- Time to Live;
- High-level protocol type;
- Header Checksum;
- Source Address;
- Destination Address;
- Options that provide network testing and debugging.

Proxy logs are produced by proxy servers. A proxy server is a server application that acts as an intermediary between a client requesting a resource and the server providing that resource. There are two types of proxy servers: anonymous proxy—this server reveals its identity as a proxy server, but does not disclose the originating IP address of the client; and transparent proxy—this server not only identifies itself as a proxy server but with the support of HTTP header fields, the originating IP address can be retrieved as well. [15]

A proxy server works by intercepting requests from users and forwarding them to the destination server. It can also modify or filter the requests or responses based on certain websites, cache web pages for faster loading, or anonymize users' IP addresses.

Proxy server logs contain the requests made by users and applications on a network. The proxy server logs might be set differently and collect the different information contained in their headers. For this work are relevant the following: date and time, content type, user agent, an authenticated username of the client, target host IP and destination port, target hostname (DNS), and proxy action.

1.2 Dataset Description

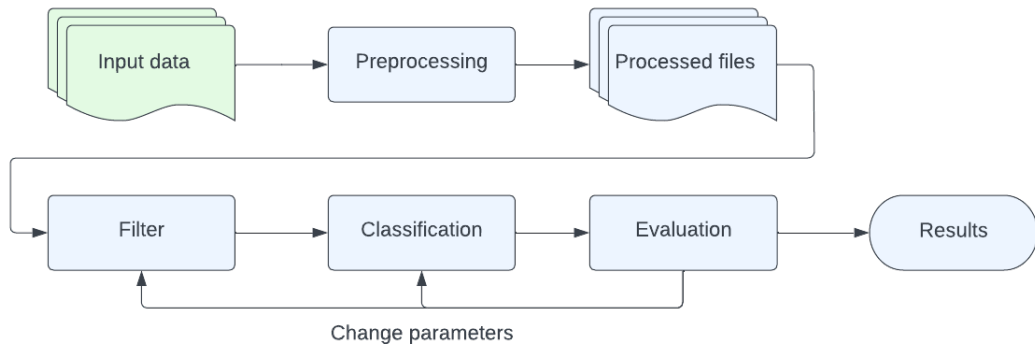


Figure 1.2: Machine learning process flow. Input files

Flow monitoring is a method for monitoring traffic in high-speed networks focused on the analysis of flows, rather than individual packets. Flow monitoring embraces the complete chain of packet observation, flow export using protocols such as Net-Flow and IPFIX, data collection, and data analysis.

The initial data export is transformed to the table in the format as in Table 1.1 contains the information collected on proxy servers during 24 hours from 9 companies' networks. The data export is represented by the table of 5 columns.

The first column called *Company* is an identifier of a company, as there are several companies' telemetry collected from it may be useful to distinguish them one from another. That column is always filled due to the process of data collection.

The second column is called *Device* and this column contains unique device identifiers across all companies. This column is also always filled due to the process of data collection.

The third column is a *Hostname* visited by the device. There are 4 possibilities to fill the row: hostname, IP address, blank field, and random symbols caused by encryption or errors. hostnames will be used as features in OS classification.

The fourth column is for *User Agent* that was used to connect to the hostname from the previous column. User agents will be used for labeling devices according to their operating system. This field also may be empty or filled with random characters due to encryption or errors.

The last column *Occurrences* contains the number of unique 4-tuples from the previous four columns.

1.2.1 Flows

For training the classifier we need data we can rely on. As will be shown below most of the devices are not active enough from a human point of view: many of them have

Company	Device	Hostname	User Agent	Occurrences
Company1	asset_id_1	www.google.com	Mozilla/5.0 (Windows NT 10.0; Win64; x64)	6
Company3	asset_id_2	10.226.111.4	com.apple.WebKit.Networking/16612.2.9.1.30	2
Company2	asset_id_3	some.webpage.com	Mozilla/5.0 (Windows NT 10.0; Win64; x64)	15
Company1	asset_id_4	other.site.io		58

Table 1.1: Example of an input file

	Device	Hostname(Raw IP)	UA	Flows
Filled rows	100%	76%(3%)	39%	100%
Filled flows	100%	90%(9%)	76%	100%
Unique values	207K	907K(73K)	34K	

Table 1.2: Density of the input data

less than 10 flows during the period. However, there is no exact answer or the exact number of how many flows are enough.

Aggregation of a number of flows by device identifier gives us an activity overview. Each device has 1 to over 27 mln flows per day. There are some statistics:

	Number of flows
mean	2.368722e+03
std	9.693423e+04
min	1.000000e+00
25 %	6.700000e+01
50 %	3.890000e+02
75 %	1.148000e+03
max	2.712943e+07

Table 1.3: Flows per device statistics

Combining the information from the picture and the table, we can say that determining the lower limit of the number of flows for each device will not be a simple task: the higher this threshold, the more devices will be filtered out of the dataset. The boundary between the exponential decrease in the number of devices for the lowest values of the threshold, which changes to almost a plateau, can be seen as a pointer to the desired threshold value.

1.2.2 Hostnames

Activity can be also measured by the number of visited hostnames. There are 907310 unique strings in the *Hostname* column. There are 73427(8%) raw IP addresses among them. An empty hostname was seen in 24, 49 % of rows and 10 % of flows.

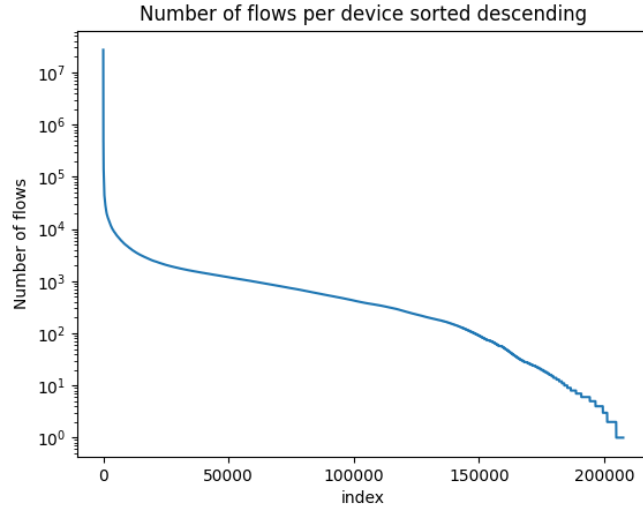


Figure 1.3: Illustration of the number of flows per device in a log-scale

	Number of hostnames
mean	65.771810
std	114.556249
min	1.000000
25 %	5.000000
50 %	29.000000
75 %	86.000000
max	12508.000000

Table 1.4: Unique hostnames per device statistics

Table 1.4 can bring closer matrix sparsity estimation after vectorisation of the list of visited hostnames. Vectorisation assigns a vector to each device, where each coordinate represents a particular hostname. The value of this coordinate is an indicator of whether or not this hostname has been visited by this device in the case of binary representation, or the number of times this hostname has been visited by the device in the case of frequency vector representation. Therefore, based on the values in Table 1.4 and the total number of unique hostnames in the dataset, it can be judged that the matrix will be quite sparse.

Table 1.5 shows the popularity of hostnames, and indicates the popularity of the individual hostnames, specifically how many unique devices visited each one. From the statistic, we can tell that at least 75% of the hostnames were visited only by 1 device. There are 120K hostnames visited by at least 2 devices, which is 13.3% of all hostnames. Thus the length of the vector can be considerably shortened if all hostnames that have been visited by only one device are weeded out of the list.

The popularity of a hostname can also be described by the number of flows. In the second column of Table 1.5 is shown the similar statistics as in the first column, but this time for the number of flows. It can be seen that the statistics are somewhat similar and some correlation can be suspected.

	Number of devices	Number of flows
mean	15.053481	5.421396e+02
std	379.080010	2.024849e+05
min	1.000000	1.000000
25 %	1.000000	1.000000
50 %	1.000000	1.000000
75 %	1.000000	2.0000
max	100035.0000000	1.854528e+08

Table 1.5: Hostname statistics

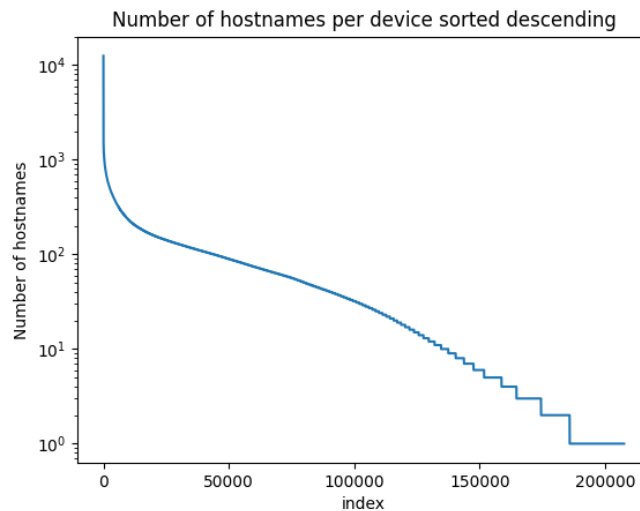


Figure 1.4: Log-scale illustration of the number of hostnames per device ordered descending

Combining those columns by device identifier, we can examine the correlation between the number of devices and the number of flows. For computing the correlation between the number of hostnames per device described in the first column in Table 1.5 and the number of flows per device described in the second column three coefficients are used and their values are as follows:

- Pearson: 0.10267
- Kendall: 0.659788
- Spearman: 0.701258

Pearson's coefficient required to both variables be normally distributed, which is not satisfied in that case.

Kendall coefficient τ is more applicable in this case, as it measures the ordinal association between two measured quantities. There is only one assumption and it is satisfied: variables should be measured on an ordinal or continuous scale. The coefficient modification adds the second condition: a monotonic relationship between

two variables, which is not examined in this case. One of the benefits of this method is its insensitivity to errors. The Kendall coefficient can be calculated by the following formula:

$$\tau = \frac{N_c - N_d}{\frac{1}{2}n(n-1)}, \quad (1.1)$$

where N_c is a number of concordant pairs; N_d is a number of discordant pairs, and n is the number of observations.

The Spearman coefficient requires data to be at least ordinal and the scores on one variable must be monotonically related to the other variable. Calculations based on deviations and much more sensitive to errors and discrepancies in data.

$$\rho = 1 - \frac{\text{cov}(R(X), R(Y))}{\rho_{R(X)}\rho_{R(Y)}}, \quad (1.2)$$

where $\text{cov}(R(X), R(Y))$ is a covariance of the rank variables, and ρ is a standard deviation of the rank variables.

Hostname	Number of flows	Flows, %
g.ceipmsn.com	185452847	37.7
NAN	49085551	9.98
weather.service.msn.com	11366343	2.31
istio-galley.platform.svc	9889967	2.01
212.23.17.73	3960073	0.81
www.msftconnecttest.com	3790441	0.77
cloud-ec-asn.amp.cisco.com	3628237	0.74
ctldl.windowsupdate.com	3224064	0.66
www.google.com	2732376	0.56
maglevserver.maglev-system.svc.cluster.local	2596825	0.53
www.ciscoconnectdna.com	2497382	0.51

Table 1.6: List of the most popular hostnames

Correlation coefficients indicate the presence of a strong correlation between these values, which can be used to create a new parameter for popular hostname indication. A list of the most popular hostnames that cover at least half a percent of the flows or 15% of devices is given in Tables 1.6 and 1.7 respectively. The lists of the 30 most popular hostnames by each parameter can be found in Appendix A.

1.2.3 User Agents

User agents are used for labeling the data. Their presence, variety, and popularity in the data set have a great impact on the results of labeling and therefore classification.

There are 33854 unique user agents in the data set filled 21 mln rows which are 38.65% of the total number of rows. The number of flows that were made with an unknown user agent is 373 mln (75.87%) among the total number of 491 mln analyzed flows.

Hostname	Devices	Devices, %
NAN	100035	48.17
ctldl.windowsupdate.com	81873	39.43
www.msftconnecttest.com	57682	27.78
x1.c.lencr.org	54370	26.18
www.google.com	41567	20.02
ocsp.digicert.com	39987	19.26
edgedl.me.gvt1.com	38866	18.72
update.googleapis.com	36903	17.77
config.edge.skype.com	35572	17.13
login.microsoftonline.com	33115	15.95
arc.msn.com	32798	15.79
18.185.217.177	31457	15.15
ocsp.pki.goog	31455	15.15
18.184.249.36	31449	15.14
18.194.154.159	31425	15.13

Table 1.7: The most popular hostnames by number of devices

Table 1.8 shows the number of unique user agents used by each device. This information will help determine the algorithm for assigning labels to devices. For an even

	Number of unique UAs	Labeled UAs ratio
mean	5.163488	45.175510
std	6.069867	2739.860590
min	1.000000	0.000005
25 %	1.000000	0.146667
50 %	3.000000	0.542299
75 %	7.000000	1.831631
max	893.000000	643162.571429

Table 1.8: Unique UAs per device statistics

better understanding of the challenges of labeling, there is a second column in Table 1.8 that shows how much information for labeling we have. The ratio represents the number of labeled user agents over user agents that we cannot label.

The most popular user agents are used by tens of thousands of devices and cover tens of millions of flows. The following Table 1.9 contains the most popular user agents according to the number of devices they used by and the percentage of flows covered with this user agent.

For labeling, we use user agents parsed with a Python library—*ua-parser* [23]. We chose the library from free and regularly updated libraries, the *ua-parser* was the most popular among them.

The function *Parse* returns information about the device (Listing A.1, Appendix A.3), OS, and user agent. Each user agent is labeled with the OS returned after the

UA	Devices	Flows	Devices, %	Flows, %	Ratio
SeaPort/3.0	52	142312769	0.03	28.93	964.33
NAN	173100	118683118	83.36	24.13	0.29
SeaPort/3.1	20	43139674	0.01	8.77	877.0
Go-http-client/1.1	6483	39419412	3.12	8.01	2.57
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.0.0 Safari/537.36	31306	38570834	15.08	7.84	0.52
Mozilla/4.0 (compatible; ms-office; MSOffice 16)	18410	11730887	8.87	2.38	0.27
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.0.0 Safari/537.36 Edg/105.0.1343.42	38438	11376297	18.51	2.31	0.12
Microsoft BITS/7.8	56229	9525351	27.08	1.94	0.07
Microsoft-CryptoAPI/10.0	105087	8903851	50.61	1.81	0.04
Mozilla/4.0 (compatible; MSIE 6.0; DynGate)	10803	5045458	5.2	1.03	0.2

Table 1.9: User agents with the most flows

parsing by the *ua-parser*. Aggregating user agents by their label will give us results shown in Table 1.10.

Label	Number of UAs
Windows	5893
Android	5438
iOS	4238
Mac OS X	1552
Linux	1052
Ubuntu	365
Debian	157
Red Hat	44
Samsung	39
Chrome OS	16
Fedora	14
CentOS	11
WebOS	10
Tizen	9
FreeBSD	2
Hofer	2
Chromecast	2
NABO	1

Table 1.10: Distribution of user agent labels in given data set

There are 33854 unique user agents and each of them was labeled with *ua-parser*. Therefore we got 19 unique labels including a label for an unknown user agent OS

in the input data set. 15 thousand(44 %) user agents cannot be labeled.

Analysis of unknown user agents, possibilities of missing label assignment, and their impact will be described in Section 2.2.

Chapter 2

Data preparation

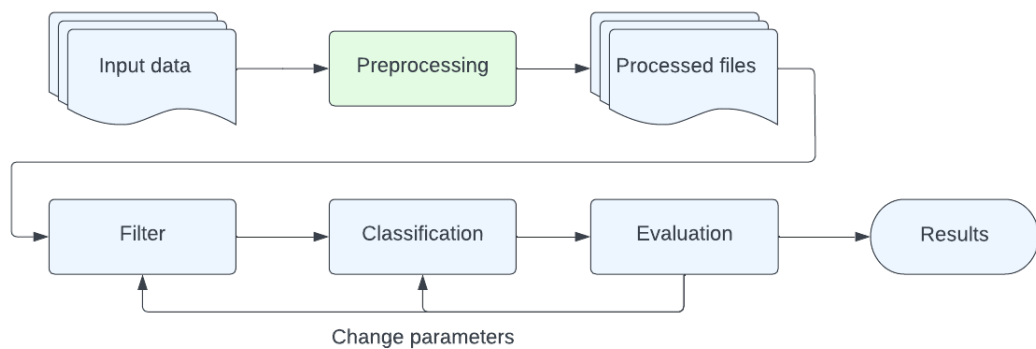


Figure 2.1: Machine learning process flow. Preprocessing.

One of the biggest and most important parts of machine learning is data preparation, which includes feature engineering and data cleaning. The cleaning process involves identifying and correcting or removing errors and inconsistencies, which are very common characteristics for real-world data, in raw data before it can be used in the analysis.

The importance of data cleaning in machine learning cannot be overstated, as it can significantly impact the accuracy and effectiveness of the models developed using that data. Clean data helps to ensure that machine learning models are trained on accurate and relevant data, and can help reduce bias in models.

Through feature engineering, we understand the process of transformation of the input file into an object that can be passed to the next stage of the learning process—classification. The process consists of filtering, labeling, and vectorization. All the before-mentioned steps result in the set of vectors with labels, which form the input of the classification stage.

2.1 Data Cleaning

Errors or missing data in some parts of the dataset affect the classification results much more than in other parts. The more valuable parts of the dataset provide the necessary information without which a model cannot be built. For example, information that is used for device labeling, which is necessary for supervised machine learning. Devices that cannot be labeled can be removed from the dataset because they cannot be used in any part of the process, i.e. they are useless.

The data cleaning process in our case will be slightly different from the classical one, as it is not possible to correct and/or find errors or add missing information, so data filtering methods will be proposed to obtain dataset which will later be used to train the classifier.

Statistics of the dataset presented in Section 1.2 may suggest removing some parts of the dataset or developing filters, conditions, and thresholds to filter other parts. Here are some suggestions for filters that will be tested in the experiments section, studying their effects on classification results and selecting the most successful combinations.

The first parameter to analyze is the number of used user agents. If the list of used user agents is empty for a device, rows with that device can be removed. The minimum number of user agents required to label a device will be discussed in the following section.

The second parameter to analyze is the activity of the device, as devices that have occurred just once in the whole telemetry do not carry any useful information. The following question is what number of occurrences is enough to mark the device as active?

The activity of the device can also be measured by the number of visited hostnames. The more hostnames the device has visited, the denser the feature vector is. A combination of activity parameters will not be used in this thesis.

Filtering can be applied to cell content too. Specific parts of the cell content can be removed, therefore, the number of cells with the same values increases, and more devices can be united into clusters or simply marked as similar devices.

An example of such cells is the *Hostname* column. Each cell has the same format of a hostname as a set of domains divided by dots. The importance of a specific domain or their combination in classification task can be explored in different ways.

The *UA* column can also be analysed in the same way. A user agent is very important for the process of the device's labeling, but the presence of the value in this column does not guarantee, that the device can be labeled. Some user agents may differ slightly, but this may lead to different labels, although in reality, they should have the same label. Machine learning can also be used to label unknown user agents, leading to more labeled devices. As this task in itself is quite non-trivial, it will not be solved within the scope of this work. In further parts of the thesis, some experiments will be done and their results will be discussed.

2.2 Labeling

The process of labeling the data is the most ambiguous step in the whole task. Data labeling is a process of assigning one or more predefined tags or labels to data samples to enable supervised learning algorithms to learn from the data. The labeled data is used to train machine learning models to recognize patterns and make predictions on new, unseen data. The quality and accuracy of the labeled data directly impact the performance of the machine-learning model. Therefore, data labeling is a crucial and time-consuming task that requires careful consideration of the labeling approach, and the quality control measures in place to ensure the consistency and accuracy of the labels.

As we discussed in Subsection 1.2.3, 44.33% of user agents from the data set cannot be labeled because they are not assigned to a specific OS. Moreover, the complexity of the labeling problem is increased with the fact that the known user agent does not guarantee the possibility of the device labeling. Most of the devices used user agents with different labels. In this section, we will use this information to develop a process for labeling devices.

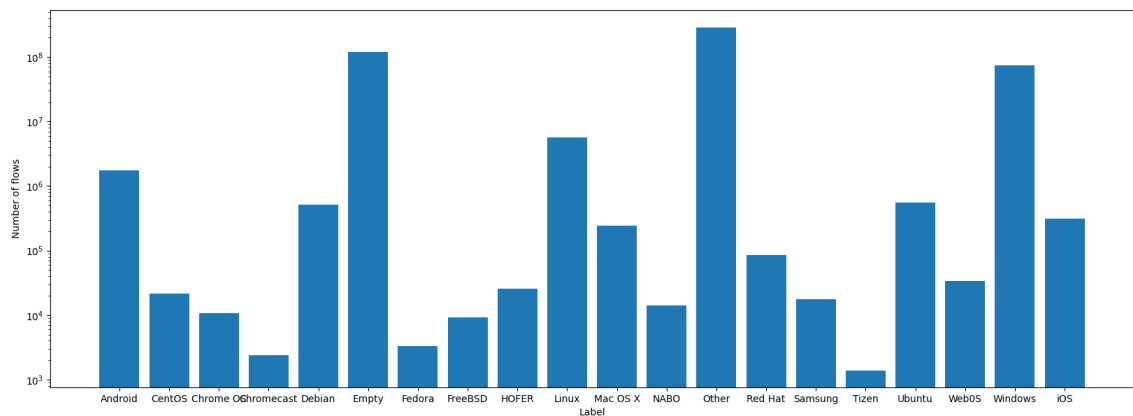


Figure 2.2: Number of flows made with user agents of each label

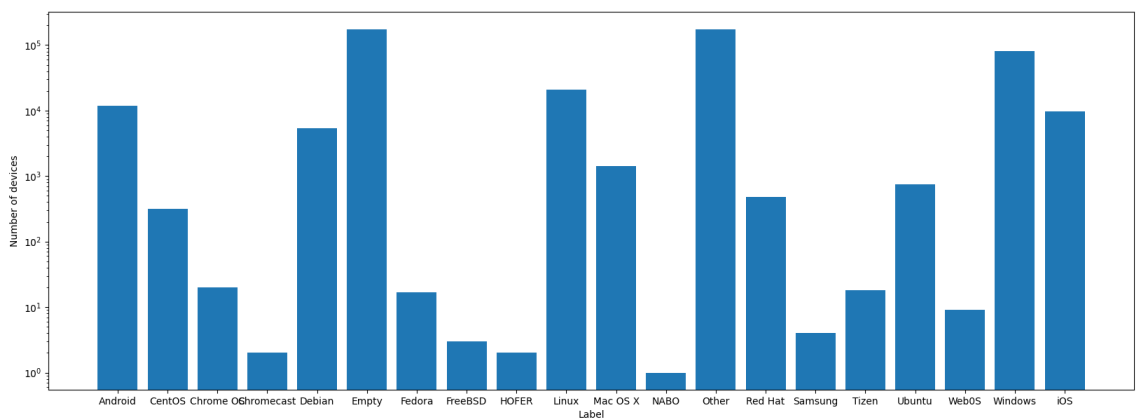


Figure 2.3: Number of devices that used user agent with certain label

First, let’s take a look at how many devices can possibly be labeled. There are 207 thousand unique devices with different amounts of information about used user agents. As can be seen from Table 1.2, the least information we have about is about user agents indeed. The availability of this information does not guarantee the possibility of labeling the device, as was already mentioned in the previous section.

Figure 2.2 shows the number of flows made by user agents with different labels. The number of flows that were made without a captured user agent and the number of flows with user agents with unknown labels are provided in *Empty* and *Other* bars respectively. For better visibility of all labels, the number of flows is on a logarithmic scale. In fact, the number of flows made with unknown user agents is twice as greater as the number of flows with an empty user agent. The exact number of flows per label can be found in Appendix B.1. Figure 2.3 demonstrates the same statistics as Figure 2.2 but per device. The exact numbers can be found in Appendix B.2.

Second, let’s consider specific devices and look at the labeling options. 83.33 % of devices had a flow with an unfilled user agent header. Devices’ labels, that used same-type known user agents and all used user agents had labels, are defined unambiguously. There are only 5735 devices that met these criteria. The number of devices and their labels are in Table 2.1.

Label	Number of devices
Debian	1999
Linux	1892
Android	1283
CentOS	192
Red Hat	159
Mac OS X	113
Ubuntu	91
Fedora	6

Table 2.1: Labels of devices which used user agents with the same label

As will be seen below in Table 2.2 and Table 2.3, those devices cannot be passed to the next stage for training the classifiers by themselves, because they do not represent the data set as the distribution of labels, and their variation is very different from the parameters of the whole data set. Those devices should be diluted with devices with mixed labels of user agents or/and with devices with some unknown or empty user agents.

To select the right ratio of known part and unknown part it is useful to know the impact of the selected value on the data set. The most general criterion is the number of devices that passed the threshold. Figure 2.4 demonstrates the number of devices left after the filtering. As can be seen, there is a rapid drop at the beginning and an almost linear decrease in the rest of the graph. The first threshold on the graph was set to 2% of known user agents and this threshold filtered out more than half of the devices.

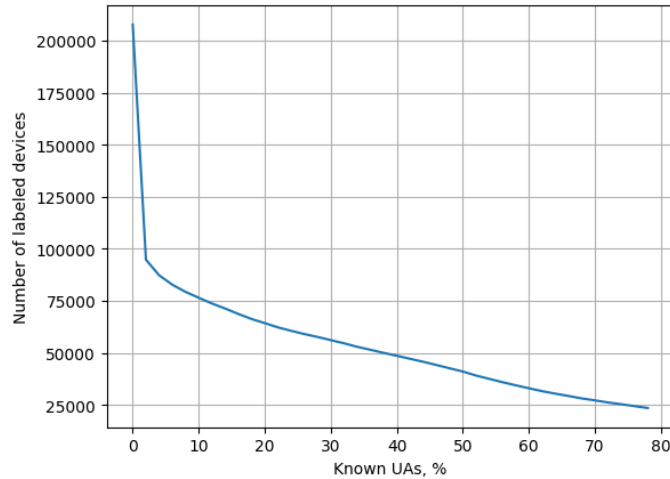


Figure 2.4: Relation between known user agents threshold and the number of devices that can be labeled

In our diploma thesis, the threshold is set to 5% to collect the devices that provide at least some information for their labeling. That threshold allows the labeling of almost 85 thousand devices. To choose the label for a device case of using multiple labels the final label was chosen by majority voting. The distribution of the labeled devices in the data set is provided in Table 2.2.

As can be seen from the lists of used OS, some of them are more common than others, and some of them are expected to have similar behavior. We decided to remove user agents that have very rare labels, such as Tizen, Samsung, FreeBSD, Hofer, Chromecast, and NABO. User agents which have the following labels: CestOS, Debian, Fedora, Red Hat, Ubuntu, and Linux, will share the same label - Linux.

After the transformation histograms in Figures 2.2 and 2.3 will change as in Figure 2.5. Those figures also demonstrate, that in the case of *iOS* labeled user agents a small portion of flows cover a greater portion of devices. The difference between these two values can be important in the process of marking devices. Some methods, such as majority voting, can significantly reduce the number of devices with a given label. Presumably, attaching these labels should be done in a different way.

After applying the same transformation to devices' labels as for user agents' labels described at the beginning of this section, the distribution of the labels was changed as demonstrated in Table 2.3.

2.3 Vectorizers

Input data in machine learning usually comes in various forms such as text, images, audio, and numerical values. In order to apply machine learning algorithms to this data, it must be transformed into a numerical form that can be understood and processed by the algorithms. This process is called vectorization or feature extraction.

In this work, we use hostnames as features. In machine learning, hostnames are

Label	Number of devices
Windows	60355
Linux	11890
Debian	4678
Android	4480
iOS	1757
Ubuntu	478
Red Hat	371
Mac OS X	371
CentOS	262
Chrome OS	13
Fedora	12
WebOS	8
Tizen	8
Samsung	3
FreeBSD	3
Hofer	2
Chromecast	1
NABO	1

Table 2.2: Number of labeled devices

Label	Number of devices
Windows	60355
Linux	17691
Android	4480
iOS	1757
Mac OS X	371

Table 2.3: Devices' labels distribution after transformations

typically represented as a string of characters, which are not directly usable by machine learning algorithms. Therefore, they must be vectorized. Here are some possible approaches to hostnames vectorization:

1. Bag-of-Words: hostnames are tokenized into individual words and a vector is created where each element represents the presence or absence of a particular word in the hostname. This approach can work well if the hostnames contain meaningful words that can help identify patterns or characteristics of interest.
2. Semantic: words are represented as dense vectors of real numbers. This method can capture more complex relationships between words and can be useful if the hostname contains subtle variations or nuances that may not be captured in the Bag-of-Words approach.
3. TF-IDF: This method represents each document as a weighted Bag-of-Words. It assigns weights to each word based on its frequency in the document and its frequency in the corpus.

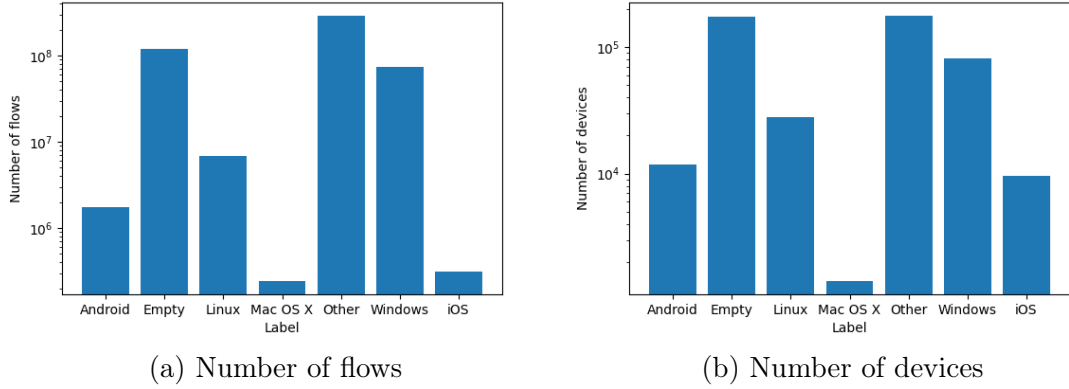


Figure 2.5: The change after merging similar labels and deleting non-popular labels

4. Domain-Specific Features: is called when custom features are created based on domain-specific knowledge or heuristics. This approach is particularly useful when the hostnames have unique characteristics that are not well-captured by more generic feature extraction methods. Those features could include the length of the hostname, use of numbers or special characters, presence of certain keywords, top-level domain, and age of the domain.

The last approach is used mainly in the field of fraud detection for identifying patterns and characteristics that are indicative of fraudulent activity. The others can be potentially used in our case.

The main advantage of the Bag-of-Words method is it is simple to implement and computationally efficient. Disadvantages of the method such as not taking into count the order in the document and not capturing the meaning of the words can lead to poor performance when dealing with complex language tasks, but are totally irrelevant in our case.

TF-IDF method can help to identify words that are unique to a document and therefore more important in describing its content. Considering the number of hostnames that are unique to the devices, there is a contradiction between the efficiency of vectorisation and filtering. On the one hand, we argue that hostnames visited by a single device are not representative and/or useful. On the other hand, this vectorisation method needs these hostnames to be more effective. Thus, this type of vectorisation lengthens feature vectors and shows better results with softer filters on hostnames.

2.3.1 CountVectorizer

CountVectorizer is a feature extraction method in natural language processing that converts a collection of text documents to a matrix of token counts. It is a bag-of-words approach that creates a fixed-length vector representation of a text document, where each element of a vector represents the frequency of a particular word in the document.

CountVectorizer works by first tokenizing the input text into individual words (tokens), then building a vocabulary of all the unique words that appear in the corpus (i.e., the collection of documents). The vocabulary is then used to create a fixed-length vector representation of each document. Each element of the vector corresponds to a unique word in the vocabulary, and the value of the element represents the frequency of that word in the document.

2.3.2 TF-IDF Vectoriser

Term Frequency–Inverse Document Frequency vectorizer is another feature extraction method similar to the CountVectorizer approach, but instead of simply counting the frequency of each word in a document, it weights the importance of each word based on how often it appears in the document and how often it appears in the entire corpus.

TF–IDF vectorizer works by first tokenizing the input text into individual words (tokens), then building a vocabulary of all the unique words that appear in the corpus. For each document in the corpus, the vectorizer then calculates two values for each word in the vocabulary:

1. Term Frequency (TF): The frequency of the word in the document, is calculated as the number of times the word appears in the document divided by the total number of words in the document.
2. Inverse Document Frequency (IDF): The rarity of the word(i) in the corpus, calculated as the logarithm of the total number of documents(n) in the corpus divided by the number of documents(\mathbf{df}_i) that contain the word.

$$\mathbf{idf}_i = \log\left(\frac{n}{\mathbf{df}_i}\right) \quad (2.1)$$

The TF–IDF score as the name suggests is a multiplication of the TF matrix with its IDF:

$$w_{ij} = \mathbf{tf}_{ij} \times \mathbf{idf}_i \quad (2.2)$$

2.4 Related Works

The identification of passive devices in a network is essential for effective network management, monitoring, and security. Various techniques have been proposed and developed for the passive identification of devices in a network, ranging from simple discovery methods to more sophisticated approaches using machine learning and artificial intelligence. This chapter will provide an overview of the related works in the field of passive device identification in a network.

This thesis is partly related to Ben Paterek’s master thesis [19], in which he proposed a method for selecting indicative hostnames for operating system detection. In his

work, 4 classes of operating systems were considered: Android, Linux, Windows, and Apple. The list of semi-manually selected hostnames is very short and efficient, allowing detection based only on the presence of a hostname in the list of hosts visited by the device in question. Detection of operating systems in this case has different accuracy for each type of OS: from 78% for Linux up to 97% for Android. Some parts of his algorithm are done manually, which we wanted to avoid in this work and at the same time increase the accuracy of the detection.

When comparing the different approaches of passive identification of the operating systems in [10], it was found that the highest accuracy was achieved using user agent identification. This method is only applicable to the low portion of the traffic that allows the identification of user agents. For the remaining volume, this method is not applicable due to encryption or incompleteness of the information provided. Moreover, this method is considered unpromising as networks evolve towards encrypted traffic where this information will not be available. On the contrary, detection based on OS-specific domains combined with TCP/IP parameters is considered a promising method with high accuracy and coverage.

A similar approach is described in [11], where machine learning methods are used instead of a ready-made list of domains, and detection is based on a specific combination of TCP/IP packet parameters settings. In this paper, several models have been tested, and not only their technical characteristics as time and memory requirements have been compared but also the results of the classifications. This approach can identify OS in 93.4 % of the sessions with an accuracy of 85.8 %.

In [7] was found that the underlying TCP variant is an important feature for predicting the remote OS. Authors develop a sophisticated tool for OS fingerprinting that first predicts the TCP flavor using passive traffic traces and then uses this prediction as an input feature for another machine learning algorithm for predicting the remote OS from passive measurements. This approach increased the accuracy of the prediction from 84% to 94% on average across all validation scenarios.

Chapter 3

Classifiers

There are many machine learning models, but not all are applicable to all types of problems. Some of them perform markedly better in some tasks and quite the opposite in others. The largest division of models occurs by the type of output given to classifiers and regressors. For our purposes of determining the operating system of a device between several variants, the classifier type is more appropriate, since the expected output is a category rather than a number. Next, our model selection can be bordered on supervised machine learning as our data is labeled.

Given the size of the dataset, which suggests a large number of parameters and a large number of instances, some methods such as Support Vector Machines, K-Nearest Neighbors K-means, and Neural Networks will not be time efficient. We also assume that our classes will overlap a lot and there will not be such an obvious separation between them.

The next big obstacle indicated in Figure 2.5 is a class imbalance, which is a problem for most models.[2] We assume that the most appropriate models for these parameters are Decision Tree, Random Forest, and AdaBoost classifier. This chapter will describe each of them in detail.

3.1 Decision Tree

3.1.1 Description

The Decision Tree (DT) classifier belongs to the family of supervised learning algorithms. It has a hierarchical, tree structure, which consists of a root node, branches, internal nodes, and leaf nodes. A DT is a classifier expressed as a recursive partition of the initial dataset. This algorithm can be used for solving regression and classification problems. Here and further we will talk about classification versions of algorithms only. Important terminology related to DT:

- Root node — a node without incoming edges;
- Decision node — a node with outgoing edges;

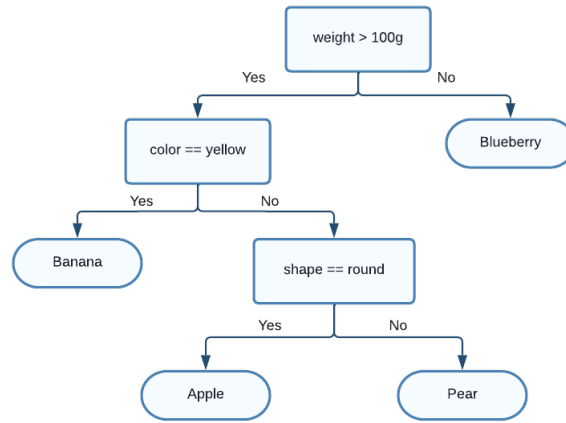


Figure 3.1: Schema of the decision tree classifier

- Terminal node or Leaf — a node without outgoing edges;
- Splitting — a process of making a decision node from the terminal node;
- Pruning — a process of removing decision nodes starting from the leaf node;
- Sub-Tree — a part of a tree, which is also a tree.

Instances are classified by navigating them from the root of the tree down to the leaf under conditions in decision nodes. DT uses multiple algorithms to decide to split a node into two or more sub-nodes. The creation of sub-nodes increases the purity of the node with respect to the target variable. The DT splits the nodes on all available variables and then selects the split which results in the most homogeneous sub-nodes. There are some types of DTs based on splitting tactics [16]:

- ID3 (Iterative Dichotomiser 3)
- C4.5 (successor of ID3)
- CART (Classification and Regression Tree)
- CHAID (Chi-square automatic interaction detection)
- QUEST (The Quick, Unbiased, Efficient Statistical Tree)

The most common algorithm of these is the ID and we will look at its algorithm to give an idea of their structure. Here are steps of the algorithm:

1. The original set \mathcal{S} is the root node;
2. It iterates through every unused feature of the set \mathcal{S} and calculates the entropy $H(\mathcal{S})$ and information gain $IG(\mathcal{S})$ of that feature;
3. Select the feature with the smallest entropy (or the largest information gain) value.

4. Split the set \mathcal{S} by selected feature into subsets;
5. The algorithm continues to recurs on each subset, considering only features never selected before until one of the conditions will match:
 - There are no remaining features;
 - There are no more instances;
 - All instances belong to the same feature value.

All tactics have a similar algorithm and differ in steps 2 and 3 when a node split occurs and a split parameter must be selected. At this point, different tactics use different parameters to define the category of elements. CHAID works on the statistical significance of differences between the parent node and child nodes. CART uses variance for calculating the homogeneity of a node. QUEST uses ANOVA and contingency Chi-square test to select variables for splitting and is shown as significantly faster than others. A comparison of tactics' speeds and their algorithms can be also found in [14].

3.1.2 Algorithm

Given training vectors $x_n \in \mathbb{R}^l$, $n = 1, \dots, N$ and label vector $y \in \mathbb{R}^N$, a decision tree recursively partitions the feature space such that the samples with the same labels or similar target values are grouped together.

Let the data at node m be represented by a subset \mathcal{S}_m with s_m samples, that are considered to be divided into 2 subsets: $\mathcal{S}_m^{\text{left}}(\theta)$ and $\mathcal{S}_m^{\text{right}}(\theta)$. For each candidate split $\theta = (j, t_m)$, where j is a feature and t_m is a threshold, partition the data into $\mathcal{S}_m^{\text{left}}(\theta)$ and $\mathcal{S}_m^{\text{right}}(\theta)$ subsets will be done as follows [4]

$$\begin{aligned}\mathcal{S}_m^{\text{left}}(\theta) &= (x, y) | x_j \leq t_m, \\ \mathcal{S}_m^{\text{right}}(\theta) &= \mathcal{S}_m \setminus \mathcal{S}_m^{\text{left}}(\theta).\end{aligned}\tag{3.1}$$

The quality ($G(\mathcal{S}_m, \theta)$) of a candidate split of node m is then computed using an impurity function or loss function $H()$, the choice of which depends on the task being solved (classification or regression)

$$G(\mathcal{S}_m, \theta) = \frac{s_m^{\text{left}}}{s_m} H(\mathcal{S}_m^{\text{left}}(\theta)) + \frac{s_m^{\text{right}}}{s_m} H(\mathcal{S}_m^{\text{right}}(\theta)).\tag{3.2}$$

Select the parameters that minimize the impurity

$$\theta^* = \underset{\theta}{\operatorname{argmin}} G(\mathcal{S}_m, \theta)\tag{3.3}$$

Then the splitting process recurses for subsets $\mathcal{S}_m^{\text{left}}(\theta^*)$ and $\mathcal{S}_m^{\text{right}}(\theta^*)$ until one of the conditions mentioned above will match. Those conditions in programming are transformed into the following. Reaching the maximum allowable depth that restricts expanding until all leaves are pure or until all leaves contain less than the

minimum number of samples for splitting. Reaching the minimum number of samples for splitting: $s_m < \text{min_samples}$ or $s_m = 1$.

If a target is a classification outcome taking on values $0, 1, \dots, K - 1$, for node m , let

$$p_{mk} = \frac{1}{s_m} \sum_{y \in \mathcal{S}_m} I(y = k) \quad (3.4)$$

be the proportion of class k observations in node m . If m is a terminal node, the predicted probability for this region is set to p_{mk} .

The most common criteria for feature selection measures are Gini, Entropy, and Log Loss. The least are both for the Shannon information gain. These will then be used when running tests using the *scikit-learn* python library [21]:

Gini impurity:

$$H(\mathcal{S}_m) = \sum_k p_{mk}(1 - p_{mk}) \quad (3.5)$$

Log Loss or Entropy:

$$H(\mathcal{S}_m) = - \sum_k p_{mk} \log p_{mk} \quad (3.6)$$

The performance of the decision tree can be optimized by changing mentioned splitting rules or modifying the parameters of the training set. [17] One of those parameters is a class weight that helps with imbalanced data sets. Weights of the classes can be set manually or automatically adjusted inversely proportional to class frequencies in the training set as

$$w_k = \frac{N}{m \cdot \sum_y I(y = k)}, \quad (3.7)$$

where w_k is an adjusted weight of the k -th class; N is a number of samples in the training set; m is a number of classes. A number of features to consider when looking for the best split can be restricted proportionally or to a specific value.

DT has a number of characteristics, which make it more flexible than other classifiers. It can handle various data types, i.e. discrete, continuous values, and categorical values. Additionally, it can also handle values with missing values, which can be problematic for other classifiers. It's also insensitive to underlying relationships between attributes. The Boolean logic and visual representations of DTs make them easier to understand and consume. The hierarchical nature of a DT also makes it easy to see which features are most important, which can help to better understand the input data.

On the other hand, DT can easily over-fit the data which can be solved with pruning. Small changes in data can cause a large change in the structure of the DT causing instability.

3.1.3 Parameters

To get a better idea of the parameters that can be used to modify the way the classifier is trained, and thus influence the quality of the classification, a full list of

parameters available in the *scikit-learn* python library [21] is provided below. At the end of each explanation, in brackets, is the name of the parameter used in the *scikit-learn*.

- The function for measuring the quality of a split. The most popular of them are Gini impurity, Entropy, and Log Loss (`criterion`);
- The condition under which the growth of the tree stops. Nodes can be expanded until all leaves are pure or until all leaves contain less than the allowed number of samples for a split (`max_depth`);
- The minimum number of samples in a leaf for a split (`min_samples_split`);
- The minimum number of samples required to be at a leaf node (`min_samples_leaf`);
- The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node (`min_weight_fraction_leaf`);
- The number of features to consider when looking for the best split (`max_features`);
- Maximum number of leaf nodes (`max_leaf_nodes`);
- Minimum impurity decrease

$$\text{ID} = \frac{N_t}{N \cdot \left(\text{impurity} - \frac{N_{tR}}{N_t \cdot \text{rightImpurity}} - \frac{N_{tL}}{N_t \cdot \text{leftImpurity}} \right)}, \quad (3.8)$$

where N is the total number of samples, N_t is the number of samples at the current node, N_{tL} is the number of samples in the left child, and N_{tR} is a number of samples in the right child. Each number of samples refers to the weighted sum if any was passed as a `hyper-parameter` (`min_impurity_decrease`);

- Weights associated with classes (`class_weight`).
- Minimal Cost-Complexity Pruning parameter α . Used for choosing the subtree to be pruned as a subtree T with the largest cost complexity $R_\alpha(T)$ that is smaller than the given parameter. By default, no pruning is performed. The complexity measure, $R_\alpha(T)$ of a given tree T :

$$R_\alpha(T) = R(T) + \alpha|\tilde{T}|, \quad (3.9)$$

where $|\tilde{T}|$ is the number of terminal nodes in T and $R(T)$ is a total misclassification rate of the terminal nodes (`ccp_alpha`).

3.2 Random Forest

3.2.1 Ensemble Methods

A single model may not perform well individually due to high variance or high bias. However, when weak learners are aggregated, they can form strong learner, as their combination reduces bias or variance, yielding better model performance.

While DTs are common supervised learning algorithms, they may have problems, such as bias and overfitting. However, when multiple DTs form an ensemble in the random forest algorithm, they predict more accurate results.

Ensemble learning methods are made up of a set of classifiers and their predictions are aggregated to identify the most popular result. The most popular ensemble methods are bagging, also known as bootstrap aggregation, and boosting. Those methods combine a set of weak learners into strong learner to minimize training errors.

Bagging is a method when a random sample of data in a training set is selected with replacement, i.e. individual data points can be chosen more than once. After that models are trained independently and the result will be in a form of the average or majority of those predictions. This redistribution of weights helps the algorithm identify the parameters that it needs to focus on to improve its performance. This approach is commonly used to reduce variance within a noisy dataset.

In boosting, a random sample of data is selected, fitted with a model, and then trained sequentially—that is, each model tries to compensate for the weaknesses of its predecessor. With each iteration, the weak rules from each individual classifier are combined to form one, strong prediction rule.

The main difference between those two methods is the way in which they are trained. [18] In bagging weak learners are trained in parallel, in boosting they are learned sequentially. Another difference between bagging and boosting is in how they are used. For example, bagging methods are typically used on weak learners that exhibit high variance and low bias, whereas boosting methods are leveraged when low variance and high bias are observed. While bagging can be used to avoid overfitting, boosting methods can be more prone to overfitting, which can be avoided by removing confusing samples.[24]

3.2.2 Description

Random Forest(RF) is a supervised machine learning tree-based algorithm that is widely used in classification and regression problems. In contrast to the original publication [5], the scikit-learn [6] implementation combines DT classifiers (Section 3.1) by averaging their probabilistic prediction, instead of letting each classifier vote for a single class.

Random forest algorithms have three main hyper-parameters, which need to be set

before training. These include node size, the number of trees, and the number of features sampled. As the random forest consists of DTs it inherits most of DT hyper-parameters that will be mutual for all trees while training. The rest parameters of the classifier describe the structure of the forest, such as the number of trees in the forest.

While the forest construction there is a possibility to use different subsets of the training set in each tree called bootstrap. Without the bootstrap, each tree will use the whole train set for building each tree. Thereby there is the other hyper-parameter that controls the possibility of using the same train sample for subsets for different trees.

Furthermore, when splitting each node during the construction of a single tree, the best split is found either from all input features or a random subset the size of which can be set as an absolute or relative value.

The purpose of these two sources of randomness is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit high variance and tend to overfit. The injected randomness in forests yields decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice, the variance reduction is often significant hence yielding an overall better model.

3.2.3 Algorithm

Given a training set $X = \{\mathbf{x}_i \in \mathbb{R}^N \text{ for } i = 1, \dots, l\}$ with responses $Y = \{y_i \in \mathbb{R} \text{ for } i = 1, \dots, l\}$ bagging repeatedly B times selects a random sample with replacement of the training set and fits trees to these samples:

For $b = 1, \dots, B$:

1. Choose a sample with replacement with k training examples;
2. Train the classification tree h_i

After training, predictions for unseen samples x can be made by taking a majority vote of predictions from all the individual trees on x :

$$H(\mathbf{x}) = \underset{Y}{\operatorname{argmax}} \sum_{i=1}^B I(h_i(\mathbf{x}) = Y), \quad (3.10)$$

where $H(\mathbf{x})$ is combination of classification model, h_i is a single decision tree model, Y is the output variable, and $I()$ is the indicator function. For a given input variable, each tree has a right to vote to select the best classification result. An example of the Random Forest classifier is on Figure 3.2

The combination of bagging and the random selection of features to split allows the RF to better tolerate noise. Classifiers can handle both continuous and categorical

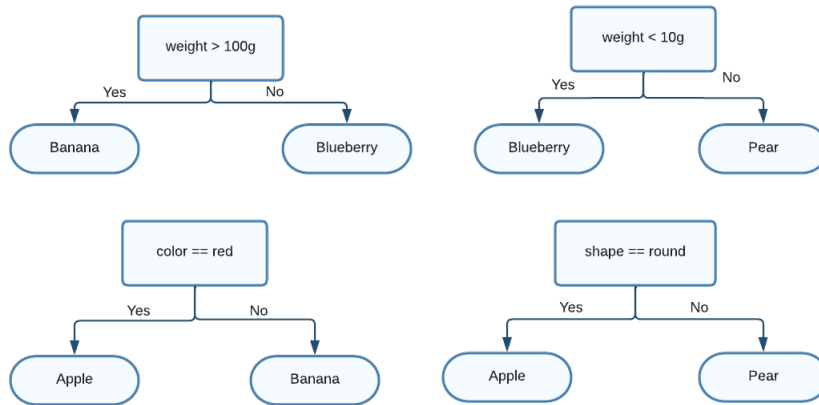


Figure 3.2: Schema of the random forest classifier

variables. The accuracy of RFs is not less than AdaBoost runs faster and does not over-fit the data. [13]

3.2.4 Parameters

As the RF classifier is a combination of multiple DT classifiers, it inherits all of their hyper-parameters but also has additional parameters for an ensemble setup. A more detailed description of inherited parameters can be found in the previous section.

- Inherited parameters:
 - The function to measure the quality of a split (`criterion`)
 - The condition under which the growth of the tree stops (`max_depth`)
 - The minimum number of samples in a leaf for a split (`min_samples_split`)
 - The minimum number of samples required to be at a leaf node (`min_samples_leaf`)
 - The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node (`min_weight_fraction_leaf`)
 - The number of features to consider when looking for the best split (`max_features`)
 - Maximum number of leaf nodes (`max_leaf_nodes`)
 - Minimum impurity decrease (`min_impurity_decrease`)
 - Weights associated with classes (`class_weight`)
 - Minimal Cost-Complexity Pruning parameter (`ccp_alpha`).
- Parameters of the ensemble:
 - The number of trees in the forest (`n_estimators`)

- Possibility of bootstrapping samples. If it is turned off, the whole dataset is used to build each tree (`bootstrap`)
- If the bootstrap is allowed, there is a possibility to use out-of-bag samples to estimate the generalization score (`oob_score`)
- If the bootstrap is allowed, the number of samples to draw from the initial dataset to train each base estimator can be chosen (`max_samples`).

3.3 Adaboost

3.3.1 Description

AdaBoost, which stands for “adaptative boosting algorithm,” is one of the most popular boosting algorithms as it was one of the first of its kind.

The core principle of AdaBoost is to fit a sequence of weak learners on repeatedly modified versions of the data. The results from all of them are then combined through a weighted majority vote to produce the final prediction. The data modifications at each boosting iteration consist of applying weights w_1, w_2, \dots, w_N to each of the training samples, which initially are set to be equal to $1/N$.

At each stage of the algorithm, AdaBoost trains a new classifier using a data set in which the weighting coefficients are adjusted according to the performance of the previously trained classifier to give greater weight to the misclassified data points. Finally, when the desired number of base classifiers have been trained, they are combined to form a committee using coefficients that give different weights to different base classifiers.

The final output of the AdaBoost algorithm is a weighted combination of the weak classifiers that have been trained. The weights assigned to each classifier reflect its performance on the training data, with more weight given to classifiers that perform better.

One of the advantages of AdaBoost is that it can handle high-dimensional data with many features. It is a flexible algorithm that can work with a variety of base classifiers, including DTs, support vector machines, and neural networks. However, AdaBoost can be sensitive to noisy data and outliers, which can negatively impact the performance of the algorithm. AdaBoost can be computationally expensive, especially when using a large number of weak classifiers, and requires a large amount of training data to achieve good performance. The algorithm can be difficult to interpret, as the final output is a weighted combination of multiple weak learners.

The AdaBoost classifier has several hyperparameters that can be tuned to improve its performance on a given problem. Here are the main hyperparameters of the AdaBoost classifier:

- Base estimator: an estimator from which the boosted ensemble is built;

- Number of estimators to use in the ensemble. Increasing the number of estimators can improve accuracy, but can also increase the risk of overfitting;
- Learning rate: weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier;
- Algorithm: SAMME or SAMME.R are two options for the algorithm, that will be demonstrated further.[1]

3.3.2 Algorithm

SAMME is a Stagewise Additive Modeling using a Multi-class Exponential loss function is a multiclass modification of the Adaboost algorithm that is fully described in the paper by Ji Zhu [8]. SAMME.R is a variant of the SAMME as its output is a real-valued number.

In AdaBoost each base classifier $y_m(\mathbf{x})$ is trained on a weights $w_n^{(m)}$ depend on the performance of the previous base classifier $y_{m-1}(\mathbf{x})$. Once all base classifiers have been trained, they are combined into the final classifier $Y_M(\mathbf{x})$.

The original algorithm for binary classification starts with the assumption that we have an input vector of N elements corresponding binary target variables t_1, \dots, t_N where $t_n \in \{-1, 1\}$, M classifiers, $w_n^{(1)}$ is an associated weighting parameter, so the algorithm of the AdaBoost will be as follows[3]:

1. Initialize the data weighting coefficients w_n by setting $w_n^{(1)} = 1/N$ for $n = 1, \dots, N$
2. For $m = 1, \dots, M$:
 - (a) Fit a classifier $y_m(x)$ to the training data by minimizing the weighted error function

$$J_m = \sum_{n=1}^N w_n^{(m)} I(y_m(\mathbf{x}_n) \neq t_n) \quad (3.11)$$

where $I(y_m(\mathbf{x}_n) \neq t_n)$ is an indicator function and equals 1 when $y_m(\mathbf{x}_n) \neq t_n$ and 0 otherwise.

- (b) Evaluate the quantities

$$e^{(m)} = \frac{\sum_{n=1}^N w_n^{(m)} I(y^{(m)}(\mathbf{x}_n) \neq t_n)}{\sum_{n=1}^N w_n^{(m)}} \quad (3.12)$$

and then use these to evaluate

$$\alpha^{(m)} = \log \left\{ \frac{1 - e^{(m)}}{e^{(m)}} \right\}. \quad (3.13)$$

- (c) Update the data weighting coefficients

$$w_n^{(m+1)} = w_n^{(m)} \exp\{\alpha^{(m)} I(y^{(m)}(\mathbf{x}_n) \neq t_n)\} \quad (3.14)$$

3. Make predictions using the final model, which is given by

$$Y^{(M)}(\mathbf{x}) = \text{sign} \left(\sum_{m=1}^M \alpha^{(m)} y^{(m)}(\mathbf{x}) \right). \quad (3.15)$$

3.3.3 Multi-Class Modifications of the AdaBoost Algorithm

The SAMME algorithm for K classes

1. Initialize the data weighting coefficients w_n by setting $w_n^{(1)} = 1/N$ for $n = 1, \dots, N$
2. For $m = 1, \dots, M$:
 - (a) Fit a classifier $y^{(m)}(\mathbf{x})$ to the training set using weights w_n .
 - (b) Evaluate the quantities

$$e^{(m)} = \frac{\sum_{n=1}^N w_n^{(m)} I(y^{(m)}(\mathbf{x}_n) \neq t_n)}{\sum_{n=1}^N w_n^{(m)}}, \quad (3.16)$$

where $I(y^{(m)}(\mathbf{x}_n) \neq t_n)$ is an indicator function and equals 1 when $y^{(m)}(\mathbf{x}_n) \neq t_n$ and 0 otherwise. Then use these to evaluate

$$\alpha^{(m)} = \ln \left\{ \frac{1 - e^{(m)}}{e^{(m)}} \right\} + \log(K - 1). \quad (3.17)$$

- (c) Update the data weighting coefficients

$$w_n^{(m+1)} = w_n^{(m)} \exp\{\alpha^{(m)} I(y^{(m)}(\mathbf{x}_n) \neq t_n)\} \quad (3.18)$$

3. Make predictions using the final model, which is given by

$$Y_M(\mathbf{x}) = \underset{k}{\text{argmax}} \left(\sum_{m=1}^M \alpha^{(m)} \cdot I(y^{(m)}(\mathbf{x}) = k) \right). \quad (3.19)$$

The SAMME.R algorithm does not use α in the algorithm as it gives all models an equal weight of one. The main difference between SAMME and SAMME.R is that SAMME is a returned value. The SAMME returns a discrete value: either 0 or 1. SAMME.R returns probabilities that the sample belongs to a certain class.

1. Initialize the data weighting coefficients w_n by setting $w_n^{(1)} = 1/N$ for $n = 1, \dots, N$
2. For $m = 1, \dots, M$:
 - (a) Fit a classifier $y^{(m)}(\mathbf{x})$ to the training set using weights w_n .

(b) Obtain the weighted class probability estimates

$$p_k^{(m)}(\mathbf{x}) = \text{Prob}_w(c = k|\mathbf{x}), k = 1, \dots, K. \quad (3.20)$$

(c) Set

$$h_k^{(m)}(\mathbf{x}) = (K - 1) \left(\log p_k^{(m)}(\mathbf{x}) - \frac{1}{K} \sum_{k'=1}^K \log p_{k'}^{(m)}(\mathbf{x}) \right), k = 1, \dots, K. \quad (3.21)$$

(d) Set

$$w_n^{(m+1)} = w_n^{(m)} \exp \left(-\frac{K-1}{K} \mathbf{y}_n^\top \log \mathbf{p}^{(m)}(\mathbf{x}_n) \right), n = 1, \dots, N. \quad (3.22)$$

3. Make predictions using the final model, which is given by

$$Y_M(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{m=1}^M h_k^{(m)}(\mathbf{x}). \quad (3.23)$$

3.3.4 Parameters

The list of parameters of the classifier, that can be adjusted:

- The base estimator from which the boosted ensemble is built (`estimator`);
- The maximum number of estimators at which boosting is terminated (`n_estimators`);
- Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier (`learning_rate`);
- The boosting algorithm (`algorithm`): SAMME or SAMME.R;

3.4 Evaluation Metrics

A confusion matrix is a table that is used to evaluate the performance of a supervised machine-learning algorithm by comparing the predicted and actual values of the target variable. In a multi-class classification problem, the confusion matrix is a square $N \times N$ matrix, where N is a number of predicted classes. The rows and columns of the confusion matrix correspond to the actual and predicted class labels, respectively. Each element of the matrix represents the number of instances that were predicted to belong to a particular class and their actual class. The actual class labels forming the rows and the predicted class labels forming the columns

Here is an example of a 3×3 confusion matrix for a multi-class classification problem with three classes. Unlike binary classification, there are no positive or negative classes here in general, but each value can be calculated for each individual class. We will demonstrate the calculations for the *Class 1*. The 4 important terms:

	Predicted Class 1	Predicted Class 2	Predicted Class 3
Actual Class 1	True Positive (TP1)	False Negative (FN2)	False Negative (FN3)
Actual Class 2	False Positive (FP4)	True Negative (TN5)	True Negative (TN6)
Actual Class 3	False Positive (FP7)	True Negative (TN8)	True Negative (TN9)

Table 3.1: Multi-class confusion matrix. Example for 3 classes

- True positive (TP): the number of instances that were correctly classified as *Class 1*. (TP1)
- True negative (TN): the number of instances that were correctly classified as not *Class 1*. (TN5)+(TN6)+(TN8)+(TN9)
- False positive (FP): the number of instances that the model predicts a sample to be a member of *Class 1*, but in reality, the sample belongs to a different class. (FP4)+(FP7)
- False negative (FN): the number of instances that a model fails to predict as a member of *Class 1*, but the sample actually belongs to that class. (FN2)+(FN3)

A confusion matrix gives very simple, yet efficient performance measures for the model. By analyzing the confusion matrix, we can calculate various performance metrics such as precision, recall, and F1 score for each class separately. In multi-class problems, the confusion matrix provides a more detailed and informative evaluation of the performance of the machine learning model compared to a simple accuracy score. It allows us to see with which class *Class 1* was confused more often and to adjust a model accordingly.

Here are some of the most common performance measures that can be used from the confusion matrix. Accuracy gives the overall accuracy of the model considering the exact class, meaning the fraction of the correct predictions.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of samples}} = \frac{TP + TN}{TP + TN + FP + FN} \quad (3.24)$$

Precision tells what fraction of predictions as a positive class were actually positive.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (3.25)$$

Recall tells what fraction of all positive samples were correctly predicted as positive by the classifier.

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3.26)$$

To combine precision and recall values into one score the f1-score is used.

$$f1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN} \quad (3.27)$$

In multi-class classification there are 2 types of f1-score: micro-average and macro-average. The macro-average f1 is an average of the f1-scores of individual classes.

The micro-average is calculated by considering the total TP, total FP, and total FN of the model. It does not consider each class individually, it calculates the metrics globally, so in our example, the micro-average f1-score will be calculated as a sum of values on the main diagonal over two times the sum of values off the main diagonal.

Chapter 4

Experiments

In previous sections, we explored the general information about given data and the amount of information contained in it. We have also broken down the possible variants of variables and their values, which can improve the classification results. Among the variables were not only characteristics of the incoming dataset but also hyperparameters of the classifiers and ways of vectorization. In this section, specific values of the proposed parameters will be tested and the most successful variants will be selected.

The chapter is divided into two parts: the first part is for finding the most efficient filters for input data, and the second part uses the results of the first part and aims to improve the achieved results by selecting the classifier and tuning its hyper-parameters. The Scikit-learn library in Python [22] was chosen for the experiments because it implements a wide range of machine-learning tools, including the necessary machine-learning models and a grid search, which will help find the desired combination of parameters.

4.1 Input Parameters Tuning

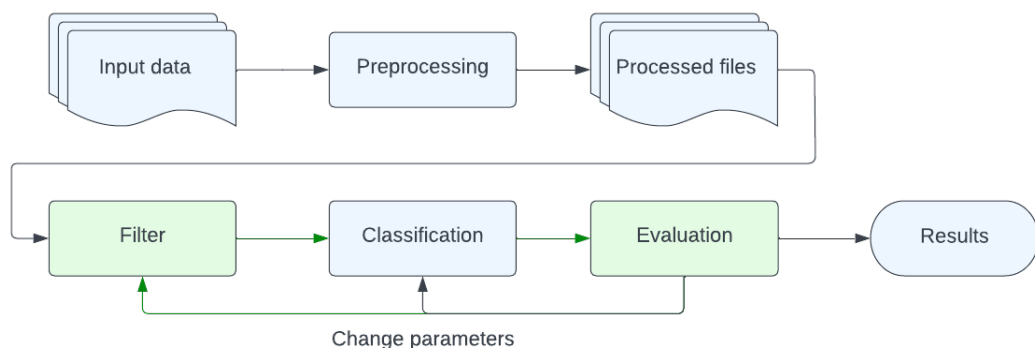


Figure 4.1: Machine learning process flow. Filtering

To train the classifier, we want to use the dataset with the lowest number of errors, with the best quality and most reliable information, reflecting the data for which the trained classifier will be used. The dataset used for training the classifier has to represent the chosen use case for the classifier, in our case, the operating system detection. Which information is relevant for the selected use case is determined by the evaluation metrics, e.g., accuracy or precision, as mentioned in Section 3.4. For our use case, the prediction accuracy is the more important parameter, as the classifier’s prediction on the operating system may be used further in the incident response and misclassification may lead to wrong decisions. Hence, we prefer more precise detection of the OS on fewer devices to less precise detection on many devices.

There are two parameters that can indicate the quantity of information for a given device: the quality of records and their quantity. The quality of records was described in the previous sections and cannot be improved by restoration. The quantity of information about a given device could be measured in two ways: the number of flows or the number of unique hostnames visited by the device, so the first 2 experiments are devoted to investigating these parameters.

For the first and the second experiments, described in the following two subsections, we will fix the values of the other filters and only change the value of the investigated one, and monitor how it affects the results of the experiments. Thus the values of the fixed parameters for the first experiment will be as follows:

- Rows with raw IPs were removed;
- Top-level domains were removed from the hostnames;
- Device was labeled if we were able to label at least 5% of its flows, and the label was chosen by the majority voting;
- Lists of visited hostnames were vectorized with TF-IDF:
 - Hostnames were tokenized as a whole, without division into domains of different levels;
 - Minimal document frequency is set to 2, i.e. hostname must be visited by at least 2 devices to be added as a feature in a feature vector;
- Vectors had a frequency representation;
- Classifier set to the default parameters except for class weights, which will be balanced during the training of the classifier.

For the second experiment, we will fix the threshold on the number of flows at its most successful value and investigate various values of the minimal document frequency.

4.1.1 Threshold on the Number of Flows

The filtering process implies a decrease in the number of devices which can lead to proportion disturbance and therefore affect the quality of the learning process. The confirmed negative effect of an unbalanced dataset [2] can be corrected by inadvertent balancing during the filtering process. To check that this effect does not occur a separate experiment was performed.

The experiment showed that the threshold reduces the number of devices while maintaining the proportions of the labels. However, for some devices that reduction could be more significant than for others due to an imbalanced data set. The exact number of devices with a certain label can be seen in Table 4.1.

	Threshold							
	1	2	10	50	100	150	200	300
Android	4381	4347	4269	3938	3535	3161	2755	1971
Linux	17684	17533	15435	11995	10310	9352	6827	4827
Mac OS X	362	362	321	240	201	177	158	126
Windows	60028	59989	58932	55835	53603	51297	49288	45412
iOS	1757	1755	1724	1539	1410	1296	1196	1035

Table 4.1: Distribution of labels depending on the minimum number of flows requirement

The following Table 4.2 demonstrates the dependency of the accuracy on the required minimum number of flows.

		Threshold							
		1	2	10	50	100	150	200	300
Android	Precision	0.94	0.96	0.95	0.96	0.92	0.95	0.93	0.95
	Recall	0.89	0.9	0.88	0.9	0.89	0.88	0.88	0.93
Linux	Precision	0.97	0.97	0.96	0.96	0.96	0.95	0.95	0.97
	Recall	0.97	0.95	0.96	0.97	0.96	0.97	0.95	0.96
Mac OS X	Precision	0.29	0.36	0.51	0.44	0.88	0.82	0.85	1
	Recall	0.49	0.52	0.48	0.34	0.34	0.3	0.38	0.25
Windows	Precision	1	0.99	1	0.99	1	1	0.99	1
	Recall	1	1	1	1	1	1	1	1
iOS	Precision	0.93	0.79	0.84	0.93	0.93	0.91	0.94	0.94
	Recall	0.94	0.94	0.97	0.95	0.95	0.91	0.94	0.97
T		3m	3m	2m50s	2m19s	1m56s	1m46s	1m30s	1m3s
ND		84K	84K	80K	73K	69K	65K	60K	53K
LV		63.4K	63.8K	64K	63.8K	63.7K	63.5K	63.5K	63.3

Table 4.2: Accuracy of the RF classifier for the different thresholds for a number of flows (*T*: training time, *ND*: number of labeled devices that met the threshold, *LV*: number of hostnames in the training dataset).

As can be seen from the table, accuracy for all labels except for one does not change much. Devices with *Mac OS X* labels show rapid growth in precision and recall values with the raising threshold, which could be explained by a total parity of that

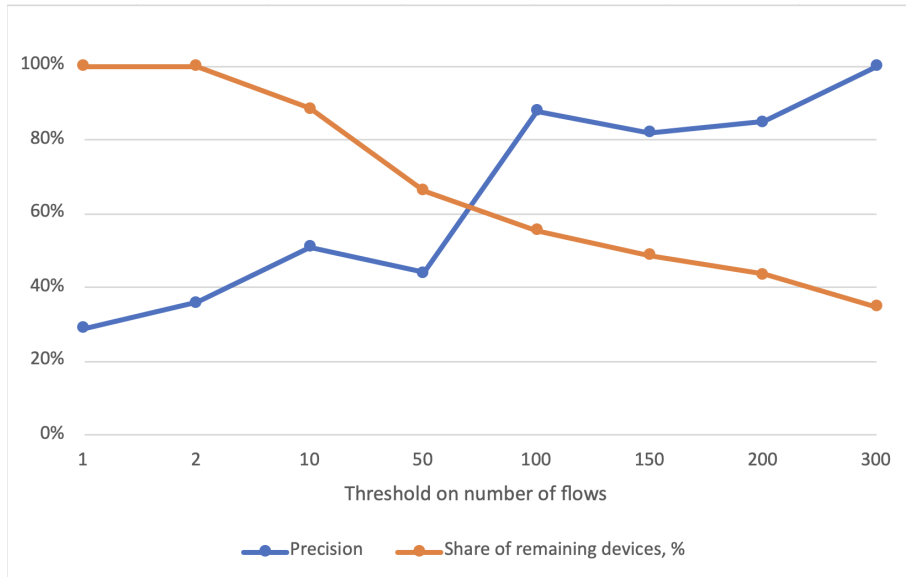


Figure 4.2: Dependency of the RF classifier precision on the minimum number of flows and

devices. With the growing threshold, devices with lower activity are filtered out, so there are fewer devices about which we know the least information, therefore devices with more uncertainty.

Figure 4.2 shows how precision and the number of devices change with the value of the threshold. The Y-axis for each graph on the figure has a different meaning. For the precision graph, the Y-axis means indeed the precision score. For the number of devices, Y-axis means the percentage of devices remaining after the applied threshold.

As can be noticed in the figure the precision score has a significant improvement between 50 and 100 required flows, but as for the number of devices, there is just a little decrease. In further experiments, the value of 100 for this parameter will be considered as optimal.

4.1.2 Threshold on the Number of Visited Hostnames

The other way to mark devices as active is by the number of visited hostnames. General information about the number of hostnames per device in the data set is in Table 1.4. In this section, we will evaluate the impact of that parameter on the accuracy of the RF classifier.

The number of visited hosts is an intuitively understandable parameter, which, however, differs a lot across the given labels, because some of them belong to mobile devices with very different usage styles. Therefore, we do not expect devices with Android label to visit more unique hosts than devices with Linux label. Furthermore, if we look at the distribution of labels depending on a lower threshold for unique labels demonstrated in Table 4.3, we can notice, that for large values of the

threshold some labels are completely filtered out of the data set.

	Threshold								
	1	2	10	20	50	100	150	200	300
Android	4381	4205	3542	2276	211	8	4	2	1
Linux	17684	15922	4696	693	133	44	22	9	3
Mac OS X	362	333	196	101	53	24	9	7	3
Windows	60028	59366	54563	47827	30650	16703	10342	7867	5049
iOS	1757	1753	508	260	34	1	0	0	0

Table 4.3: Distribution of labels depending on the requirement on the minimum number of visited hostnames

The results of the experiments in Table 4.4 showed that this filter is not as efficient as the previous one. There is some increase in accuracy after filtering out devices that had visited one host, but the further increase of the threshold value leads to a decrease in accuracy, a significant reduction of the total number of flows, and a minor decrease in the vectors’ length.

		Threshold					
		1	2	10	20	50	100
Android	Precision	0.94	0.96	0.96	0.96	0.93	x
	Recall	0.89	0.88	0.9	0.98	0.97	x
Linux	Precision	0.97	0.96	0.93	0.93	1	x
	Recall	0.97	0.97	0.94	0.73	0.59	x
Mac OS X	Precision	0.29	0.4	0.73	1	0.6	x
	Recall	0.49	0.49	0.39	0.24	0.3	x
Windows	Precision	1	0.99	0.99	1	1	x
	Recall	1	1	1	1	1	x
iOS	Precision	0.93	0.96	0.84	0.87	1	x
	Recall	0.94	0.94	0.83	0.74	0.64	x
T		3m	2m51s	1m20s	29s	7s	x
ND		84K	81K	63.5K	51K	31K	16K
LV		63.4K	63.7K	63.5K	62.8K	60K	56K

Table 4.4: Accuracy of the RF classifier for the different thresholds for a number of visited hostnames (*T*: training time, *ND*: number of labeled devices that met the threshold, *LV*: number of hostnames in the training dataset)

4.1.3 Hostname Format

Some hostnames can differ by only one single domain; the remaining domains may be the same. Those hostnames may have the same characteristics and be common for some groups of labels and completely atypical for others. For example, *www.bbc.com* and *www.m.bbc.com* lead to the same web, and *joe.wordpress.com* and *marry.wordpress.com* lead to the different hosts with different popularity, but those

pages could be combined into one group "URLs created with WordPress". Those hostnames seemed distinct in the telemetry, but uniting such cases may help to improve the accuracy of the classification and reduce the dictionary size, therefore, the length of the feature vectors. In this section, we will test various modifications of the hostname's format and compare the impact on the results of the classification.

The hostnames consist of domains of various levels. The Top Level domains, located at the end of the hostname, are only a few types so they might not bring additional information. On the other side, the low-level domains are too specific for a given hostname and also might be useless for differentiating OS. Hence we try different combinations of the domains to evaluate the value of the information carried.

Formats of hostnames in each experiment presented in Table 4.5 are as follows:

- I: without modification;
- II: hostnames were separated into individual domains;
- III: Raw IP addresses were removed from the feature vectors;
- IV: Second level domain (SLD) and top-level domain (TLD) were combined together and the rest of the hostname was as it is;
- V: Only SLD and TLD were left in hostnames together, lower-level domains were removed;
- VI: The same as the previous one, but SLD and TLD were separated this time;
- VII: TLDs were removed from hostnames;

		Experiment						
		I	II	III	IV	V	VI	VII
Android	Precision	0.9	0.91	0.9	0.89	0.92	0.93	0.91
	Recall	0.83	0.81	0.81	0.83	0.79	0.79	0.84
Linux	Precision	0.95	0.95	0.95	0.96	0.93	0.94	0.95
	Recall	0.96	0.96	0.97	0.96	0.97	0.97	0.97
Mac OS X	Precision	0.84	0.8	0.8	0.76	0.63	0.64	0.7
	Recall	0.57	0.56	0.51	0.51	0.38	0.46	0.45
Windows	Precision	0.99	0.99	0.99	0.99	0.99	0.99	0.99
	Recall	1	1	1	1	0.99	0.99	1
iOS	Precision	0.91	0.91	0.93	0.96	0.94	0.94	0.93
	Recall	0.95	0.95	0.93	0.95	0.9	0.91	0.94

Table 4.5: Accuracy of the RF classifier for the different formats of hostnames

From experiments, it follows that accuracy in results for each class is varying for different filters. The best accuracy for *Android* is achieved for a shortened version of a hostname, where only TLD and SLD are used and each of them is a feature on its own. However, devices with *Mac OS X* label for the best precision score required

all available information and the precision score is decreasing with every additional limitation.

To summarize the results of the experiments, we conclude, the best 2 options for a hostname’s format are specified in experiment number *III* and number *VII*. The results of combining those 2 transformations can be seen in Table 4.6 in column *III*. Slight degradation in precision for devices with *Linux* label can be observed along with improvement in precision for other devices when compared with the results of each transformation separately.

4.1.4 Vectorization

In this subsection, we present a series of experiments with different text vectorizers described in Section 2.3 to investigate their impact on the performance of a classification task. To compare two classifiers mentioned in Section 2.3 we conducted experiments with a different set of filters and token variations and compared the results of experiments where the only difference was the selected vectorizer.

		Experiment						
		I	II	III	IV	V	VI	VII
Android	CV	0.88	0.91	0.91	0.89	0.88	0.96	0.94
	TF-IDF	0.94	0.95	0.98	0.94	0.93	0.96	0.94
Linux	CV	0.91	0.92	0.92	0.93	0.93	0.89	0.81
	TF-IDF	0.96	0.96	0.96	0.95	0.95	0.92	0.95
Mac OS X	CV	0.67	0.84	0.89	0.78	0.74	0.6	0.88
	TF-IDF	0.81	0.71	0.82	0.85	0.71	0.79	0.71
Windows	CV	0.99	0.99	0.99	1	1	0.99	1
	TF-IDF	0.99	1	1	1	1	1	1
iOS	CV	0.96	0.92	0.95	0.94	0.94	0.85	0.82
	TF-IDF	0.94	0.92	0.94	0.92	0.88	0.97	0.88

Table 4.6: Change in prediction accuracy depending on the selected vectoriser

Experiments, as in the previous section were made with the RF classifier with the default parameters. A second mutual parameter is a minimum number of flows which is set to 50 flows. The rest filters, including parameters of the vectorisers, can be described by the scheme in Table 4.7 with colors in the column of the experiment. The last, blue, row on the scheme shows the format of hostnames used in the experiment. For example, filters in Experiment *V* described in Table 4.7 are as follows:

- Minimum document frequency of a hostname is 50;
- Minimum number of visited hostnames for each device is 5;
- Hostnames that are left in the data set do not include raw IP addresses

Experiment no.	I	II	III	IV	V	VI	VII
Flow Threshold (\geq)	10			50		10	
Visited Domains (\geq)	2			5		10	20
Hostnames	-	No IPs	No IP & No TLD		No IP		

Table 4.7: Scheme of applied filters and transformations for experiments described in Table 4.6

4.2 Hyper-parameters of the classifiers

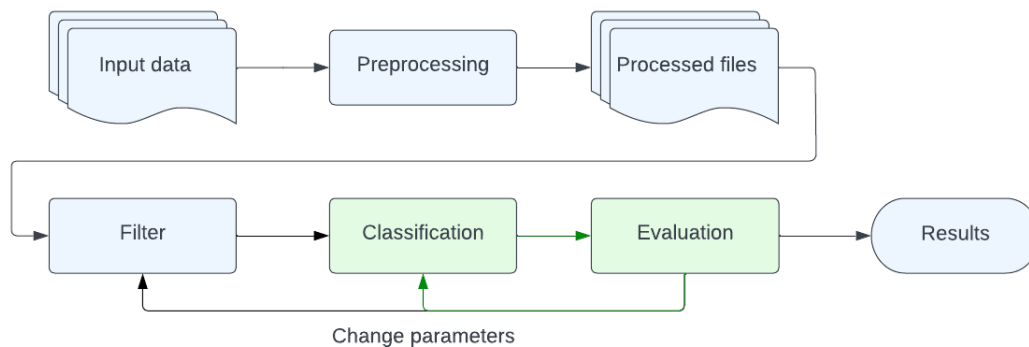


Figure 4.3: Machine learning process flow. Classification

The previous section was dedicated to the filtering and transformation of the input dataset and the representation of feature vectors. In this section, we will discuss the possibilities for improving the parameters of the classifiers and compare their precision. The proposed new parameter values will be tested along with the default parameters that were used in all previous experiments. Thus, we do not exclude the possibility of choosing the default parameters as the best combination, and we will also try out a partial improvement of the default parameters by combining them with new ones.

We used *GridSearchCV*¹ function to try the different combinations of suggested parameters. This function requires the scoring parameter which will numerically evaluate the classification. For multi-class classification, balanced accuracy was chosen as the scoring parameter. This parameter avoids inflated performance estimates on imbalanced datasets.

If y_i is a true value if the i -th sample and w_i is the weight of the sample, then the adjusted weight w_i^a will be:

$$w_i^a = \frac{w_i}{\sum_j I(y_j = y_i)w_j}, \quad (4.1)$$

¹http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

where $I(x)$ is the indicator function. Therefore, for predicted \hat{y}_i for the i -th sample balanced accuracy is defined as:

$$\text{balanced accuracy}(y, \hat{y}, w) = \frac{1}{\sum w_i^a} \sum_i I(\hat{y}_i = y_i) w_i^a. \quad (4.2)$$

For the experiment, we will fix the filters and transformation of the input data on values selected in the previous section, so the only difference in experiments is the classifiers and their parameters. The parameters chosen in the previous sections are:

- The minimum number of flows per device is 100;
- The minimum number of visited hostnames is 1;
- Feature vectors have frequency representation;
- Raw IP addresses and TLDs are removed from the list of hostnames;
- The rest hostnames should be visited by at least 2 devices in the data set.

Most of the parameters influence the fine splitting of the data, which is very important for highly imbalanced data sets with a small number of samples in some classes. Tuning of hyper-parameters can significantly improve the time of training and save computational resources. As can be seen from the previous experiments on classifiers set to default parameters, the time spent on training the classifier is quite small and is not the main problem. The experiments in this chapter are primarily aimed at improving the prediction of underrepresented/overlapped classes.

We chose three classifiers to compare: Decision Tree (DT), Random Forest (RF), and AdaBoost. Detailed information about those classifiers and the description of their hyper-parameters are in Chapter 3.

4.2.1 Decision Tree

	Default	Chosen by <i>GridSearchCV()</i>
<code>criterion</code>	Gini impurity	Gini impurity
<code>max_depth</code>	none	none
<code>min_samples_split</code>	2	2
<code>min_samples_leaf</code>	1	1
<code>min_weight_fraction_leaf</code>	0	0
<code>max_features</code>	none	none
<code>max_leaf_nodes</code>	none	none
<code>min_impurity_decrease</code>	0	0
<code>class_weight</code>	none	Balanced
<code>ccp_alpha</code>	0	0

Table 4.8: Hyper-parameters of the Decision Tree classifier

The depth of the tree is already set to its maximum, as well as the number of samples to split and the number of samples in the leaf.

By trying the different `criterion` values classifier changes the way of measuring the quality of a split, and by testing different weights of the classes the focus of the classifier can be shifted to weaker classes. The suggestion is to compare the imbalanced (the default setting, already tested), balanced, and custom weights, that are shifted to the classes with worse results.

The default balanced weights are calculated in inverse proportion to class frequencies in the training set (3.7). Suggested sets of weights are in Table 4.9.

	Default	Balanced	Set 1	Set2
Android	1	3.95	1	10
Linux	1	1.33	1	3
Mac OS X	1	69.93	100	100
Windows	1	0.26	1	1
iOS	1	9.8	5	30

Table 4.9: Variety of class weights for training the classifier

The output of the `GridSearchCV()` function is an optimal combination of suggested parameters, and in this case, they are, as also shown in Table 4.8: gini criterion and balanced class weights.

The results of the classification for each class with tuned hyper-parameters are presented in Figure 4.4.

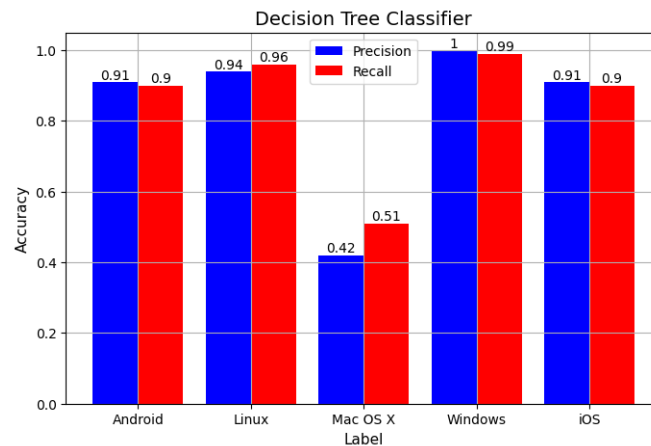


Figure 4.4: Results of the classification with the DT classifier with tuned parameters.

4.2.2 Random Forest

The next tested classifier is a Random forest, which is an ensemble of decision trees, described in Section 3.2. The default parameters of the classifier are in Table 4.10.

	Default	Chosen by <i>GridSearchCV()</i>
<code>criterion</code>	Gini impurity	Gini impurity
<code>max_depth</code>	none	none
<code>min_samples_split</code>	2	2
<code>min_samples_leaf</code>	1	1
<code>min_weight_fraction_leaf</code>	0	0
<code>max_features</code>	none	none
<code>max_leaf_nodes</code>	none	none
<code>min_impurity_decrease</code>	0	0
<code>class_weight</code>	none	Balanced
<code>ccp_alpha</code>	0	0
<code>n_estimators</code>	100	100
<code>bootstrap</code>	False	False
<code>oob_score</code>	False	False
<code>max_samples</code>	None	None

Table 4.10: Hyper-parameters of the Random Forest classifier

In the case of the Random Forest classifier, some of the hyper-parameters which are inherited from the DT classifier, the results of the previous experiments can be used. In that experiment, we will focus on parameters that tune the ensemble setup, which is `n_estimators`, `bootstrap`, `oob_score` and `max_samples`.

Similarly to the previous experiment, we will use the *GridSearchCV()* function, which will test combinations of the following values.

- `bootstrap`: True, False;
- `n_estimators`: 50, 100, 120;
- `oob_score`: True, False;
- `max_samples`: None, 0.5, 0.3;

The following parameters have been chosen by the function as the best: `n_estimators` = 100, `bootstrap` = False, therefore, `oob_score` = False and `max_samples` = None. That set of parameters matches the default setup. The precision score for each class can be found in Figure 4.5.

4.2.3 AdaBoost

The AdaBoost classifier uses boosting algorithm, described in Section 3.3, applied to the decision tree classifiers. Unlike the RF classifier, ensembles of the AdaBoost can be used with different base classifiers. In that experiment was used DT as a base classifier. Default parameters of the classifier:

- The base estimator `estimator` is the DT classifier with a maximum depth equal to 1;

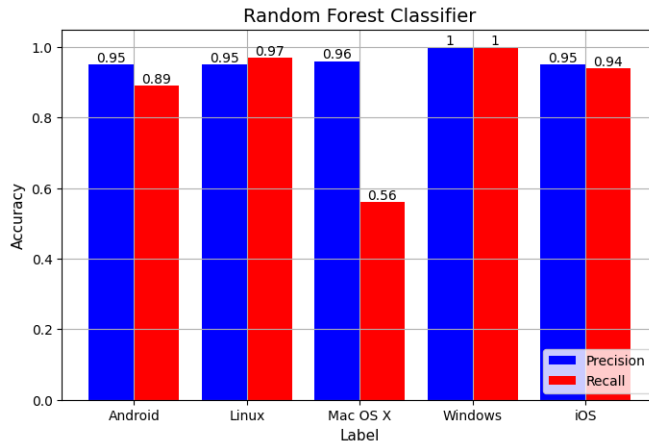


Figure 4.5: Results of the RF classifier with tuned hyper-parameters

- The maximum number of estimators `n_estimators = 50`;
- The learning rate `learning_rate = 1`;
- `algorithm` is SAMME.R;

Tested parameters:

- `n_estimators`: 30, 50, 70;
- `learning_rate`: 0.5, 1, 2;
- `algorithm`: SAMME, SAMME.R;

Parameters that have been chosen by `GridSearchCV()`: `algorithm = SAMME.R`, `learning_rate = 1`, and `n_estimators = 70`. The results of training the classifier with those parameters are in Figure 4.6.

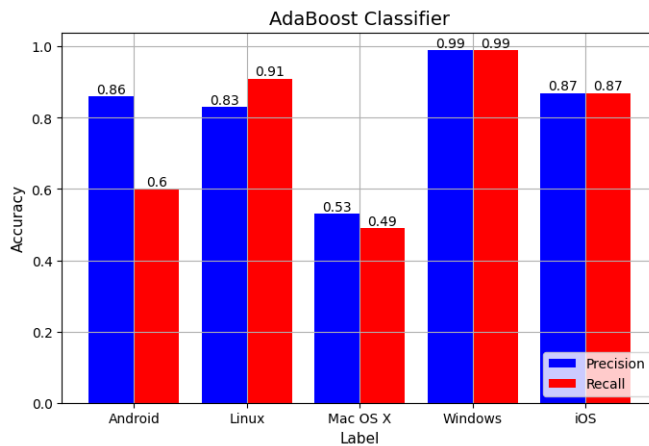


Figure 4.6: Results of the classification with AdaBoost classifier with tuned parameters

The results are as expected worse than the results of the RF classifier because boosting the ensemble of DTs was also tested in experiments on RF hyper-parameters and did not show a better-balanced accuracy score than the ensemble without boosting.

There is a possibility, that changing the base classifier can give better results.

4.3 Discussion

As in any research work, the results of experiments and new explorations of new depths of the topic open up new horizons for future work. Unfortunately, not all of these discoveries fit into this work due to their volume, but we would like to mention them. In this section, we would like to tell about some methods that have not been implemented, as well as some possibilities that have been found during the experiments, and achieved results.

The first such possibility is a different way of labeling devices. The confusion matrix was not fully shown in this thesis, however, its values showed the biggest confusion between the *Mac OS X* and the *Windows* classes. Section 2.2 also described the labeling method and separately pointed out the specifics of the *iOS* devices. We assume, that devices with those two labels can be mislabeled and the labeling approach should differ from other devices or a new labeling algorithm should be developed for all devices. the other possibility is to join those two classes as their behaviour could be similar.

The second improvement can be done by filtering active hostnames by the combination of the number of flows and the number of devices. In Subsection 1.2.2 found a correlation between those values, that can be used for developing of a new criterion for hostnames filtering. Analogically can be investigated and developed filter for devices by combining the number of visited hostnames and the number of flows.

Experiments can be repeated with different ratios of known/unknown user agents and different labeling processes, than was used in this thesis. This could improve the labeling of the devices, and improve the results of the classification. The results demonstrated above were performed on data on the lower border of confidence, where labels were assigned to devices that we know at least something about. Certainly, by raising a confidence border for labeling more and more devices will be filtered out, and the results of the classification can change too.

Experiments on hyper-parameters of classifiers show that the default parameters are a pretty good estimation. For the Decision Tree and Random Forest, only one hyper-parameter was improved.

In comparison with other classifiers, the AdaBoost classifier has the worst results, which could be improved with the different base classifiers.

Conclusion

In this thesis, a method was developed to apply machine learning to detect the operating system of a device on a network based on its behaviour, namely based on visited hostnames. The method was developed using data collected from network devices from 9 different networks over a 24-hour period. The collected data set was in a form of a table with information from the headers of the TCP packets. From the table were used: device ID, hostname, user agent, and a number of flows.

The resulting table was analysed for completeness and diversity of data and, based on this analysis, methods were proposed to help identify the most useful and complete information for training the operating system classifier. The results of the analysis showed that the information needed to label devices were available for 76% of the traffic, but only 44% of devices were labeled using the most permissive labeling method. There was also a strong imbalance of classes, where the most underrepresented class was the device class with the *Mac OS X* label, which was partly explained by the labeling method: the user agents with this label were dominant only in a small proportion of cases.

The greatest improvement of the classifier was achieved by introducing the threshold on a minimum number of flows, which increased the accuracy of the prediction for the most difficult class of devices, with *Mac OS X* label, by 44%. On the contrary, using the threshold for the number of visited hostnames proved to be ineffective. When comparing vectorisation methods used for processing the hostnames, the TF-IDF showed distinctly better results than the Count Vectorizer. Similarly, experiments on feature representation showed a strong correlation between its shape and the classification results. The most successful experiment was the one in which hostnames were without top-level domains and there were no raw IPs among them.

In the final part, experiments were carried out to tune the parameters of the classifiers, which revealed that the default settings of hyper-parameters were quite close to the most optimal ones and were only slightly adjusted. The Random Forest classifier with balanced class weights had the best performance: achieved precision for Android devices is 95%, Linux — 95%, Mac OS X — 96%, Windows — ~100%, and iOS — 95%.

The resulting algorithm has achieved, by a combination of simple transformations and filters, a high level of accuracy in the device operating system prediction. The other advantage of the algorithm lies in its versatility and not being tied to a specific company, region, or period of time. The method proved to be fast, simple, and easily adjustable if necessary.

Bibliography

- [1] *AdaBoost classifier documentation*. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html> (visited on 08/24/2022).
- [2] Haseeb Ali et al. “Imbalance class problems in data mining: A review.” In: *Indonesian Journal of Electrical Engineering and Computer Science* 14.3 (2019), pp. 1560–1571.
- [3] C.M. Bishop. *Pattern Recognition and Machine Learning*. New York: Springer-Verlag, 2006.
- [4] Leo Breiman. *Classification and regression trees*. Routledge, 2017.
- [5] Leo Breiman. “Random forests.” In: *Machine learning* 45 (2001), pp. 5–32.
- [6] *Decision tree classifier documentation*. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html> (visited on 08/24/2022).
- [7] Desta Haileselassie Hagos et al. “Advanced Passive Operating System Fingerprinting Using Machine Learning and Deep Learning.” In: *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. 2020, pp. 1–11. DOI: 10.1109/ICCCN49398.2020.9209694.
- [8] Trevor Hastie et al. “Multi-class adaboost.” In: *Statistics and its Interface* 2.3 (2009), pp. 349–360.
- [9] Rick Hofstede et al. “Flow monitoring explained: From packet capture to data analysis with netflow and ipfix.” In: *IEEE Communications Surveys & Tutorials* 16.4 (2014), pp. 2037–2064. DOI: 10.1109/COMST.2014.2321898.
- [10] Martin Lastovicka et al. “Passive os fingerprinting methods in the jungle of wireless networks.” In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. 2018, pp. 1–9. DOI: 10.1109/NOMS.2018.8406262.
- [11] Martin Laštovička, Antonín Dufka, and Jana Komárková. “Machine learning fingerprinting methods in cyber security domain: Which one to use?” In: *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*. IEEE. 2018, pp. 542–547.
- [12] D Levi, P Meyer, and B Stewart. *Simple network management protocol (SNMP) applications*. Tech. rep. 2002.

- [13] Yanli Liu, Yourong Wang, and Jian Zhang. “New machine learning algorithm: Random forest.” In: *Information Computing and Applications: Third International Conference, ICICA 2012, Chengde, China, September 14-16, 2012. Proceedings 3*. Springer. 2012, pp. 246–252.
- [14] Wei-Yin Loh and Yu-Shan Shih. “Split selection methods for classification trees.” In: *Statistica sinica* (1997), pp. 815–840.
- [15] Ari Luotonen and Kevin Altis. “World-wide web proxies.” In: *Computer Networks and ISDN systems* 27.2 (1994), pp. 147–154.
- [16] Oded Z Maimon and Lior Rokach. *Data mining with decision trees: theory and applications*. Vol. 81. World scientific, 2014.
- [17] Rafael G. Mantovani et al. “Hyper-Parameter Tuning of a Decision Tree Induction Algorithm.” In: *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*. 2016, pp. 37–42. DOI: 10.1109/BRACIS.2016.018.
- [18] David Opitz and Richard Maclin. “Popular ensemble methods: An empirical study.” In: *Journal of artificial intelligence research* 11 (1999), pp. 169–198.
- [19] Benjamin Paterek. “Behavioral identification of operating systems.” Master’s Thesis. České vysoké učení technické v Praze, 2021.
- [20] Vern Paxson et al. “An architecture for large scale Internet measurement.” In: *IEEE Communications Magazine* 36.8 (1998), pp. 48–54.
- [21] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python.” In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.
- [22] Sebastian Raschka and Vahid Mirjalili. *Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2*. Packt Publishing Ltd, 2019.
- [23] Matt Robenolt. *Ua-parser*. URL: <https://github.com/ua-parser/uap-python> (visited on 07/29/2020).
- [24] Alexander Vezhnevets and Olga Barinova. “Avoiding boosting overfitting by removing confusing samples.” In: *Machine Learning: ECML 2007: 18th European Conference on Machine Learning, Warsaw, Poland, September 17-21, 2007. Proceedings 18*. Springer. 2007, pp. 430–441.

Appendix A

Additions to Chapter 1

A.1 Top 30 Popular Hostnames. Devices

Hostname	Devices	Devices, %
NAN	100035	48.17
ctldl.windowsupdate.com	81873	39.43
www.msftconnecttest.com	57682	27.78
x1.c.lencr.org	54370	26.18
www.google.com	41567	20.02
ocsp.digicert.com	39987	19.26
edgedl.me.gvt1.com	38866	18.72
update.googleapis.com	36903	17.77
config.edge.skype.com	35572	17.13
login.microsoftonline.com	33115	15.95
arc.msn.com	32798	15.79
18.185.217.177	31457	15.15
ocsp.pki.goog	31455	15.15
18.184.249.36	31449	15.14
18.194.154.159	31425	15.13
ipv6.msftconnecttest.com	31121	14.99
login.live.com	29872	14.39
13.107.21.239	28852	13.89
204.79.197.239	28852	13.89
3.120.91.16	28620	13.78
18.196.194.92	28589	13.77
3.121.5.209	28510	13.73
40.127.240.158	28487	13.72
api.msn.com	28435	13.69
safebrowsing.googleapis.com	28336	13.65
51.104.136.2	28275	13.62
smartscreen-prod.microsoft.com	27835	13.4
assets.msn.com	27748	13.36
www.bing.com	27588	13.29
51.124.78.146	27210	13.1

Table A.1: The most popular hostnames by number of devices

A.2 Top 30 Popular Hostnames. Flows

Hostname	Number of flows	Flows, %
g.ceipmsn.com	185452847	37.7
NAN	49085551	9.98
weather.service.msn.com	11366343	2.31
istio-galley.platform.svc	9889967	2.01
212.23.17.73	3960073	0.81
www.msftconnecttest.com	3790441	0.77
cloud-ec-asn.amp.cisco.com	3628237	0.74
ctldl.windowsupdate.com	3224064	0.66
www.google.com	2732376	0.56
maglevserver.maglev-system.svc.cluster.local	2596825	0.53
www.ciscoconnectdna.com	2497382	0.51
staging.ciscoconnectdna.com	2278257	0.46
officecdn.microsoft.com	2235599	0.45
detectportal.firefox.com	2130231	0.43
connectdna.cisco.com	2098499	0.43
monitoring.googleapis.com	2057820	0.42
login.microsoftonline.com	2041318	0.41
136.117.74.62	1884563	0.38
dl.delivery.mp.microsoft.com	1856126	0.38
cndp-20211202-monitor.dev.cndp-cloud-proxy.com	1811161	0.37
tetra-defs.amp.cisco.com	1698475	0.35
tlu.dl.delivery.mp.microsoft.com	1659932	0.34
cndp-20211202-configbe.dev.cndp-cloud-proxy.com	1502701	0.31
edgedl.me.gvt1.com	1481729	0.3
staging-connectdna.cisco.com	1345288	0.27
maglev.maglevcloud3.tesseractinternal.com	1290580	0.26
assets.msn.com	1283379	0.26
outlook.office365.com	1186002	0.24
download.windowsupdate.com	1152534	0.23
self.events.data.microsoft.com	1145690	0.23

Table A.2: List of the most popular hostnames by the number of flows

A.3 Ua-Parser Return Examples

```
from ua_parser import user_agent_parser
import pprint
pp = pprint.PrettyPrinter(indent=4)
ua_string = 'Mozilla/5.0_(Macintosh;_Intel_\
Mac_OS_X_10_9_4)_AppleWebKit/537.36_(KHTML,_\
like_Gecko)_Chrome/41.0.2272.104_Safari/537.36'
parsed_string = user_agent_parser.Parse(ua_string)
pp.pprint(parsed_string)
{  'device': {  'brand': 'Apple',
                'family': 'Mac',
                'model': 'Mac'},
   'os': {     'family': 'Mac_OS_X',
               'major': '10',
               'minor': '9',
               'patch': '4',
               'patch_minor': None},
   'string': 'Mozilla/5.0_(Macintosh;_Intel_Mac_OS_X_10_9
....._4)_AppleWebKit/537.36_(KHTML,_like_Gecko)
....._\
Chrome/41.0.2272.104_Safari/537.36',
   'user_agent': {  'family': 'Chrome',
                    'major': '41',
                    'minor': '0',
                    'patch': '2272'}}
```

Listing A.1: Example of using the *Parse* function [23]

Appendix B

Additions to Chapter 2

B.1 Number of Flows per User Agent Label

Label	Number of flows
Other	289797163
Empty	118683118
Windows	74152949
Linux	5661875
Android	1741906
Ubuntu	557032
Debian	513565
iOS	312982
Mac OS X	242470
Red Hat	85230
WebOS	34253
HOFER	25874
CentOS	21531
Samsung	17772
NABO	14017
Chrome OS	10633
FreeBSD	9289
Fedora	3293
Chromecast	2366
Tizen	1400

Table B.1: Number of flows per user agent label captured in the given data set

B.2 Number of Devices per User Agent Label

Label	Number of devices
Other	176192
Empty	173100
Windows	81872
Linux	21038
Android	11748
iOS	9652
Debian	5416
Mac OS X	1420
Ubuntu	746
Red Hat	485
CentOS	319
Chrome OS	20
Tizen	18
Fedora	17
WebOS	9
Samsung	4
FreeBSD	3
HOFER	2
Chromecast	2
NABO	1

Table B.2: Number of devices that used user agents with the certain label

Appendix C

Attachment Information

The following files are part of the electronic attachment of this work:

- Three Python notebooks with the source code:
 - The first file contains the filtering process: reading the input file from the directory, applying filters, showing some statistics, graphs, and other information about the resulting dataset, and saving the output file to the external directory. (*Filtering.ipynb*)
 - The second file describes the labeling process: it downloads some files made by the first notebook, plots information about the dataset, and labels the given list of devices. The list of devices and their labels are saved into a given directory. (*Labeling.ipynb*)
 - The last notebook loads some output files of the first notebook, a list of labeled devices from the second notebook, and combines the information to prepare a dataset for vectorization and training the classifiers. The input dataset is filtered, transformed, vectorized, divided into train and test sets, and passed for training the classifiers with the default parameters or running the *GridSearchCV* function. (*TrainingAndEvaluation.ipynb*)
- The file with the information about the notebooks (*README.txt*)

Computational results and outputs of some cells were redacted and anonymized to protect personally identifiable information.