User Privacy Implications of DNS Fingerprinting

Study Project Report

Ishan Lamsal

Samuel Amogbonjaye

March 2023

This report is submitted to the

Chair of IT Security

Brandenburg University of Technology, Germany

Examiners:
Prof. Dr.-Ing. Andriy Panchenko
Asya Mitseva, M.Sc.

Contents

1 Introduction

The rise of the Internet and the increased usage of digital technologies in our daily lives have led to an increase in the amount of personal data collected and processed by organisations. While personal data can be helpful for various purposes, such as marketing and research, it can also be used to track and monitor individuals online without their knowledge or consent [19]. When users visit websites over the Internet via browsers, the browser uses the Domain Name System (DNS) to know the Internet Protocol (IP) address of the website. While using DNS, personal data such as the user's IP address and the website queried to DNS servers are visible by a passive observer observing network traffic. These data can be used to create a profile and identify the user [20, 54]. DNS over Transport Layer Security (TLS) (DoT) is a standard that encrypts DNS traffic, thereby protecting user's personal data (e.g. websites queries) from adversaries such as malicious actors who may intend to run phishing attacks, Internet Service Provider (ISP)s who may intend to censor user's activities and advertisers who may intend to spam users with adverts based on the user's interest. However, even with the usage of DoT, a technique such as DNS fingerprinting [66, 67] can be used by adversaries – malicious actors, ISPs, advertisers – to identify users by observing encrypted DNS traffic patterns. DNS fingerprinting is a type of attack that involves defining a set of websites to identify, collecting encrypted DNS / network traffic for these websites and using machine or deep learning techniques to develop a model based on complex relationships between the obtained network traffic and defined set of websites. This developed model is then used later along with new DNS / network traffic of a user to possibly identify websites (from the defined set) that a user visits.

Users who browse the Internet have the right to keep personal data (such as the websites they visit) private, especially from adversaries (see above) who have not received consent from these users to use their personal data. The use of techniques like DNS fingerprinting can violate that right by allowing adversaries to know what websites these users visit and then use that information for phishing campaigns, censoring, targeted advertising and other forms of digital profiling, which can be harmful to individuals and communities. Similarly, governments and other organisations can use DNS fingerprinting to monitor and track the activities of individuals and groups.

In this project, we implement and evaluate a DNS fingerprinting attack. Our experimental results demonstrate that DNS encryption alone does not provide sufficient confidentiality to user data. The rest of this report is structured as follows. Chapter 2 gives background information on the operation of DNS and how it has evolved to incorporate confidentiality. Chapter 3 provides the methodology used in our project to identify simulated user-visited websites from encrypted DNS traffic. This chapter also provides a detailed description of every practical aspect of this project, such as tools, programming languages, drivers and libraries needed to implement this project. The chapter expands on the websites we used to

collect encrypted DNS network traffic from and how we analysed and extracted features from this DNS traffic for our machine and deep learning models. In Chapter 4, we evaluate our implementation and show the reliability of the DNS fingerprinting attack in our experiments - that is, to what extent can an adversary effectively determine with certainty the websites visited by a user who uses the encrypted domain name system – DoT – while browsing. Finally, we conclude this report in chapter 5, discussing possible interesting questions and setting a path for future work in the technique of DNS fingerprinting.

2 Background

Domain Name System (DNS) [51] is a critical component of the Internet that translates human-readable domain names into IP addresses [39] that computers can understand. The DNS system is made up of a hierarchy of servers, beginning at the root server. The root servers are the starting point for domain name resolution. It hosts records of the list of Authoritative Name Server (ANS) for the different top-level domain names (e.g., .com, .edu, .de, etc) [37]. When a user visits a website via a browser, the user inputs a readable domain name (e.g., www.example.com). The browser does not know where to fetch the necessary web information. Thus, the user's computer contacts a DNS resolver and requests to obtain the IP address for www.example.com. The DNS resolver, in turn, contacts a root server and requests the IP address of the name server that is authoritative for .com. The DNS resolver then contacts the ANS for .com to obtain the ANS for example.com. This process continues until the DNS resolver obtains the IP address of the ANS for the complete domain name (in this case, www.example.com). The DNS resolver, in turn, contacts the last ANS and requests the A record for www.example.com. An A record is a domain name record to IP address mapping. Finally, the DNS resolver sends back a response to the user's computer with the IP address of www.example.com. The browser now knows the IP address of the web server to contact in order to fetch the web content for www.example.com. It is important to note that while the user computer can act as a DNS resolver that follows the process described above to resolve a domain, the user's computer typically delegates another independent DNS resolver. For our project, we adopt the latter.

DNS was initially designed without any security mechanisms, and the communication between a user's computer and the DNS resolver is insecure. Man in The Middle (MiTM) attacks such as DNS spoofing and DNS Cache poisoning were possible due to no validation of DNS responses at the DNS resolver to ensure the responses came from authoritative sources. This meant an attacker could poison DNS caches of DNS servers by sending fake DNS responses with IP addresses pointing to some server under the attacker's control. Every computer using these poisoned DNS resolvers would receive incorrect DNS entries. An attacker could further launch a phishing attack on the spoofed IP address. Domain Name System Security Extension (DNSSEC) [1] came along to address the issues with MiTM attacks. The idea of DNSSEC is to add a security layer to DNS by using digital signatures to verify the authenticity of DNS records. In DNSSEC, each DNS record is digitally signed using public key cryptography. The public key for a set of DNS servers is stored as a resource record on the respective name servers and cryptographically signed. A DNS resolver first contacts the root DNS server, uses DNSSEC to verify the records received and in turn follows a chain of trust from the root DNS server down to the lower hierarchy domain servers. The DNS resolver can then respond with a DNSSEC-signed response, and the client, on receiving the DNS query response, can choose to validate it through a process known as DNSSEC validation.

Even with DNSSEC, the issue of confidentiality is still not addressed. An adversary with a network traffic listener like Wireshark [18] can eavesdrop on DNS communications and know what domain name is being resolved and sent to the client. This is because DNSSEC does not provide encryption for the DNS queries and responses, which are sent in plain text.

DNS over TLS (DoT) [35] is an approach that provides encryption to the domain name resolution process. With DoT, DNS requests and responses between the client and DNS resolver are encrypted using Transport Layer Security (TLS) [22], making it more difficult for attackers to snoop on DNS traffic.

To establish a secure DoT connection, the client initiates a TLS handshake with the resolver by sending a "ClientHello" message that includes the TLS version and cyphers supported by the client. The resolver responds with a "ServerHello" message that includes the TLS version and cipher suite selected for the connection. Once the TLS handshake is complete, the client and resolver exchange DNS queries and responses over the encrypted channel using Transfer Control Protocol (TCP) [24]. The use of TCP ensures that larger DNS responses can be transmitted without being dropped. Additionally, DoT clients can authenticate the resolver's TLS certificate, ensuring they communicate with the intended resolver and not an imposter.

However, while DoT, due to its use of TLS to encrypt DNS traffic, makes it much harder for adversaries to determine what websites or domains users visit, it is still not sufficient as we will show in our study project. We will show how DNS fingerprinting can be used by adversaries to make a very good guess about what websites a user visits.

DNS fingerprinting is a type of traffic analysis attack based on supervised machine learning. Firstly, the attacker selects a list of websites to detect. Secondly, the attacker collects encrypted DNS traces for each of these websites by recording network traffic while visiting the websites. A DNS trace is an aggregation of packets from DNS network traffic collected while a website is visited. Thirdly, the attacker analyzes the traces to identify any features (unique traffic characteristics) that could be used to create a DNS fingerprint for each website. A DNS fingerprint is a distinctive sequence of characteristics or features (number of packets, packet ordering, transmission time statistics) that (possibly) identifies a subpage/domain name.

Fourthly, the attacker divides the generated DNS fingerprints into training and testing data. Training data is the data used to teach a machine learning classifier to make predictions, while testing data is used to evaluate the classifier's performance after training by comparing its predictions to the known outcomes. In all our Machine Learning (ML) classifiers, we also apply k-fold cross-validation. Cross-validation is used to evaluate the performance of a classifier by splitting an input dataset of into a specified ratio of training and testing data (better referred to as the validation data). Cross-validation helps to get a feeling of how a classifier is likely to perform on new, unseen data. K-fold cross-validation [3] goes one step further and offers a better generalization of the classifier's performance. The dataset is split into training and testing sets, for 'k' times, where each data is present in the testing set just once. This results in a classification model capable of associating an unknown DNS fingerprint – obtained from new network traffic from a real user that the classifier was not trained on – with a particular website. A classifier is a machine or deep learning model

that assigns input data to predefined categories or classes based on learned patterns and relationships.

To evaluate the performance of our classifier, we consider the following common ML metrics – accuracy, precision, recall and Mean Squared Error (MSE) [5]. Accuracy is the proportion of correctly classified samples out of the total number of samples. Precision is the proportion of true positives (correctly predicted positives) out of all positive predictions (both true positives and false positives). Recall is the proportion of true positives (correctly predicted positives) out of all actual positives (both true positives and false negatives) while MSE is the average squared difference between the predicted and actual values [5].

We focus on three machine learning – Multinomial Naive Bayes (MNB), k Nearest Neighbors (kNN), Stochastic Gradient Descent (SGD), and two deep learning classifiers – Residual Networks (ResNet), Convolutional Neural Networks (CNN). MNB [65] is a probabilistic algorithm used in text classification that estimates the likelihood of each feature for each class based on Bayes' theorem [6], kNN [30] is a non-parametric algorithm used for classification and regression that predicts the class label of a new data point based on the class labels of its k nearest neighbors in the training data while SGD [17, 68] is an iterative optimization algorithm used to minimize the loss function in a machine learning model by updating the model's parameters in the direction of the negative gradient of the loss function. ResNet [33] is a Deep Neural Network (DNN) used for classification that involves the use of residual blocks (consisting of convolutional layers), which allows the network to skip over some layers (called a shortcut connection) in order to better preserve important information from the input. CNN [29] is a DNN that uses convolutional layers to extract features from input images and pooling layers to reduce the spatial dimensions of the features, followed by one or more fully connected layers for classification. Finally, the attacker applies this classification model to identify the website corresponding to an unknown encrypted DNS trace from a real user [61].

In our project, we adopt the following threat model. We consider an adversary located between the computer of the user and the DNS resolver. The adversary is only able to passively observe encrypted DNS traffic communication. The adversary, by simply observing encrypted DNS traffic communication, is only able to tell whether or not a user's computer is using DoT and what DNS resolver is contacted by looking up the IP address of the DNS resolver. The passive adversary cannot determine what websites the user visits, given the communication is encrypted. We follow this threat model to illustrate how such a passive adversary, by applying DNS fingerprinting, can determine the index webpages of websites the user visits with certainty.

We additionally consider the adversary observes encrypted DNS traffic communication in both closed- and open-world scenarios. In a closed-world scenario, the adversary has a predefined list of websites of interest (also called foreground websites) and assumes that a user only visits the websites from this list. In an open-world scenario, the user can also visit other websites, not on the adversary's list of websites of interest. The attacker considers the websites that are not of interest as noise (often called background websites). Furthermore, for our closed-world scenario, we consider multiclass classification while for our open-world scenario, we consider binary and multiclass classification. Binary classification aims to predict whether the class of a feature vector/fingerprint is either part of the foreground websites or

background websites, while multiclass classification aims to predict the exact class/website a DNS fingerprint belongs to.

3 Our Approach

3.1 Overview of Our Approach

Our approach involves five key parts. We elaborate on these key parts in the coming sections:

- Part 1: Implementation of a website crawler tool to automate the collection of domain names from the Tranco top-site ranking list [48].
- Part 2: Implementation of browser automation using Selenium to simulate a real user who visits the collected domain names as well as the automated collection of network traffic generated through the visit of the websites.
- Part 3: Manual and automatic generation of features needed for classification of the websites, through traditional machine learning methods.
- Part 4: Creation of an ensemble machine learning model for classification of websites.
- Part 5: Creation of deep learning models based on two different deep neural networks and evaluation of these networks.

3.2 Experimental Setup and Used Software

In order to orchestrate the experiments executed in this study project, we made sure to:

1. Install and setup the following software:
    - Linux Ubuntu Virtual Machine (VM) as a base Operating System (OS) [13],
    - Python 3.9 programming language [59],
    - Firefox as a browser application [23],
    - The Selenium framework for browser automation [63],
    - Geckodriver to interface between Firefox and Selenium [28],
    - The Scapy Python Library for capturing traffic information [7],
    - The Pyshark Python Library for analysing and extracting data from the collected network traffic [44],
    - The Scikit-learn Python Library for machine learning [73],
    - The Tensorflow [2] and Keras [15] Python libraries for deep learning,

- The Joblib Python library for serialization of Python objects in certain cases [41],
- The Matplotlib [36], Pandas [81], and Numpy [32] Python libraries for data pre-processing, analysis and visualisation,
- The Urllib [58] and Beautiful Soup [60] Python libraries for handling and parsing of Hypertext Transfer Protocol (HTTP)/Hypertext Transfer Protocol Secure (HTTPS) requests and responses,
- The Dnspython [31] Python library for handling DNS requests and responses
- The Argparse [25] Python library for argument parsing from the command line,
- The Threading [27] and Concurrent [26] Python libraries for multithreading and multiprocessing purposes,

2. Configuration of DoT at OS level by editing the `/etc/systemd/resolved.conf` file of the `systemd` software [69] – a Linux system service for managing DNS resolvers – to perform the following actions:
   - Enabling of DoT – to enable DNS resolution via DoT; this was done by setting `DNSOverTLS=yes` in the `resolved.conf` file,
   - Enabling of DNSSEC – to ensure authentic DNS responses (and no spoofed responses) are obtained from DNS resolvers; this was done by setting `DNSSEC=no` in the `resolved.conf` file,
   - Disabling of DNS caching – to ensure there is no temporary storage of DNS records and that the DNS resolver is always contacted for a complete DNS response; this was done by setting `Cache=no` in the `resolved.conf` file,
   - Usage of Cloudflare's DoT resolvers (1.1.1.1, 1.0.0.1) – as DNS resolvers by setting `DNS` = `1.1.1.1, 1.0.0.1` in the `resolved.conf` file.
3. Apply the following preferences to Firefox to disable DNS caching in the browser's backend:

Preference	Value
browser.cache.disk.enable	False
browser.cache.memory.enable	False
network.dns.disablePrefetch	True
network.dnsCacheEntries	False
network.dnsCacheExpiration	0
network.dnsCacheExpirationGracePeriod	0

Table 3.1: Firefox preferences and values configured to disable browser caching

3.3 Collection of Domain Names and Generation of List of URLs

In order to analyze the efficiency of the DNS fingerprinting attack, domain names are needed. To this end, we had to make sure to collect a certain number of domain names that will be computationally feasible to work with throughout the project given the limited computing resources available to us. In total, we collected 7,050 valid and unique domain names – after implementing several checks and filters. These checks and filters are explained later in this chapter. These domain names were collected in random order from the Tranco Top 1 million domain names[1] [48]. We implemented random domain name collection by dividing the list of 1 million domain names into 8 large groups of 125,000 domain names and each large group is further divided into chunks of 100 domain names. Each chunk is in turn divided into 5 equal parts with each part having the following probability of being selected – 45%, 25%, 15%, 10.5% and 4.5%. In one complete iteration for randomly selecting one domain name, one large group is randomly selected and subsequently, one chunk from the selected group is chosen randomly. Then, a part of the chunk is selected based on its respective probability for the 5 parts of a chunk and finally, a domain name is randomly chosen from the selected part. The iterations continue until 7,050 domain names are selected. One of the reasons for a random collection of domain names was to prevent the possibility of alarming server administrators when we begin probing these domain names since several domains could be hosted by a single server. We also made sure to filter out bad domain names i.e., domain names that did not resolve after a five seconds timeout. We used a five seconds timeout to avoid spending unnecessary waiting times on domain names that may end up not resolving.

Each domain name was then converted to a valid Uniform Resource Locator (URL) of the main page of the corresponding website. To obtain the main page of a corresponding website for a domain name, we prepended *https://* to the domain name string and then used the `session.get` method of the Urllib Python library [58] to make a Hypertext Transfer Protocol (HTTP) GET request to the link. While the HTTP GET response will also get received if only a domain name string is specified in the request, we prepended *https://* to prioritise the visit of the secure (i.e., HTTPS) version of the website since certain websites, while offering support for Hypertext Transfer Protocol Secure (HTTPS), may serve a HTTP response unless explicitly specified in the HTTP GET request. Upon receiving each HTTP GET response for the main page URL, we carry out certain checks to determine if the received main page URL is valid. We check that there is no redirection between sending an HTTP GET request and receiving the response. We also check that the main page URL has not contain CAPTCHAs and that the main page URL shows some content. If the returned URL response contains the domain name specified in the HTTP GET request, we are able to check that no redirection occured. Note that we do not consider redirections as bad as long as it is within the same domain. To ensure no CAPTCHAs existed on the URL, we used the `webdriver.FirefoxOptions` class of the Selenium Wire library [43] to obtain the complete HyperText Markup Language (HTML) code for the URL. We in turn looked through the HTML code to see if there are any Application Programming Interface (API) calls to popular Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) providers like Google reCAPTCHA [4], hCaptcha [38] and Geetest [85].

[1] Available at https://tranco-list.eu/list/W9W39.

One way we check if API calls are made to these websites is by doing a string search on the HTML call to see if there are URLs to these APIs exist in the URL code. We could only consider these CAPTCHA providers given the time limitations of our project. By checking that the HTML code is at least 7000 bytes, we are able to confirm content exists on the URL page. We considered 7000 bytes a reasonable threshold after comparing the content size of several webpages. If a URL fufilled these criterias – a HTTP GET response is received containing the domain name, there is no redirection between HTTP GET request and response, there is at least 7000 bytes of web content and no CAPTCHA exists – we accept it as the main page URL of the website. We obtained 7,050 URLs of main pages for the corresponding websites of the 7,050 domain names.

The next step in the generation of a list of URLs was the collection of 5 URLs of five other unique webpages for each of our domain names making a total of 35,250 other webpages. To collect these subpages, we always start from the URL of the main page (which we had previously recorded) and select 5 pages from it. We use the *BeautifulSoup* class of the Beautiful Soup Python Library to automatically parse and find href attributes in the HTML code collected for each URL of a main page. If the href attribute starts with a http:// or a / and like the URL of the main page, contains the domain name for the website, we consider the link in the href attribute to be a subpage for that website. If a main page URL had links to less than 5 webpages, we recursively searched through the available webpages to get up to 5 subpages from the same domain. For each subpage, we also ensured that it fufilled the criterias presented in an earlier paragraph – a HTTP GET response is received containing the domain name, there is no redirection between HTTP GET request and response, there are at least 7000 bytes of web content, and no CAPTCHA exists – before accepting the subpage as valid for that website. Our reason for collecting the URL of the index webpage and 5 non-index webpages for a website was to identify the webpage for that website having the highest number of web resources. A web resource is any file having a URL and that is accessible through the World Wide Web (WWW). Each web resource provides a piece of the content of a webpage to load. Examples of web resources include image, HTML, Extensible Markup Language (XML), JavaScript files, etc. We considered both web resources loaded from the web server of a given website and web resources loaded from other web servers needed for that website. In Figure 3.1, we provide a Cummulative Distribution Function (CDF) plot of the number of web resources needed to load all of the 35,250 non-index pages for the 7,050 domain names. We observe that about 60% of all webpages require no more than 200 web resources to load content. We stored all this information – the 7,050 domain names, the URLs of their corresponding main pages, the URLs of five additional non-index pages for each of the domain names, the number of local and foreign web resources needed for each subpage and the URL of the selected subpage per domain – into a database. To uniquely identify each of the selected subpages, we mapped their URLs to an identity (ID) number which we chose to be the position of the corresponding name in the Tranco Top 1 million domain name ranking [48].

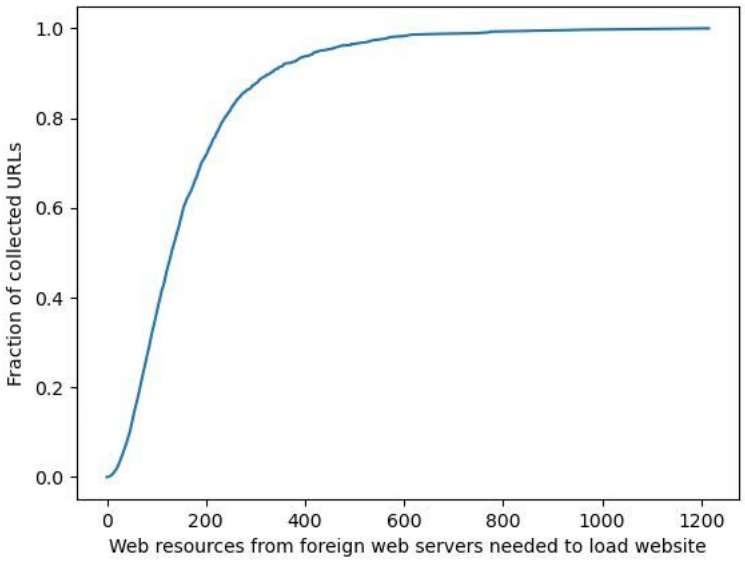

Figure 3.1: Fraction of URLs consisting of foreign web resources at least or below the specified number

3.4 Browser Automation and Collection of DNS Traces

So far, we have created a list of webpages we want to detect. Next, we need to create DNS fingerprints for these webpages. These fingerprints form the dataset used for machine and deep learning models. To create DNS fingerprints, we automated the browser visit to each subpage as a real user would. We used the Selenium framework [63] to automate browser visits to webpages by setting the *general.useragent.override* preference in Firefox to a user agent request header[2] that a user is likely to use while browsing. A user agent request header is simply a string included in a HTTP request that allows the webserver to detect the browser software, version and operating system used to send the HTTP request. We can set this user agent using the *webdriver.FirefoxOptions* method of the Selenium Wire library [43]. Other preferences we set are shown in Table 3. For the Selenium framework to automate webpage visits, the Gecko driver [28] is required on our OS. The Gecko driver [28] is the actual driver software of the OS that launches the Firefox browser [23] while the Selenium [63] and Selenium-Wire [43] Python Libraries allows a programmer to interface with Gecko driver [28] to apply these settings or preferences to Firefox. We visited 50 of the subpage URLs 80 times and the rest of 7000 URLs of them once, totalling 11,000 fetches. We call the *webdriver.Firefox.get* method of the Selenium Wire library [43] and passing in a variable containing the Firefox preferences[3], we can obtain the complete HTML code for a webpage. While automating webpage visits via the Selenium framework [63], we recorded the encrypted DNS traffic by using the Scapy library to listen to and capture DNS traffic on port 853. We

[2]Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:106.0) Gecko/20100101 Firefox/106.0
[3]See Table 3 and the paragraph that follows it

capture DNS traces by defining `filter="tcp and port 853"` for the `AsyncSniffer` class of Scapy [7] given the DoT standard uses the TCP over this port by default. After a page load, we use the `stop` method of the `AsycSniffer` class to stop DNS traffic capture and in turn, write the captured DNS traffic to a `.pcap` file using the `wrpcap` function of the Scapy Python library [7]. To ensure that complete DNS traces are collected, and the entire information exchange for a specific visit to a webpage takes place, we disabled DNS caching and disabled URL caching on Firefox (see part 3 of section 3.2 for how we achieved this). While automating browsing visits to obtain HTML code for the webpage, we set a timeout of 20 seconds to avoid unnecessary waiting times. For each page load, we implemented a check to validate the success of the page load to avoid distorted datasets in our experiments. That is, we consider URL fetch as successful and valid if there is no empty page or multiple tabs loads, no new tab page, no redirection and no error. Validating each URL visit before moving on to other URL visits is important as invalid page loads will yield DNS traces and, in turn, DNS fingerprints that may negatively influence the detection rate of our DNS fingerprinting attack.

To detect if an empty or a new page is loaded, we check the URL of the requested webpage returned by the *webdriver.FirefoxOptions.current_url* method of the Selenium Wire library [43] for the strings `about:blank`, `about:newtab` and `about:home`. If any of the strings are present in the returned URL, we determine that either an empty or new page got loaded. To detect multiple tabs, we fetched the value returned by the *seleniumwire.webdriver.FirefoxOptions.windows_handle* method [43]. We know multiple tabs got loaded if the returned value is more than 1 for a URL visit. Furthermore, to check for redirections, we adopt the similar approach we used to check for redirections of the URL of a main page. We observed the URL string returned by the *webdriver.FirefoxOptions.current_url* method of the Selenium Wire Python library [43] and check if the domain name for that webpage exists in the returned string. Finally, to check for errors during page loads, we checked the status code returned by *webdriver.FirefoxOption.requests.response.status_code* method of the Selenium Wire library [43]. If the status code is either between 400 to 499 or 500 to 599, we determine that an error occurred during page load and discard the URL. These status codes are standardized status codes that can be used to identify client and server error responses [52]. We would be visiting a URL more times than a normal user would do in a short time, a website operator could assume a likely Denial of Service (DoS) attack and choose to display a CAPTCHA dynamically. Because we do not consider URLs showing CAPTCHAs, we also implement the CAPTCHA check from section 3.3 to filter out when a website dynamically loaded CAPTCHA for a given URL visit. We do not consider webpages with CAPTCHAs because a webpage showing CAPTCHA is likely to hide a chunk of its content until the CAPTCHA is completed. If the DNS traces from the variation in the display of CAPTCHA for each load of the same webpage is used to generate DNS fingerprints, the detection rate of our DNS fingerprinting attack can be negatively influenced. In the next section, we further outline how a DNS fingerprint for a webpage is generated.

URL	Start Timestamp	Packet 1	Packet 2	Packet 3	Packet 4
https://www.aviso.bz/pay_today	1670587251	1670587251.155225:+90	1670587251.155243:+152	1670587251.155327:+176	1670587251.161312:-558
https://www.aviso.bz/pay_today	1670587267	1670587267.169783:-558	1670587267.170031:+90	1670587267.170042:+167	1670587267.170142:+191
https://www.aviso.bz/pay_today	1670587283	1670587283.437076:+90	1670587283.437089:+152	1670587283.437131:+176	1670587283.458518:-1028
https://www.aviso.bz/pay_today	1670587300	1670587300.188353:+90	1670587300.188377:+152	1670587300.188426:+176	1670587300.209941:-1028
https://www.aviso.bz/pay_today	1670587316	1670587316.415361:+90	1670587316.415375:+152	1670587316.415455:+176	1670587316.460846:-1028
https://www.aviso.bz/pay_today	1670587333	1670587332.762409:+90	1670587332.762435:+152	1670587332.762499:+176	1670587333.033735:-1028
https://www.aviso.bz/pay_today	1670587359	1670587358.747727:+90	1670587358.747755:+152	1670587358.747782:+176	1670587358.769405:-558
https://www.aviso.bz/pay_today	1670587376	1670587376.188115:-1498	1670587376.188375:+90	1670587376.188383:+174	1670587376.194095:+90
https://www.aviso.bz/pay_today	1670587392	1670587391.971012:+90	1670587391.971024:+152	1670587391.971104:+176	1670587391.990918:-1028
https://www.aviso.bz/pay_today	1670587408	1670587408.129524:-1026	1670587408.12973:+90	1670587408.12974:+152	1670587408.129787:-1968

Figure 3.2: Sample of our dataset containing 11,000 DNS traces

3.5 Dataset Generation and Feature Extraction

3.5.1 Dataset Generation

As previously mentioned, out of the 7050 collected URLs for the selected webpages, we randomly selected 50 of them to be the foreground URLs (see Table B.1) and fetched them 80 times. We visited the remaining 7000 URLs to be the background URLs and visit them only once. This resulted in a dataset of 4000 DNS traces (which we call foreground traces) and 7000 DNS traces (which we call background traces), totalling 11,000 DNS traces. Each foreground URL represents a class to be classified by the classifier, resulting in 50 classes for our closed-world scenario. The background websites are not websites of interest and are considered noise, we grouped all of them into a single class. This resulted in 51 classes for our open-world scenario. This meant that our closed-world scenario involved the use of all 4000 foreground DNS traces, while our open-world scenario involved the use of 11,000 DNS traces. In addition to storing the HTML code and the DNS traffic for each URL fetch, we also store a `.png` file containing a screenshot of the complete page. Also, we stored important logs in a text file, e.g., the URL name being fetched, the start and end timestamps of a page load, whether or not page load, packet capture and extraction of DNS traces were successful. While the DNS traces for each URL fetch were the main data needed for the evaluation of the DNS fingerprinting attack, we stored this additional information for manual checks to identify problems in case the implemented code stops working. In total, we made sure to have a dataset of 11,000 DNS traces (approximately 11 GB). Each DNS trace in our dataset is represented as follows:

[`URL`], [`start timestamp`], [`timestamp`][4], [`direction`][4], [`size`][4], [`timestamp`][5], [`direction`][5], ...

Figure 3.2 shows a snippet from our extracted DNS traces.

Note that a DNS trace contains several packets. We placed every DNS trace into individual Comma Seperated Values (CSV) files per URL to allow for better organisation of our dataset. Finally, for easy recollection of another dataset and for future reference and usage, we created a toolbox. The toolbox can parse multiple arguments from the command line, such as the list of URLs to be fetched, the number of DNS traces to be collected for each URL, the name of the network interface to listen on for network traffic and the time (in seconds) to wait before moving on to fetch the next `URL`.

[4] These represent the first packet
[5] These represent the second packet

3.5.2 Feature Extraction

In the previous section, we mention two groups of URLs that were collected – 50 URLs that were fetched 80 times (these are the foreground URLs) and 7000 URLs that were only fetched one time (these are the background URLs). Each foreground

After collecting both the foreground and the background traces, now we can extract features from each DNS trace. In the context of machine learning, features refer to a measurable aspect or characteristic of a data point that is used as an input to a machine learning model for prediction [9]. For example, in a dataset like ours, information such as the number of packets in a DNS trace, packet ordering and transmission time statistics are characteristic of a DNS trace that can be used to define features for a DNS trace. Our project considers two types of features – manually and automatically generated features. The manually generated features are hard-crafted and explicitly defined by a human expert with domain expertise, while the automatically generated features are automatically extracted from raw data using deep learning algorithms [50]. These features are computed for each DNS trace. While manual feature generation can be a time-consuming and labour-intensive process [42], it can result in a better representation of the data itself and in more interpretable models [83] given the inclusion of expert domain knowledge to define features to be extracted. In total, we have 70 manually generated features per DNS trace.

When it comes to manual features, it is important a domain expert defines the right features, as choosing the wrong features can greatly impact the accuracy and efficiency of the classifier [83]. This is especially important for manually generated features so that the most relevant and informative features that capture the underlying structure of the data are extracted. In contrast, with automatically generated features, the algorithm itself selects the features that it deems most useful. To this end, we generated the following features and only consider packets in our DNS trace, given the DNS fingerprinting attack is mostly focused on collecting and analysing DNS traces collected from DNS traffic, to detect patterns.

- Packet count – incoming packet[6] count, outgoing packet[6] count, total packet count, the ratio of incoming packets and the ratio of outgoing packets,
- The fraction of incoming and outgoing packets in the first 20 packets and in the last 20 packets of DNS trace,
- Length with the direction of the first 20 packets in a DNS trace,
- Packet inter-arrival time – between incoming packets and between outgoing packets,
- Mean, standard deviation, and third quartile of packet inter-arrival times measured between incoming packets and between outgoing packets respectively,
- Packet bursts[7] – maximum burst length, mean burst length and number of bursts

[6] Incoming packets are packets sent from the DNS resolver to the client computer while outgoing packets are packets sent from the client computer to the DNS resolver. We denote incoming packets with a + and denote outgoing packets with a - (see figure 3.2)

[7] In our project, we consider a packet burst as a sequence of outgoing packets following which there exist no two subsequent incoming packets.

To generate features automatically, we implemented an unsupervised DNN, called a Variational Autoencoder (VAE). A VAE [45] consists of an encoder network, a decoder network, and a latent space. The DNS traces are passed as input data into the encoder and are then mapped to the mean and variance parameters of a probability distribution. The combination of the mean and variance parameters constitutes the latent space while the output of the latent space is simply a compressed dimensive representation of the initial input data. This input data is in turn passed into the decoder in an attempt to reconstruct the initial input data in its initial dimension [45, 64, 71].

For our project, we implemented our VAE according to [64] using the Tensorflow Python library [80] and extract the output of the latent space as automatic features for our experiments. As DNN is typically used for image processing [8, 33, 64, 82], most DNN implementations and libraries are implemented using two-dimensional layers. However, the DNS traces that we give to our VAE as input are one-dimensional. Therefore, working with these implementations was quite challenging. We had to tweak the implementation in [64] for our use case by implementing the DNN layers as 1 dimensional. We did this by changing the `shape` keyword argument of the `keras.Input` layer class to the number of features we had for each DNS trace. We also replace the usages of the `Conv2D` class of the Tensorflow library [80], to `Conv1D` classes, among other things. Each declaration of the `Conv1D` class of the [80] Python library creates a hidden DNN layer, so called because a hidden layer is neither an input nor output layer and is not directly observable from the input or output layer. For each VAE model, we implemented two hidden layers (at both the encoder and decoder networks). The encoder and decoder part of each VAE model has its own input, output and hidden layers. Additionally, we apply three activation functions to each hidden layer – namely Exponential Linear Unit (ELU) [16], Rectified Linear Unit (ReLU) [10] and sigmoid [12]. We do this by setting the `activation` keyword of each instance of the `keras.Input` class [80] to the respective name of the activation function as defined in the Tensorflow API [80]. Additionally, we passed the values 3 and 2 into the `kernel_size` and `stride` arguments of the `Conv1D` class respectively, in order to change the kernel size and stride of each layer accordingly. We make all these changes to the VAE implementation in [64] to adapt the implementation to our one-dimensional DNS traces.

While the intention was to completely implement and evaluate two VAE models – one implementing Kullback-Leibler Divergence (KLD) [79] and the other categorical cross entropy respectively [78] – we only implement and evaluate one VAE model with KLD. This is due to time constraints. For the implementation of KLD as the loss function at the decoder, we adopt the KLD implementation of [64]. Both KLD and categorical cross entropy are loss functions. Loss functions are mathematical functions applied in machine learning that can be used to measure the error between the actual and predicted outcome. By applying a loss function, a machine learning model (with the help of an optimization function) is able to make intrinsic changes to keep the error to a minimum while increasing the performance of the model [56].

Now that our VAE model was ready, each DNS trace of several packets is passed into the VAE. The VAE in turn generates and provides the output features for each DNS trace. We extracted 44 features per DNS trace. We further extended our toolbox, introduced in section 3.5.1, to parse parameters from the command line that specify what VAE model is applied,

what decoder is used, the number of k-folds used (we explain the k-fold cross-validation in the next section) and the number of automatically generated features to be extracted from the VAE per DNS trace. Given we had both manually and automatically generated features for our dataset, we begin implementing our machine learning and deep learning based classifiers used for the detection of websites.

A sample of manual features can be seen in Table B.2, while a sample of automatically generated features can be seen in Table B.3

3.6 Machine Learning based Classifiers

MNB Parameters	Values
force_alpha	True
fit_prior	True

kNN Parameters	Values
n_neighbors	14
leaf_size	1
metric	cityblock
weights	distance

SGD Parameters	Values
loss	log_loss
learning_rate	adaptive
alpha	0.0001
class_weight	balanced
eta0	10

Table 3.2: Hyperparameters used for MNB, kNN, and SGD.

Following the steps of the DNS fingerprinting attack, we now make use of ML classification algorithms needed to identify websites based on their fingerprints. In particular, we focus on the following machine learning classifiers using scikit-learn [73].

- Multinomial Naive Bayes (MNB) classifier using the MultinomialNB API [76],
- k Nearest Neighbors (kNN) classifier using the KNeighborsClassifier API [77],
- Stochastic Gradient Descent (SGD) classifier using the SGDClassifier API [74].

In addition to the above, we implemented a custom classifier that ensembles the three classifiers above and chooses one prediction from the predictions of the three classifiers. We implement three independent decision-making methods for this Ensemble Classifier. In the first method, the Ensemble classifier randomly selects a prediction from the three classifiers MNB, kNN, and SGD). In the second method, the Ensemble classifier selects the prediction of the classifier, from the three classifiers, with the highest confidence value. In the third method, the decision is made in favour of the classifier that has the highest difference between its first two confidence values per class prediction. Each prediction from a classifier is accompanied by a set of confidence values assigned to each class in a given dataset. Confidence values in a classifier prediction refer to the level of certainty or probability that the predicted label or class is correct.

As part of dataset preparation for ML, we applied scaling. Scaling ensures input features have similar scale and range, avoiding bias in predictions and thus improving the performance and accuracy of machine learning models [84]. Two main types of scaling are normalization and standardization [84] – both implemented as MinMaxScaler and StandardScaler modules, respectively, in scikit-learn [73]. In normalization, every datapoint is scaled between two values, usually 0 and 1. In standardization, all datapoints are scaled to a standard normal distribution such that the mean and the standard deviation of the entire dataset is between 0 and 1 respectively. More information on their differences and how they are mathematically derived is available here [53]. For our dataset, we utilise the standardization scaling technique for the reason that it offers better accuracy and precision than normalization in our experiments. For MNB however, we utilize the normalization scaling technique because the standardization technique sometimes produces datapoints with negative values and MNB is unable to work with negative datapoints.

We also use k-fold cross-validation to generate training and testing sets from our dataset. We use k-fold given a single split of a dataset into training and testing sets creates a bias due to the fact that the generated model may not offer a proper generalization to new, unseen data [46]. In particular, we apply 10-fold cross-validation, given $k = 10$ is a common value for k [11]. Our dataset of DNS traces is passed into the classifier 10 times and at each time, the dataset is divided into 9 training and 1 testing groups. To implement 10-fold cross-validation, we invoked the `model_selection.StratifiedKFold` API of the Scikit-learn Python library [73] and set the `n_splits` parameter to 10. We use this API because it is able to automatically handle class imbalance and split each dataset into equal proportions, ensuring that each class of the foreground traces is equally represented in each of the 10 folds.

To implement MNB, we use the `MultinomialNB` API [76], provided by scikit-learn [73] and pass in values for the `force_alpha` and `fit_prior` parameters [76]. The `force_alpha` parameter is either set to True or False to specify whether the algorithm enforces the default value of 1 provided by the `alpha` parameter of the `MultinomialNB` API. We retain the default value of 1 for the `alpha` parameter to solve the zero probability problem in the MNB algorithm [40]. The `fit_prior` parameter defines whether or not class probabilities are learned from the training data. Because this is set to True, the algorithm does not assume all classes have equal probabilities prior to learning given in the case of our open-world scenario, for example, we have an unbalanced class distribution [76].

To implement kNN, we use the `KNeighborsClassifier` API [77] of scikit-learn [73] and pass in values for the parameters `n_neighbors`, `leaf_size`, `metric` and `weight` [77]. The `n_neighbors` parameter determines the number of nearest neighbors used to make predictions in the `KNeighborsClassifier`. The `leaf_size` parameter controls the number of points at which the `KNeighborsClassifier` switches its mode of calculation to compute the nearest neighbors. The `metric` parameter specifies the distance metric used to compute distances between points in the `KNeighborsClassifier`. The `weight` specifies the weight function used in prediction for `KNeighborsClassifier` [77].

To implement SGD, we use the `SGDClassifier` [74] respectively, and supply values for the parameters `loss`, `learning_rate`, `alpha`, `class_weight` and `eta0` [74]. The `loss` parameter is used to specify the loss function to be optimized during the training process. The `class_weight` parameter controls the regularization strength. The `learning_rate`

parameter determines the step size at each iteration while updating model parameters in the SGD algorithm while the `eta0` parameter specifies the `learning_rate` value to begin with.

We provide the parameters and values used for each classifier in Table 3.2. We came to the conclusion of using these parameters and values after using scikit-learn's `GridSearchCV` API [75] to automatically try out several combinations of parameters (called hyperparameters) for each classifier, in a process referred to as hyperparameter tuning. `GridSearchCV` works by taking as an input the entire dataset of features and a range of hyperparameter values. It in turn performs k-fold cross-validation and then simply returns the only hyperparameter values that generally performed best with our dataset across all folds.

3.7 Deep Learning based Classifiers

We also implement two types of Deep Neural Network (DNN), Residual Networks (ResNet) [33] and Convolutional Neural Networks (CNN) [29]. We create two Deep Learning (DL) models based on ResNet and one DL model based on CNN. ResNet consists of a defined number of residual blocks. Instead of having each layer directly output its own feature representation, residual blocks output the original input plus the feature representation learned by the layers. The addition is done in such a manner that the dimension of the input data remains the same. By doing this, the network can better propagate gradients through the layers. The output from the last residual block is then passed into an average pooling layer, then a fully connected layer with softmax activation function applied. The output is the predicted class for the initial input data [33]. CNN [29] consists of multiple layers including a convolution layer, a pooling layer, and a fully connected layer. The convolution layer applies a set of filters to an input image to extract its features. The pooling layer then reduces the dimensionality of the output by down sampling the feature maps. The fully connected layer then takes the resulting feature maps and performs classification or regression to produce the final output.

We provide further information on how we used all three models for the purpose of classification in Chapter 4. Like our VAE implementation we pass in raw DNS traces into the DL classifiers. If the DNS traces are not up to a predefined number of packets, then we pad the DNS trace with packets that have packet sizes of 0. For DL classifiers, we used Adam optimizer as an optimization function where we specify the learning rate (this is a hyperparameter, see chapter 4) and use categorical cross entropy as a loss function (given the goal is classification). A hyperparameter is a parameter that is set before the training of a model begins, and that cannot be learned from the data or during the training process. The learning rate in the Adam optimizer is a hyperparameter that controls the step size at each iteration during the optimization process. It determines how much to update the model parameters in the direction of the gradient of the loss function [21].

3.7.1 ResNet 18

We implement two variants of ResNet 18 called ResNet 1 and ResNet 2, following the implementation in [33]. These variants only differed in the type of input dataset passed into them. ResNet 1 that takes as input only raw DNS traces while ResNet 2 takes as input raw DNS traces and 5 features from the manually generated features presented in Section 3.5.2, namely incoming packet count, outgoing packet count, total packet count, the ratio of incoming packets and the ratio of outgoing packets. We implement both variants to provide variety for our experiments and to observe how the performance of a DNN may change with the inclusion of manually generated features.

While the authors of [33] make use of 2D convolution layers to implement the ResNet 18 network, given the goal is image classification, we instead implemented 1D convolutional layers similar to our VAE implementation in section 3.5.2. Furthermore, we implement one additional fully connected layer (with Scaled Exponential Linear Unit (SELU) activation function) before the softmax activated fully connected layer. We do this to ensure that the inputs to the softmax layer are properly scaled, thus improving the performance of the ResNet 18 model.

3.7.2 CNN

We model our CNN based classifier similar to the implementation in [70]. Our model consists of an input layer, 4 convolutional blocks and a flatten layer. Each convolutional block has a convolution layer, batch normalization layer, activation layer and dropout layer. Each layer in the convolutional block is activated with ReLU. We apply ReLU activation to introduce nonlinearity into the network, which is necessary for the network to learn complex representations from the input data [10]. The flatten layer flattens out the output from the 4 blocks into a one-dimensional vector of features capable of being passed into a fully connected layer. The fully connected layer is needed at the end and activated with softmax so that classification can be performed by the model.

Hyperparameter	Values tuned
No. of units	32, 64, 96, 128, 160, 192, 224, 256
Activation function	elu, relu, selu, sigmoid
Learning rate	0.01, 0.001, 0.0001

Table 3.3: Hyperparameter search space to determine best values per fold

For all our DL classifiers, we apply hyperparameter tuning using the `Hyperband` [49] class of the Keras Tuner [72]. In summary, for each DNN we implement hyperparameter tuning for (1) the number of units and (2) the activation function of the fully connected layer before the softmax activated layer and (3) the learning rate of the Adam optimization function. The number of units refers to the number of neurons in a particular layer and can affect the complexity and representational power of the model. The activation function is the parameter that affects the model's ability to learn complex patterns and avoid vanishing gradients. The learning rate, as explained in Section 3.6, is the parameter that determines the step size at

each iteration during training while updating model parameters. Table 3.3 shows the values we tuned for these 3 hyperparameters. The `Hyperband` API implements a more efficient and adaptive algorithm, than `GridSearchCV`, that uses a random search strategy coupled with early stopping to find the best hyperparameters in a shorter amount of time with fewer computational resources. We used the `Hyperband` class for our DL classifiers, instead of a time-exhaustive hyperparameter tuner like `GridSearchCV`, to save time. DL alone without hyperparameter tuning requires significantly more time and computing resources for training [14]. Hence, we did not use `GridSearchCV` to avoid increasing the time complexity for training our DL classifier, given the time constraints of our project.

4 Evaluation

The last step of our project is to evaluate the performance of the DNS fingerprinting attack. In this chapter, we discuss the evaluation results of all our classifiers as well as the experiments carried out on each classifier. In Section 3.6, we discuss the experiments we carried out on our traditional ML classifiers – MNB, kNN and SGD – and our Ensemble (ENS) classifier. We discuss the results accordingly in Section 4.1. In Section 4.2, we discuss the experiments used for DL classifiers – ResNet 1, ResNet 2 and CNN. We present the results of our DL classifiers in Section 4.2 and discuss the results. In the next two paragraphs, we discuss some general variables, aspects and metrics that apply to every experiment as well as discuss them. We also in this section, discuss the statistically metric used to summarise our results.

We use 10-fold cross validation as a constant variable throughout our experiments. We explain how we implemented 10-fold cross-validation in Chapter 3.6. For every case involving foreground websites, we make sure, that while applying cross validation, an equal proportion of foreground fingerprints per foreground website appears in every fold of training and testing sets. This is done to ensure that each fold contains a representative sample of our dataset thus avoid biases towards any particular class. We do not do this for background websites given that background websites are considered as noise and all belong to the same URL class. Furthermore in every experiment, we evaluated both manually generated and automatically generated features. We note that for each experiment, we do not record individual results per fold. Instead, we take the mean across all folds per experiment and this is the result we record. We consider the mean rather than individual results because the mean gives a more representative estimate of the central tendency of the data.

Last but not least, we observe the accuracy for a closed-world scenario and both precision and recall for an open-world scenario. As closed world consists of a balanced distribution of fingerprints per class, the accuracy offers a truthful representation of the performance of the classifier. In an open-world scenario, however, the background class has a warping 7000 fingerprints (compared to 80 for the remaining 50 classes) and as a result, the accuracy no longer becomes suited for properly evaluating the performance of models given the significantly unbalanced sizes of the classes. Instead, we observe the precision and recall. Also for both closed and open world scenarios, we measure the MSE to observe how often a fingerprint from the testing dataset is wrongly predicted by all three classifiers. For each experiment, certain variables are defined and the goal in general is to evaluate, given these variables, the performance of the classifier in correctly predicting a website for a DNS fingerprint (from our testing set), that was not seen during training.

4.1 Evaluation of Machine Learning classifiers

We evaluate all 4 ML classifiers — MNB, kNN, SGD and Ensemble. For the Ensemble classifier, we evaluate it using the second case as described in section 3.6. In order to evaluate our ML classifiers, we implement 12 experiments. We divide these experiments into 4 groups as a way of grouping experiments with similar characteristics. This allows for a better presentation and understanding of the results. Each experiment in an experiment group only differs by the number of input fingerprints used. In the next sections, we present and discuss the results of these experiment groups.

Experiment group 1

Experiment group 1 was executed in a closed world scenario using multi-class classification. Every DNS fingerprint from a DNS trace to be predicted in this experiment is from a foreground website. For each experiment in this experiment group, we used 20, 50 and 80 DNS fingerprints for classification of our websites respectively. In particular, for each foreground website in this experiment, we use 18, 45, and 72 DNS fingerprints as part of our training set for each website while 2, 5, and 8 DNS fingerprints were used as part of our testing set for each website. This resulted in a total of 3600 DNS fingerprints for our training set and 400 DNS fingerprints for our testing set, for each fold of the cross-validation process. These figures derive from our implementation of 10-fold cross-validation, given that 90% of the entire dataset is used for training while 10% of the entire dataset is used for testing in each fold.

Experiment group 2

Each experiment in this group was executed in an open world scenario using multi-class classification. To carry out this experiment, we combined the foreground fingerprints with the background fingerprints and implemented the `model_selection.StratifiedKFold` class of the scikit-learn Python library [73], as explained in Chapter 3, to divide the set of fingerprints into 10 folds of training and testing sets respectively. Each fold of foreground fingerprints consists of 20, 50 and 80 fingerprints per foreground website respectively. After splitting, the training set of each experiment had 900, 2250, 3600 foreground fingerprints, plus 6300 background fingerprints respectively. In the testing set, each experiment had 100, 250, 400 foreground fingerprints plus 700 background fingerprints.

Experiment group 3

The experiments in this group were executed in an open world scenario using binary classification. Each experiment used 20, 50 and 80 fingerprints per foreground website respectively, as well as all background fingerprints. Like in Experiment group 2, we combine the set of foreground and background fingerprints and then apply cross-validation to divide the dataset into 10 folds of training and testing sets. As well, the training set of each experiment had 900,

2250, 3600 foreground fingerprints, plus 6300 background fingerprints respectively. In the testing set, each experiment had 100, 250, 400 foreground fingerprints plus 700 background fingerprints.

Experiment group 4

In this experiment group, we execute an open world scenario using multi-class classification with each experiment using 1000, 5000 and 7000 background fingerprints. We added a constant 50 foreground fingerprints per foreground website to each experiment. This resulted in all 3 experiments having 1000 background fingerprints plus 2500 foreground fingerprints, 5000 background fingerprints plus 2500 foreground fingerprints, and, 7000 background fingerprints and 2500 foreground fingerprints respectively. In each of these experiments, 10-fold cross validation was applied to split the dataset into 10 folds of training and testing sets.

4.1.1 Results of Traditional Machine Learning classifiers

Tables 4.1, 4.2, 4.3, and 4.4 show the results of our traditional machine learning models. In every case, we observe better results when classifiers use manually generated features as opposed to automatically generated features. According to [62], this is quite expected as suitability of manually generated features to a particular problem is often the crucial factor that determines the better performance of a system when compared with automatically generated features [62]. We believe we have made use of domain specific knowledge to properly engineer manual features for our dataset, and, as a result, we have gotten more accurate predictive models than in the cases where automatically generated features were used[57]. We note that the worse performance of automatically generated features compared to manually generated features may be because the implementation of our autoencoder is not optimal and still has room for improvement. We notice that MNB appears to be the best performing traditional ML algorithm in our use case. At the same time, MNB requires the least runtime and memory usage in our experiments. This is evidently a win-win situation as in the real world, we do not have to trade cost for better performance. ENS takes the most time to run and consumes the highest amount of memory arguably due to the extra decision making process (explained in Section 3.4) – compared to the others – that it goes through before making a prediction.

Type of features	Using 20 FT[8]				Using 50 FT[8]				Using 80 FT[8]			
	MNB	kNN	SGD	ENS	MNB	kNN	SGD	ENS	MNB	kNN	SGD	ENS
Manually generated	0.13	0.27	0.18	0.2	0.15	0.28	0.26	0.27	0.15	0.3	0.32	0.35
Automatically generated	0.02	0.02	0.02	0.02	0.03	0.01	0.03	0.02	0.02	0.01	0.01	0.02

Table 4.1: Classification results for closed world with multi-class classification using all foreground websites. Metric used is accuracy.

[8]FT – number of foreground traces used per website for classification
[9]BT – number of background websites used for classification

Type of features	Metric	Using 20 FT				Using 50 FT				Using 80 FT			
—	—	MNB	kNN	SGD	ENS	MNB	kNN	SGD	ENS	MNB	kNN	SGD	ENS
Manually generated	Precision	0.93	0.95	0.05	0.67	0.93	0.62	0.1	0.65	0.92	0.18	0.17	0.65
	Recall	0.1	0.05	0.05	0.05	0.1	0.08	0.17	0.06	0.1	0.15	0.22	0.08
Automatically generated	Precision	0.98	0.27	0.19	0.21	0.09	0.08	0.1	0.07	0.08	0.05	0.05	0.05
	Recall	0.03	0.05	0.04	0.05	0.04	0.03	0.04	0.04	0.03	0.03	0.02	0.03

Table 4.2: Classification results for open world with multi-class classification for using all foreground websites and all background websites.

Type of features	Metric	Using 20 FT				Using 50 FT				Using 80 FT			
—	—	MNB	kNN	SGD	ENS	MNB	kNN	SGD	ENS	MNB	kNN	SGD	ENS
Manually generated	Precision	0.83	0.32	0.38	0.52	0.61	0.5	0.73	0.8	0.63	0.57	0.9	0.92
	Recall	0.03	0.12	0.85	0.68	0.3	0.62	0.92	0.84	0.92	0.92	0.91	0.92
Automatically generated	Precision	0.91	0.41	0.33	0.91	0.91	0.05	0.65	0.12	0.9	0.5	0.5	0.7
	Recall	0.05	0.05	0.91	0.05	0.05	0.07	0.91	0.05	0.52	0.45	0.4	0.4

Table 4.3: Classification results for open world with binary classification using all foreground websites and all background websites.

4.2 Evaluation of Deep Learning classifiers

We further evaluate the three DL-based classifiers – ResNet 1, ResNet 2, and CNN. In order to evaluate our DL classifiers, we implement 5 groups of experiments with 3 experiments per experiment group using multiclass classification in all cases, similar to the implementation of our experiments in the ML classifiers. We explain each of these experiment groups in the following sections.

Experiment group 1

Experiment group 1 was executed in a closed world scenario using multi-class classification and 2500 packets per DNS trace. Every DNS trace to be predicted in this experiment was from a foreground website. For each experiment in this experiment group, we used 20, 50 and 80 DNS traces for classification of our websites respectively. In particular, for each foreground website in this experiment, we use 18, 45, and 72 DNS traces as part of our training set for each website while 2, 5, and 8 DNS traces were used as part of our testing set for each website. This resulted in a total of 3600 DNS traces for our training set and 400 DNS traces for our testing set, for each fold of the cross-validation process.

Experiment group 2

Each experiment in this group was executed in an closed world scenario using multi-class classification and 5000 packets per DNS trace. We implement all the experiments in this group exactly like experiment group 1 of our DL classifier mentioned above. The only difference is the number of packets used per DNS trace.

Type of features	Metric	Using 1000 BT[9]				Using 5000 BT[9]				Using 7000 BT[9]			
—	—	MNB	kNN	SGD	ENS	MNB	kNN	SGD	ENS	MNB	kNN	SGD	ENS
Manually generated	Precision	0.76	0.05	0.22	0.05	0.92	0.44	0.15	0.5	0.93	0.62	0.1	0.68
	Recall	0.17	0.3	0.2	0.28	0.1	0.1	0.18	0.08	0.1	0.06	0.18	0.05
Automatically generated	Precision	0.98	0.49	0.08	0.78	0.99	0.8	0.07	0.99	0.99	0.81	0.05	0.99
	Recall	0.05	0.05	0.08	0.05	0.05	0.05	0.08	0.05	0.05	0.05	0.08	0.05

Table 4.4: Classification results for open world with multi-class classification using 50 DNS traces per foreground website.

Experiment group 3

The experiments in this group were executed in an open world scenario using multi-class classification and 5000 packets per trace. Each experiment used 20, 50 and 80 traces per foreground website respectively, as well as all background traces. To create our dataset for this experiment group, we borrow from our implementation of experiment group 2 in our ML classifier, where we combine the set of foreground and background traces, and then apply cross-validation to divide the dataset into 10 folds of training and testing sets. As well, the training set of each experiment had 900, 2250, 3600 foreground traces, plus 6300 background traces respectively. In the testing set, each experiment had 100, 250, 400 foreground traces plus 700 background traces.

Experiment group 4

In this experiment group, we execute an open world scenario using multi-class classification and 2500 packets per DNS trace. We implement all the experiments in this group exactly like experiment group 3 of our DL classifier mentioned above. The only difference is the number of packets used per DNS trace.

Experiment group 5

In this experiment group, we execute an open world scenario using multi-class classification and 2500 packets per DNS trace.

We with each experiment using 1000, 5000 and 7000 background fingerprints. We added a constant 50 foreground fingerprints per foreground website to each experiment. This resulted in all 3 experiments having 1000 background fingerprints plus 2500 foreground fingerprints, 5000 background fingerprints plus 2500 foreground fingerprints, and, 7000 background fingerprints and 2500 foreground fingerprints respectively. In each of these experiments, 10-fold cross validation was applied to split the dataset into 10 folds of training and testing sets.

4.2.1 Results of Deep Learning classifiers

Tables 4.5, 4.6, 4.7, 4.8, and 4.9 show the results of our deep learning models. All our experiments show no significant difference between results of all the implementations of our ResNet 1 (where only raw DNS traces are used) and ResNet 2 (where DNS traces are used with 5 manually engineered features). Furthermore, we notice in the second experiment of experiment group 3, the second experiment of experiment group 4, and in the second experiment of experiment group 5, ResNet 1 performs slightly better than ResNet 2. This is interesting as clearly, there was no need to specially engineer features for our ResNet 1 classifier in our use case. We now continue to compare the classification performance of ResNet 1, and not ResNet 2, with CNN given ResNet 1 peforms better than ResNet 2. We observe that CNN performs better than ResNet 1 consistently across all experiments. In an open-world scenario, we notice the existence of few experiments – experiment 1 of experiment group 5, experiment 3 of experiment group 4 and experiment 3 of experiment group 3 – where ResNet seems to not perform at all (with 5% precision and recall). See Tables 4.7, 4.8, and 4.9. We argue that this may be likely because in these cases, there are unrepresentative samples in the testing set [47]. We hypothesize that if we isolate the results of cross-validation folds for these three experiments instead of taking a mean across all folds, we are likely to find a fold with more than 5% precision and recall.

In our experiments, we noticed that in every case of closed-world scenarios, we have rather poor performance results from our models.For example, in Table 4.1 of the closed world with multi-class classification using machine learning, the best result we observe is 35% accuracy with the Ensemble classifier. Similarly, using deep learning with multi-class closed world scenarios as shown in Tables 4.5 and 4.6, the best result is 22% accuracy

Using 20 FT			Using 50 FT			Using 80 FT		
CNN	ResNet 1	ResNet 2	CNN	ResNet 1	ResNet 2	CNN	ResNet 1	ResNet 2
0.1	0.125	0.135	0.16	0.155	0.145	0.22	0.17	0.19

Table 4.5: Classification results for closed world using 2500 packets per DNS trace, multi-class classification, and all foreground and background websites. The metric used is accuracy.

Using 20 FT			Using 50 FT			Using 80 FT		
CNN	ResNet 1	ResNet 2	CNN	ResNet 1	ResNet 2	CNN	ResNet 1	ResNet 2
0.17	0.13	0.132	0.17	0.155	0.156	0.19	0.19	0.18

Table 4.6: Classification results for closed world using 5000 packets, multi-class classification, and all foreground and background websites. The metric used is accuracy.

Metric	Using 20 FT			Using 50 FT			Using 80 FT		
—	CNN	ResNet 1	ResNet 2	CNN	ResNet 1	ResNet 2	CNN	ResNet 1	ResNet 2
Precision	0.83	0.92	0.92	0.92	0.67	0.52	0.68	0.05	0.05
Recall	0.82	0.92	0.92	0.62	0.5	0.41	0.58	0.05	0.05

Table 4.7: Classification results for open world using 5000 packets per DNS trace, multi-class classification, and all foreground and background websites.

Metric	Using 20 FT			Using 50 FT			Using 80 FT		
—	CNN	ResNet 1	ResNet 2	CNN	ResNet 1	ResNet 2	CNN	ResNet 1	ResNet 2
Precision	0.83	0.93	0.93	0.74	0.64	0.58	0.68	0.05	0.05
Recall	0.83	0.93	0.93	0.68	0.48	0.24	0.6	0.05	0.05

Table 4.8: Classification results for open world using 2500 packets, multi-class classification, and all foreground and background websites.

Metric	Using 1000 BT			Using 5000 BT			Using 7000 BT		
—	CNN	ResNet 1	ResNet 2	CNN	ResNet 1	ResNet 2	CNN	ResNet 1	ResNet 2
Precision	0.49	0.04	0.04	0.82	0.78	0.61	0.93	0.72	0.77
Recall	0.1	0.04	0.04	0.59	0.67	0.65	0.61	0.78	0.77

Table 4.9: Classification results for open world using 5000 packets per DNS trace, 50 DNS traces per foreground website and multi-class classification.

5 Conclusion

Understanding the implications and techniques of DNS fingerprinting is essential for protecting individuals' privacy (e.g, the websites a user visits). In our study project, we are able to show that even with encrypted communications over the Internet, it is possible to determine with high certainty which websites are visited, given a known set of websites. We are also able to show that a malicious actor does not necessarily need domain expertise to carry out DNS fingerprinting attacks. We show in our project that the use of deep learning models is possible for accurate predictions without the need for manual engineering of features that require domain expertise of DNS. Particularly, CNN was able to achieve higher classification results across most experiments. Furthermore, we assume that the additional use of timing information – in particular, the individual inter-arrival time differences between each pair of packets – could likely improve the results as our classifiers will have more representative datasets to learn from [34]

As observed from our study project, using only packet size and direction extracted from network traffic information (DNS traces) has enabled us to determine websites visited by a user with approximately 70% precision and recall using CNN in an open-world scenario for multi-class classification. We prefer our results for the open world scenario. A user will likely be browsing in an open-world scenario. So an attacker would always have to assume an open-world scenario when implementing a DNS fingerprinting attack. Heretofore, an open-world scenario is more realistic than a close-world scenario [55].

Based on the results of our experiments, we conclude that DNS fingerprinting is a sophisticated and effective attack. There is a threat to the privacy of users that can be executed by a malicious actor with no expert knowledge in the DNS domain. This can be carried out with high precision spanning from 70% to 93%.

A Abbrevations and Acronyms

DNS	Domain Name System
TLS	Transport Layer Security
DoT	DNS over TLS
DNSSEC	Domain Name System Security Extension
CDF	Cummulative Distribution Function
ANS	Authoritative Name Server
TCP	Transfer Control Protocol
CSV	Comma Seperated Values
VM	Virtual Machine
OS	Operating System
IP	Internet Protocol
ISP	Internet Service Provider
MiTM	Man in The Middle
URL	Uniform Resource Locator
WWW	World Wide Web
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart
DoS	Denial of Service
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
HTML	HyperText Markup Language
XML	Extensible Markup Language
ML	Machine Learning
DNN	Deep Neural Network
DL	Deep Learning
VAE	Variational Autoencoder
KLD	Kullback-Leibler Divergence
kNN	k Nearest Neighbors
MNB	Multinomial Naive Bayes

SGD	Stochastic Gradient Descent
ENS	Ensemble
ELU	Exponential Linear Unit
ReLU	Rectified Linear Unit
SELU	Scaled Exponential Linear Unit
API	Application Programming Interface
CNN	Convolutional Neural Networks
ResNet	Residual Networks
MSE	Mean Squared Error

B Datasets

www.aviso.bz	aupair.com	flood.com	skullbliss.com	bio-link.info	onebestporno.com
designil.com	blauergockel.de	amara.org	xvideos.casa	moneyrates.com	athealth.com
oteglobe.gr	apotekasrbotrade.rs	umanaidoomd.com	bajabikes.eu	devoxx.be	20govtjobs.com
secure-i.jp	pdscourses.com	asianbride.me	fffuel.co	mercatometropolitano.com	fireballwhisky.com
foreign-girlfriend.net	olcha.uz	100daysofcode.com	toposla.com	soomfoods.com	guardian.net
dnsaktif.net	vttpi.com	biographics.org	immowelt.de	gdssecurity.com	ezoteriker.ru
wnp.pl	sportsmanfinder.com	bis.ru	nifa.org	parkster.com	hizlikargola.com
knowledgewalls.com	peaceoverviolence.org	beaumier.com	intipseleb.com	thekidneydietitian.org	tamilarasan.info
	vishalmegamart.com			najox.com	

Table B.1: Domain names used for forground URLs.

URLID	TRACEID	STARTTIMESTAMP	INCOMING_COUNT	OUTGOING_COUNT	TOTAL_COUNT	INCOMING_FRACTION	OUTGOING_FRACTION
4	0	1670587251.14552	172	80	252	0.682539682539683	0.317460317460317
4	1	1670587267.16763	237	116	353	0.671388101983003	0.328611898016997
4	2	1670587283.42696	297	121	418	0.710526315789474	0.289473684210526
4	3	1670587300.17824	217	85	302	0.718543046357616	0.281456953642384
4	4	1670587316.40383	263	109	372	0.706989247311828	0.293010752688172
4	5	1670587332.75219	337	146	483	0.697722567287785	0.302277432712215
4	6	1670587358.73652	267	123	390	0.684615384615385	0.315384615384615
4	7	1670587376.18	225	95	320	0.703125	0.296875
4	8	1670587391.96085	192	77	269	0.713754646840149	0.286245353159851

Table B.2: Sample of manual features used.

URLID	TRACEID	STARTTIMESTAMP	feature_0	feature_1	feature_2	feature_3	feature_4
4	0	1670587251	-1.1658091545105	-1.23471021652222	0.661990284919739	-0.39919438958168	0.141526818275452
4	1	1670587267	2.07344603538513	-0.664045810699463	0.812465965747833	-0.474332422018051	-0.471457898616791
4	2	1670587283	-0.386243373155594	-1.33498954772949	0.299132347106934	-0.494464039802551	-1.60033869743347
4	3	1670587300	0.491511046886444	0.836461186408997	-1.24342584609985	-1.54468607902527	-1.8343757390976
4	4	1670587316	1.17876410484314	-0.824996411800385	-1.0772465467453	-0.682961821556091	1.82796907424927
4	5	1670587332	-1.49562668800354	-1.11139917373657	1.1794958114624	-0.42986798286438	-1.0925145149231
4	6	1670587358	1.75452208518982	-2.27867722511292	-1.29787611961365	-0.480070918798447	1.05817592144012
4	7	1670587376	-0.768660604953766	0.140654385089874	-0.951414406299591	0.0581724643707275	0.413897305727005
4	8	1670587391	-2.00411534309387	0.290665626525879	-0.303337037563324	-1.43976426124573	0.682648956775665
4	9	1670587408	0.0290848258882761	1.01903867721558	0.477183043956757	-1.81922340393066	-0.93375039100647
4	10	1670587424	1.11843168735504	-0.972525715827942	1.86991930007935	1.31830191612244	0.2438775151968
4	11	1670587441	0.242990463972092	0.242134362459183	-0.95162832736969	0.608880162239075	-0.161616280674934

Table B.3: Sample of automatic features generated by VAE.

Bibliography

[1] D. E. E. 3rd. *Domain Name System Security Extensions*. RFC 2535. 03/1999.

[2] M. Abadi, A. Agarwal et al. *TensorFlow: Large-scale Machine Learning on Heterogeneous Systems*. `https://www.tensorflow.org/`. 2015.

[3] I. Affiah. *How to Implement K fold Cross-Validation in Scikit-Learn*. `https://www.section.io/engineering-education/how-to-implement-k-fold-cross-validation/`. 2022.

[4] L. von Ahn, B. Maurer et al. *reCAPTCHA: Human-Based Character Recognition via Web Security Measures*. `https://www.google.com/recaptcha`. 2007.

[5] A. Bajaj. *Performance Metrics in Machine Learning [Complete Guide]*. `https://neptune.ai/blog/performance-metrics-in-machine-learning-complete-guide`. 2023.

[6] D. Berrar. *Bayes' theorem and naive Bayes classifier*. In: *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics* (2018), pp. 403–412.

[7] P. BIONDI. *Scapy: Interactive Packet Manipulation Tool*. `https://scapy.net`. Version 2.4.5.

[8] S. Bosse, D. Maniry et al. *A Deep Neural Network for Image Quality Assessment*. In: *2016 IEEE International Conference on Image Processing (ICIP)*. 09/2016, pp. 3773–3777.

[9] R. Brown. *What are Features in Machine Learning and Why it is important?* `https://cogitotech.medium.com/what-are-features-in-machine-learning-and-why-it-is-important-e72f9905b54d`. Accessed: 2023-02.

[10] J. Brownlee. *A Gentle Introduction to the Rectified Linear Unit (ReLU)*. `https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/`. 2019.

[11] J. Brownlee. *How to Configure k-Fold Cross-Validation*. `https://machinelearningmastery.com/how-to-configure-k-fold-cross-validation/`. 2020.

[12] J. Brownlee. *A Gentle Introduction To Sigmoid Function*. `https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/`. 2021.

[13] Canonical. *Enterprise Open Source and Linux | Ubuntu*. `https://ubuntu.com/`.

[14] F. Chollet. *Deep Learning with Python*. Manning Publications, 2018.

[15] F. Chollet et al. *Keras*. `https://keras.io`. 2015.

[16] D.-A. Clevert, T. Unterthiner and S. Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. https://arxiv.org/abs/1511.07289v5. 2016.

[17] P. with Code. *Stochastic Gradient Descent*. https://paperswithcode.com/method/sgd. Accessed: 2023-02.

[18] G. Combs. *Wireshark – The world's most popular network protocol analyzer*. https://www.wireshark.org/. Accessed: 2023-03.

[19] E. Commission. *Guidelines on Automated individual decision-making and Profiling for the purposes of Regulation 2016/679*. https://ec.europa.eu/newsroom/document.cfm?doc_id=47742. 2017.

[20] E. Commission. *What is personal data?* https://commission.europa.eu/law/law-topic/data-protection/reform/what-personal-data_en. 2022.

[21] Deepchecks. *Learning Rate in Machine Learning*. https://deepchecks.com/glossary/learning-rate-in-machine-learning/. Accessed: 2023-03.

[22] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*. Tech. rep. RFC2246. RFC Editor, 01/1999, RFC2246.

[23] *Download the Fastest Firefox Ever*. https://www.mozilla.org/en-US/firefox/new/. Accessed: 2023-03. Mozilla.

[24] Z. Durumeric, Z. Ma et al. *The Security Impact of HTTPS Interception*. In: *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2017.

[25] P. S. Foundation. *argparse — Parser for command-line options, arguments and sub-commands — Python 3.11.2 documentation*. https://docs.python.org/3/library/argparse.html.

[26] P. S. Foundation. *concurrent.futures — Launching parallel tasks — Python 3.11.2 documentation*. https://docs.python.org/3/library/concurrent.futures.html.

[27] P. S. Foundation. *threading — Thread-based parallelism — Python 3.11.2 documentation*. https://docs.python.org/3/library/threading.html.

[28] *Geckodriver*. https://github.com/mozilla/geckodriver. Mozilla.

[29] I. Goodfellow, Y. Bengio and A. Courville. *Deep Learning*. Chapter 9. http://www.deeplearningbook.org. MIT Press, 2016.

[30] B. K. Gupta. *K-Nearest Neighbor (KNN) Algorithm in Machine Learning*. https://www.scaler.com/topics/knn-algorithm-in-machine-learning/. Accessed: 2023-02.

[31] B. Halley. *Dnspython: DNS Toolkit*. https://www.dnspython.org. Version 2.2.1.

[32] C. R. Harris, K. J. Millman et al. *Array Programming with NumPy*. In: *Nature* 585.7825 (09/2020), pp. 357–362.

[33] K. He, X. Zhang et al. *Deep Residual Learning for Image Recognition*. https://arxiv.org/abs/1512.03385. Accessed: 2015.

[34] *How the Web Works - Learn Web Development | MDN*. https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/How_the_Web_works. 03/2023.

[35] Z. Hu, L. Zhu et al. *Specification for DNS over Transport Layer Security (TLS)*. RFC 7858. 05/2016.

[36] J. D. Hunter. *Matplotlib: A 2D Graphics Environment*. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95.

[37] IANA. *Root Servers*. (Visited on 31/03/2023).

[38] I. M. Inc. *hCaptcha*. https://hcaptcha.com. Accessed: 2022-11.

[39] U. o. S. C. Information Sciences Institue. *Internet Protocol*. https://datatracker.ietf.org/doc/html/rfc791. Accessed: 2023-03.

[40] V. Jayaswal. *Laplace smoothing in Naïve Bayes algorithm*. https://towardsdatascience.com/laplace-smoothing-in-na%C3%AFve-bayes-algorithm-9c237a8bdece. 2020.

[41] Joblib Development Team. *Joblib: Running Python Functions as Pipeline Jobs*. https://joblib.readthedocs.io/. 2020.

[42] A. Kaul, S. Maheshwary and V. Pudi. *AutoLearn — Automated Feature Generation and Selection*. https://ieeexplore.ieee.org/document/8215494. 2017.

[43] W. Keeling. *Selenium Wire*. https://github.com/wkeeling/selenium-wire. Accessed: 2023-03.

[44] KimiNewt. *Pyshark: Python Wrapper for Tshark, Allowing Python Packet Parsing Using Wireshark Dissectors*. https://github.com/KimiNewt/pyshark. Version 0.5.3.

[45] D. P. Kingma and M. Welling. *Auto-Encoding Variational Bayes*. https://arxiv.org/abs/1312.6114. 2013.

[46] A. Kumar. *K-Fold Cross Validation – Python Example*. https://vitalflux.com/k-fold-cross-validation-python-example/. 2022.

[47] S. Kumar. *9 Reasons why Machine Learning models not perform well in production*. https://towardsdatascience.com/9-reasons-why-machine-learning-models-not-perform-well-in-production-4497d3e3e7a5. 2020.

[48] V. Le Pochat, T. Van Goethem et al. *Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation*. 2019.

[49] L. Li, K. Jamieson et al. *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization*. In: *Journal of Machine Learning Research* 18.185 (2018), pp. 1–52.

[50] MathWorks. *Feature Extraction*. https://www.mathworks.com/discovery/feature-extraction.html. 1994.

[51] P. Mockapetris and K. J. Dunlap. *Development of the Domain Name System*. In: ACM SIGCOMM Computer Communication Review 18.4 (08/1988), pp. 123–133.

[52] Mozilla. *HTTP response status codes*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#server_error_responses. Accessed: 2023-03.

[53] A. Nair. *Standardization vs Normalization*. https://towardsdatascience.com/standardization-vs-normalization-dc81f23085e3. 2022.

[54] L. Olejnik. *Web browsing histories are private personal data*. https://blog.lukaszolejnik.com/web-browsing-histories-are-private-personal-data-now-what/. 2020.

[55] A. Panchenko, F. Lanze et al. *Website Fingerprinting at Internet Scale*. In: *Network and Distributed System Security Symposium*. 2016.

[56] R. Parmar. *Common Loss functions in machine learning*. https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23. 2018.

[57] H. Patel. *What is Feature Engineering — Importance, Tools and Techniques for Machine Learning*. https://towardsdatascience.com/what-is-feature-engineering-importance-tools-and-techniques-for-machine-learning-2080b0269f10. 2021.

[58] A. Petrov. *Urllib3: HTTP Library with Thread-Safe Connection Pooling, File Post, and More*. https://urllib3.readthedocs.io/. Version 1.26.13.

[59] Python Software Foundation. *Python 3.9.16 Documentation*. https://docs.python.org/3.9/index.html. Accessed: 2023-03.

[60] L. Richardson. *Beautifulsoup4: Screen-scraping Library*. https://www.crummy.com/software/BeautifulSoup/bs4/. Version 4.11.1.

[61] V. Rimmer, D. Preuveneers et al. *Automated Website Fingerprinting through Deep Learning*. https://arxiv.org/abs/1708.06376. 2017.

[62] D. Sarkar. *Continuous Numeric Data*. https://towardsdatascience.com/understanding-feature-engineering-part-1-continuous-numeric-data-da4e47099a7b. 2018.

[63] *Selenium*. https://www.selenium.dev. Version 4.6.1.

[64] A. Sharma. *Variational Autoencoder in TensorFlow*. https://learnopencv.com/variational-autoencoder-in-tensorflow/. 2021.

[65] Shriram. *Multinomial Naive Bayes Explained: Function, Advantages & Disadvantages, Applications in 2023*. https://www.upgrad.com/blog/multinomial-naive-bayes-explained. 2022.

[66] S. Siby, M. Juarez et al. *Encrypted DNS –> Privacy? A Traffic Analysis Perspective*. 2019. arXiv: 1906.09682 [cs.CR].

[67] M. Singh, M. Singh and S. Kaur. *Detecting bot-infected machines using DNS fingerprinting*. In: *Digital Investigation* 28 (2019), pp. 14–33.

[68] A. V. Sriniwasan. *Stochastic Gradient Descent – Clearly Explained*. https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31. 2019.

[69] systemd. *System and Service Manager*. https://github.com/systemd/systemd. Accessed: 2023-03.

[70] G. B. Team. *Convolutional Neural Network (CNN)*. https://www.tensorflow.org/tutorials/images/cnn. Accessed: 2023-01.

[71] G. B. Team. *Convolutional Variational Autoencoder*. https://www.tensorflow.org/tutorials/generative/cvae. Accessed 2023-02.

[72] G. B. Team. *Introduction to the Keras Tuner*. https://www.tensorflow.org/tutorials/keras/keras_tuner. Accessed 2023-01.

[73] G. B. Team. *scikit-learn. Machine learning in Python*. https://scikit-learn.org/stable/. Accessed 2023-02.

[74] G. B. Team. *sklearn.linear_model.SGDClassifier*. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html. Accessed 2023-02.

[75] G. B. Team. *sklearn.model_selection.GridSearchCV*. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. Accessed 2023-02.

[76] G. B. Team. *sklearn.naive_bayes.MultinomialNB*. https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html. Accessed 2023-02.

[77] G. B. Team. *sklearn.neighbors.KNeighborsClassifier*. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html. Accessed 2023-02.

[78] G. B. Team. *tf.keras.losses.CategoricalCrossentropy*. https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy. Accessed 2023-02.

[79] G. B. Team. *tf.keras.losses.KLDivergence*. https://www.tensorflow.org/api_docs/python/tf/keras/losses/KLDivergence. Accessed 2023-02.

[80] G. B. Team. *Create production-grade machine learning models with TensorFlow*. https://learnopencv.com/variational-autoencoder-in-tensorflow/. 2019.

[81] T. P. D. Team. *Pandas-Dev/Pandas: Pandas*. https://doi.org/10.5281/zenodo.7344967. Version v1.5.2. Zenodo, 11/2022.

[82] V. Tiwari, C. Pandey et al. *Image Classification Using Deep Neural Network*. In: *2020 2nd International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. 2020, pp. 730–733.

[83] T. Verdonck, B. Baesens et al. *AutoLearn — Automated Feature Generation and Selection*. https://ieeexplore.ieee.org/document/8215494. 2021.

[84] Y. Verman. *Why Data Scaling is important in Machine Learning & How to effectively do it*. https://analyticsindiamag.com/why-data-scaling-is-important-in-machine-learning-how-to-effectively-do-it/. 2021.

[85] L. Wuhan Jiyi Network Technology Co. *Geetest CAPTCHA*. `https://hcaptcha.com`. Accessed: 2022-11.