BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PHISHING WEBPAGE DETECTION USING MACHINE LEARNING METHODS

DETEKCE PHISHINGOVÝCH STRÁNEK POMOCÍ METOD STROJOVÉHO UČENÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **PETER POLÓNI**
AUTOR PRÁCE

SUPERVISOR **Ing. RADEK HRANICKÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

153621

Institut: Department of Information Systems (DIFS)
Student: **Polóni Peter**
Programme: Information Technology
Title: **Phishing Webpage Detection using Machine Learning Methods**
Category: Security
Academic year: 2023/24

Assignment:

1. Learn how to use machine learning for the classification of web pages and their content. Study the characteristics used for phishing detection.
2. In consultation with your supervisor, create a dataset (e.g., page contents, domain information, data from external sources, etc.) for a suitably selected set of phishing and trusted sites.
3. Create a classifier for detecting phishing websites.
4. Using the created dataset, experimentally validate the usability of your solution. Assess the results using standard metrics.
5. Evaluate and discuss the results.

Literature:

- Abu-Nimeh, Saeed, et al. "A comparison of machine learning techniques for phishing detection." *Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit*. 2007.
- Xiang, Guang, et al. "Cantina+ a feature-rich machine learning framework for detecting phishing web sites." *ACM Transactions on Information and System Security (TISSEC)* 14.2 (2011), pp. 1-28.
- Almseidin, Mohammad, et al. "Phishing detection based on machine learning and feature selection methods." *International Journal of Interactive Mobile Technologies.* 13.171 (2019), pp.171-183, doi:10.3991/ijim.v13i12.11411.
- Sahingoz, Ozgur Koray, et al. "Machine learning based phishing detection from URLs." *Expert Systems with Applications* 117 (2019), pp. 345-357.

Requirements for the semestral defence:
Points 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Hranický Radek, Ing., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: 1.11.2023
Submission deadline: 9.5.2024
Approval date: 30.10.2023

Abstract

Phishing web pages are a very dangerous threat, which means that successful and reliable detection of these pages is essential. I detect these threats by utilizing a machine learning based approach. This approach is effective and can detect even threats it has never encountered. As credible sources of URLs, I used sources like OpenPhish and PhishTank. I gathered the HTML and JavaScript code of web pages from the trusted URLs by utilizing a data-gathering program that I created. Using the feature vector composed of 82 numerical features, I created four classifiers. Then, I tuned and experimentally tested the performance of these classifiers. The best-performing model is the XGBoost classifier, which achieved a balanced accuracy score of 97.03% and a false positive rate of 2.22% while making predictions on previously unseen data. Results show that this detection approach can identify phishing web pages even in a non-training environment, which I verified by implementing a phishing-detecting web extension for the Chrome browser. Implementing this extension is beyond the scope of the assignment of this thesis.

Abstrakt

Phishingové stránky sú veľmi nebezpečnou hrozbou, čo znamená, že úspešná a spoľahlivá detekcia týchto stránok je veľmi doležitá. Tieto hrozby detekujem s využitím prístupu strojového učenia. Tento prístup je efektívny a dokáže odhaliť aj hrozby, s ktorými sa nikdy predtým nestretol. Ako dôveryhodné zdroje dát URL som využil OpenPhish a PhishTank. Z dôveryhodných URL som nazbieral HTML a JavaScript kód webových stránok. Zber dát som vykonal pomocou programu, ktorý som pre tento účel vytvoril. S využitím vektoru príznakov, ktorý sa skladá z 82 numerických príznakov, som vytvoril štyri klasifikátory. Následne som ich vyladil a experimentálne overil presnosť ich predikcií. Najpresnejší model je XGBoost klasifikátor, ktorý dosiahol vyváženú presnosť až 97.03% a FPR 2.22%, počas predikovania dát, ktoré nikdy predtým nevidel. Výsledky ukazujú, že tento prístup detekcie je schopný identifikovať phishingovú stránku aj v praxi. Toto som overil aj implementovaním webového rozšírenia pre prehliadač Chrome, ktoré detekuje phishigové stránky. Toto rozšírenie je vytvorené nad rámec zadania.

Keywords

HTML, JavaScript, dataset, gathering data, machine learning, phishing detection

Klíčová slova

HTML, JavaScript, dátová sada, zber dát, strojové učenie, detekcia phishingu

Reference

POLÓNI, Peter. *Phishing Webpage Detection using Machine Learning Methods.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Hranický, Ph.D.

Rozšířený abstrakt

Práca sa zaoberá detekciou phishingových webových stránok pomocou využitia metód strojového učenia. V práci sa phishingové webové stránky odhaľujú pomocou analýzy ich HTML a JavaScript kódu a následným vyhodnotením vlastností analyzovanej stránky pomocou vytrénovaného klasifikátoru. Pred samotným tréningom klasifikátoru je nutné získať dáta nutné pre tréning klasifikátoru. Pre účel zberu tréningových dát je v práci implementovaná scrapingová aplikácia, ktorá funguje na princípe vysielania asynchronných požiadaviek na webové stránky, ktorých HTML a JavaScript kód chce získať. Využitie asynchronných požiadaviek má výhodu v tom, že kým jeden pracovník čaká na odpoveď odoslanej požiadavky, program dovoľuje ostatným pracovníkom pracovať ďalej. Samotný zber prebieha v troch fázach.

V prvej fáze aplikácia na zber dát dostáva ako vstup csv súbor, ktorý obsahuje zoznam URL, ktorých kód bude získavať. Ďalej obsahuje informáciu o tom či je daná URL phishingová alebo nie. Tieto informácie aplikácia spracuje a pokračuje druhou fázou. V tejto fáze aplikácia posiela spomínané asynchrónne požiadavky na obdržané URL. Ako odpoveď obdrží HTML kód stránky, ktorý sa uloží do databázy. Zo získaného HTML sa následne vytiahnu všetky odkazy na využívané externé JavaScript kódy. V tretej fáze sa následne pomocou týchto odkazov získa JavaScript kód využitý v HTML a uloží sa do databázy.

Práca sa zaoberá aj analýzou nazbieraných dát, ktorá odhaľuje potencionálne rozdiely medzi kódom phishingových a legitímnych stránok ako je napríklad priemerný počet znakov v HTML kóde. Po analýze dát sa zo získaného kódu následne extrahuje 82 numerických príznakov, ktoré popisujú rôzne vlastnosti nazbieraného kódu. Extrakciou príznakov vzniká dátová sada o veľkosti 31481 nazbieraných stránok, ktorá obsahuje extrahované príznaky. Táto dátová sada predstavuje tréningové dáta. V práci je táto sada využitá na tvorbu štyroch binárnych klasifikátorov vytvorených pomocou algoritmov XGBoost, LightGBM, SVM a Neurónových sietí. Každý vytvorený klasifikátor prešiel procesom ladenia hyperparametrov, ktorý zaistil čo najlepšiu úspešnosť týchto klasifikátorov. Následne boli tieto klasifikátory experimentálne overené na tréningových a aj na predtým nikdy nevidených dátach.

Tieto experimenty porovnávali úspešnosť predikcií jednotlivých klasifikátorov a zistili, že najúspešnejšie predikcie na predtým nikdy nevidených dátach, vykonáva klasifikátor vytvorený pomocou algoritmu XGBoost a to s vyváženou presnosťou 97.03%. Avšak všetky klasifikátory dokázali pri predikovaní predtým nevidených dát udržať svoju úspešnosť nad 90%, čo indikuje, že odhaľovanie phishingových stránok pomocou analýzy HTML a JavaScript kódu je pomerne úspešná taktika boja proti nikdy predtým nevideným phishingovým útokom. Ďalej sa v práci experimentálne overujú spôsoby vylepšenia úspešnosti klasifikátorov ako je napríklad ladenie prahovej hodnoty, ktorá určuje či je výsledkom predikcie phishingová alebo legitímna stránka. Overuje sa aj benefit princípu väčšinového hlasovania, ktorý kombinuje viacero klasifikátorov, ktoré hlasujú o výsledku predikcie.

Po experimentálnom overení klasifikátorov sa v práci predstavuje praktické využitie týchto modelov. V práci je nad rámec zadania implementované aj webové rozšírenie, ktoré dokáže odhaliť, či sa užívateľ nachádza na stránke, ktorá je podozrivá z phishingu a upozorní ho na túto skutočnosť. Toto webové rozšírenie po obdržaní príkazu na začatie činnosti získa URL stránky, na ktorej sa užívateľ nachádza, získa HTML a JavaScript kód tejto stránky a extrahuje z neho príznaky, ktoré zadá klasifikátoru a ten následne rozhodne, či sa jedná o phishingovú stránku alebo nie. Webové rozšírenie o tom informuje užívateľa pomocou notifikácie. Toto rozšírenie dokáže pracovať aj v režime väčšinového hlasovania, ak si ho užívateľ zvolí.

Phishing Webpage Detection using Machine Learning Methods

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Radek Hranický, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Peter Polóni
9.5.2024

Acknowledgements

I want to express gratitude to my supervisor Ing. Radek Hranický, Ph.D. for his guidance and friendly approach while creating this thesis.

Contents

List of Figures

Chapter 1

Introduction

In our time, the Internet has become man's second nature. People communicate, do their shopping, and even pay their bills via the Internet. However, while the Internet brings many favorable aspects to our lives, it also brings many problems, such as malicious websites. Many techniques, such as user education, were developed to prevent the damage these sites cause on the Internet. Despite these efforts, malicious content, specifically phishing, continues to expand on the web. To crack this problem, it is essential to be able to determine phishing web pages and benign web pages successfully. However, phishing detection techniques based on blocklists or heuristics are not able to provide protection against brand-new threats, which makes them insufficient.

On the contrary, approaches based on machine learning are capable of detecting even unfamiliar attacks. I aim to detect phishing web pages by utilizing machine learning. To successfully detect these web pages, it is required to gather data of both benign and phishing web pages. The data used to differentiate between these pages includes HTML and JavaScript code that assembles these pages. To gather such data, I will start by creating a web scraping application. The obtained data will be then analyzed. After the analysis, I will propose a feature vector formed by features that will examine multiple factors of the gathered HTML and JavaScript code. Following the establishment of the feature vector, I will extract features from the scraped data and store these features in the database. The extracted features will form a dataset that will allow me to utilize machine learning algorithms and create classification models capable of detecting a phishing web page. After assembling these classifiers, I will fine-tune these models and experimentally verify their performance on both training and unseen data, which will allow me to determine what classifiers are most accurate and reliable. The best-performing classifier, which achieved a balanced accuracy score of 97.03% while making predictions on previously unseen data, will be utilized in the proof-of-concept web extension capable of detecting phishing web pages. Implementation of this web extension is beyond the scope of the assignment of this thesis, and it will demonstrate that detection based on the machine learning approach also works in practice.

This bachelor's thesis begins with Chapter 2, where I introduce several approaches to phishing detection with their applicable use, compare their resemblances, and talk about their individual and shared shortcomings. Chapter 3 then introduces and explains various machine learning algorithms that are being used for classification purposes. Chapter 4 then presents the significance of trustworthy sources for data collecting. It describes in detail how the program that carries out data gathering works and what techniques are used to solve the problems with scraping. This chapter discusses technologies utilized for storing

data and filtering the scraped data. Chapter 5 then provides an analysis of the scraped data. Chapter 6 introduces a feature vector and describes how feature extraction is carried out in the program. It also explains why certain features were included and how they should benefit the classifiers. The following Chapter 7 explains the nature of the created dataset and analyzes features included in the dataset. Chapter 8 introduces evaluation metrics and data preprocessing techniques utilized during the training of various classifiers. This chapter also explains how model training and fine-tuning are carried out in the program and compares the results of tuned and untuned models. Chapter 9 provides the details of experiments that aim to potentially improve the tuned classifiers by employing various strategies or evaluate the performance of tuned classifiers on both training and unseen data. Finally, Chapter 10 explains the implementation of the web extension that utilizes the best-performing classifiers to detect phishing web pages. This chapter also verifies the ability of classifiers to make predictions in practice by utilizing them to make predictions in real-time in the real environment.

Chapter 2

Approaches to phishing detection

There are diverse techniques available for purposes related to the detection of phishing. The earliest techniques relied on non-machine learning approaches such as sandboxing [25]. This chapter discusses several of these phishing detection methods, their advantages and disadvantages, and it mentions studies that employ these strategies. It also provides a comparison between these approaches.

2.1 User awareness

Khonji [19] states that phishing attacks are often aimed at individuals who lack experience identifying them. Enlightening these users is crucial for reducing their vulnerability to these attacks. Various user-enlightening strategies have been suggested over the years to make people aware of various social engineering techniques. One such strategy is to educate users through regular messages like emails. For example, e-services employ this strategy, these service providers often send messages to their clients to alert them about potential phishing scams.

A different strategy utilizes user interfaces to display safety warnings when a user attempts to access a phishing website. These warnings are especially adequate when they block the displayed data [13]. Another applied approach is incorporating educational notices into the end-user's daily activities [20]. These methods can help reduce the probability of users falling victim to a phishing attack. However, it is necessary to note that educating end-users alone may not be enough to alter their behavior [19].

2.2 Blacklists

Blacklists utilized for phishing detection are regularly updated lists of URLs that have already been detected as phishing attempts. Google Safe Browsing API is an example of such blacklist. However, blacklists are not effective against zero-hour attacks. Zero-hour attacks are attacks that were not seen before, blacklists are not effective against them because the web page needs to be already identified as a phishing attempt to be added to the blacklist. [19]

According to Sheng [42], blacklists were only able to detect 20% of zero-hour phishing attacks. Sheng also found that up to 83% of phishing URLs were added to blacklists after 12 hours, which is a problem since 63% of phishing campaigns end in the first 2 hours.

2.3 Heuristics

Phishing heuristics are characteristics that are present in phishing attacks. These mechanisms can be based on finding the source IP address of the attacker or analyzing the content of emails or web pages. However, it is essential to mention that these characteristics may not always be present in phishing attacks. Nevertheless, recognizing a set of general heuristic tests makes it possible to detect zero-hour attacks. But, using generalized heuristics carries a risk of incorrectly classifying legitimate content. [19]

2.4 Visual similarity

These phishing detection methods rely on identifying phishing attacks through visual appearance instead of analyzing the source code or network-level data [19]. Visual similarity detection includes methods that need the web browser to take a picture of every suspected site they try to investigate. The picture is matched against a whitelist composed of legitimate websites commonly targeted by phishers [9]. On the other hand, some visual methods do not need the whitelist of legitimate websites. These methods are based on the fact that most phishing websites aim to match their target website visually [15].

2.5 Machine learning

Janiesch [18] states that machine learning automatically tries to learn patterns and relationships from given data. It aims to automate the creation of an analytical model capable of executing cognitive objectives, such as natural language translation. This is accomplished by utilizing algorithms that learn from training data, allowing these algorithms to uncover hidden understandings and challenging patterns presented in the given data. Machine learning is a technique that can be applied to tasks that involve high-dimensional data. Examples of these tasks are clustering, regression, or classification.

Machine learning technologies are very valuable, particularly when processing enormous amounts of data. These technologies offer significant value by saving time and maximizing computing resources [49]. Many machine learning algorithms are available in the field, each with numerous variations depending on the learning task. Machine learning algorithms are now commonly used in different industries, including fraud detection [18].

For example, McGahagan's work described in [31] used the examination of JavaScript and HTML code to create a model for detecting phishing websites. He carefully picked the 26 most powerful features that determine if a site is malicious. These features are shown in the Figure 2.1. This work also included creating a dataset. The dataset was formed by extracting features from 34778 benign and 5931 malicious websites. McGahagan used the package PySelenium[1] to scrape these web pages. However, this technique of scraping can be very time-consuming. Another problem is that the benign part of the dataset utilized in this research was taken from top-rated websites in Alexa ranking[2], but this alone does not guarantee that these websites are benign. It only means they are frequently visited. Overall, this research achieved slightly better results than previous researchers, using only half the features.

[1] https://selenium-python.readthedocs.io/

[2] https://www.digitaltrends.com/business/what-is-alexa-rank-everything-you-need-to-know/

Hou [17] presented a similar idea, suggesting using additional document level features from HTML and JavaScript, as they are easy to extract and preprocess. Malicious code is often camouflaged within the HTML by encryption, making features such as document length and average word length highly effective. The encrypted or obfuscated code in HTML is also connected to JavaScript features such as functions eval or unescape, as they may indicate the execution of encrypted code inside the HTML.

Identified Features Ranked			
Feature	*No*	*Over*	*Under*
Total tag count	1: 0.3206	1: 0.2705	1: 0.2239
Total href attributes	2: 0.1025	2: 0.1190	2: 0.1723
<link href> OoD	3: 0.0644	3: 0.0943	3: 0.1018
<p> count	4: 0.0567	5: 0.0601	4: 0.0642
<a href="https*">	5: 0.0554	8: 0.0403	6: 0.0581
<meta> count	6: 0.0515	6: 0.0471	8: 0.0340
<script_async=true>	7: 0.0462	7: 0.045	5: 0.0634
<link type="text/css">	8: 0.0298	9: 0.0327	11: 0.0257
<script src> OoD	9: 0.0289	14: 0.0141	7: 0.0535
<link href="http*">	10: 0.0271	11: 0.0224	10: 0.0283
push()	11: 0.0258	4: 0.0627	9: 0.0325
<link href="*.css">	12: 0.0258	12: 0.0205	13: 0.0125
indexOf()	13: 0.0175	25: 0.0071	16: 0.0119
<form action="http*">	14: 0.0168	19: 0.012	12: 0.0136
count	15: 0.0151	15: 0.0132	18: 0.0114
<iframe src="https*">	16: 0.015	10: 0.0271	24: 0.0078
count	17: 0.0141	16: 0.0131	19: 0.0093
setTimeout()	18: 0.0136	26: 0.0066	15: 0.0121
<a href="*.com">	19: 0.0133	13: 0.0186	20: 0.0090
document.write()	20: 0.0112	17: 0.0129	22: 0.0084
addEventListener()	21: 0.0096	20: 0.011	14: 0.0124
get()	22: 0.0093	21: 0.0107	26: 0.0023
<link type= "application/rsd+xml" >	23: 0.0079	22: 0.0103	21: 0.0088
find()	24: 0.0077	24: 0.0078	25: 0.0035
<link rel="shortlink">	25: 0.0073	23: 0.0085	23: 0.0080
replace()	26: 0.0069	18: 0.0123	17: 0.0114

Figure 2.1: McGahagan's list of 26 features [31]

2.5.1 Supervised machine learning

As described in [51], supervised learning is a technique for learning a function from a set of given training data. This data includes input objects and their expected outputs. The function can either output a continuous value, known as regression, or predict a class label for the input, known as classification. A supervised learner is supposed to predict the output for any valid input object.

2.5.2 Unsupervised machine learning

Unsupervised machine learning is a type of learning where the algorithm does not use any manual labels of inputs. It uses no human examples to learn [51].

2.6 Deep learning

Guo [14] defines deep learning as a subfield of machine learning that aims to comprehend high-level abstractions in data. Deep learning can also be explained as an advanced form of artificial neural network known as a deep neural network [49]. There are two typical network structures in deep learning: CNNs[3] and RNNs[4]. Nowadays, Convolutional Neural Networks are widely used in many fields of machine learning, particularly in computer vision. On the other hand, Recurrent Neural Networks are primarily utilized in processing time series data, such as natural language processing or speech recognition [49].

Deep learning is also used to detect phishing attacks, as it can handle visual and text data. As described in [34], Naim focused in his work on visual and non-visual elements that a web page includes. He observed HTML code and hierarchies, JavaScript, CSS, color tables, styles, font types, and objects. In addition, Naim also observed the actual appearance of the website once its content was loaded and generated. Thanks to this, he created a hybrid technique that enhances the static analysis technique with aspects of dynamic examination. His dataset contained 35,707 website records. However, only 697 of these were malicious, as he wanted his dataset to reflect the actual probability of a web page being malicious. His approach was able to effectively detect more than 83% of malicious websites while maintaining a low false positives ratio of 2%. It is essential to mention that although the feature vector of the model being used in this research takes into account various properties, it fails to consider the document level properties of the HTML files. Additionally, the dataset used in Naim's research is highly imbalanced, which means that the created model may need more data to detect malicious websites more consistently.

[3] https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/
[4] https://www.simplilearn.com/tutorials/deep-learning-tutorial/rnn

Chapter 3

Machine learning algorithms

This chapter introduces and explains the principles of popular machine learning algorithms. Although machine learning is a broad field that includes many disciplines, in this thesis, I focus only on differentiating between two classes, which is carried out by binary classification. Thus, algorithms introduced in this chapter are popular choices in this field.

3.1 Decision trees

A decision tree is a type of supervised machine learning algorithm that can be used for classification or regression tasks based on how previously raised questions were answered while using a tree-like pattern of decisions [41, 46]. The foundation of a decision tree is called the root node, which symbolizes the whole dataset. As shown in Figure 3.1, the root node has a series of decision nodes growing out, representing the dataset's features [46]. Each decision node represents a raised question, and these decision nodes either sprout leaf nodes that embody possible answers to the questions raised by the decision nodes or they sprout another decision node, which raises another question [41, 46]. On the other hand, leaf nodes do not contain any further branches, as each leaf node represents one of the classes being classified [46].

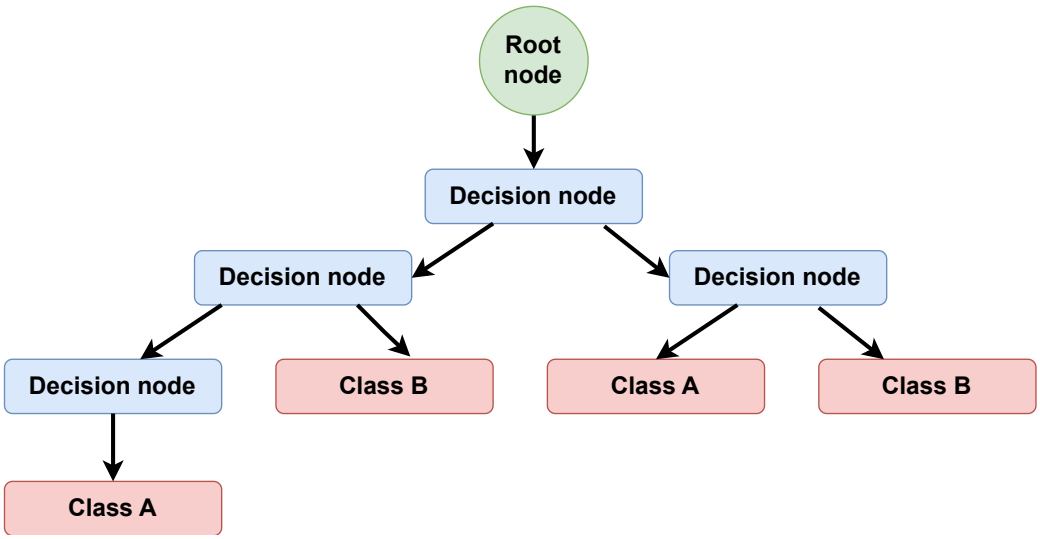

Figure 3.1: Example of a decision tree

3.2 Support vector machine

The support vector machine is a supervised machine learning algorithm utilized to solve regression and classification assignments. This algorithm is particularly effective in binary classification, in which data are split into two classes. The goal of a support vector machine algorithm is to find the best possible decision boundary, also known as a hyperplane, which splits the data points of the different classes. Support vector machines operate by converting the given data into a higher-dimensional space. This transformation is accomplished by using a kernel function, a mathematical function utilized to calculate the product between two data points in the transformed feature space. In the training phase, support vector machines identify the best hyperplane in a higher-dimensional feature space by utilizing a mathematical formula. Identifying the best hyperplane is crucial as it helps maximize the margin between data points of different classes while also minimizing misclassification. Figure 3.2 shows that margin is the gap between the decision boundary and the support vectors. [44]

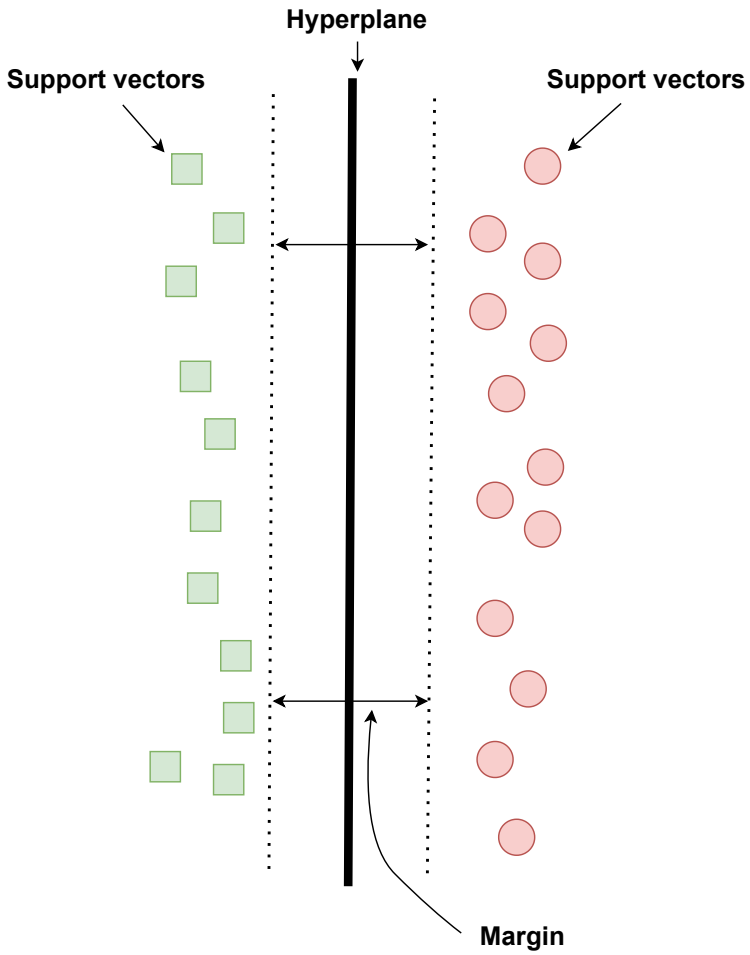

Figure 3.2: SVM algorithm.

3.3 Random forest

A random forest is a supervised machine learning algorithm utilized for both regression and classification tasks. It employs an ensemble learning approach that uses multiple classifiers to provide solutions [30]. The algorithm employs bagging to construct full decision trees parallelly from random bootstrap samples of the input data [35]. The output of a random forest algorithm is obtained by taking the average of the predictions made by the generated trees [30]. Figure 3.3 displays how random forest works in detail.

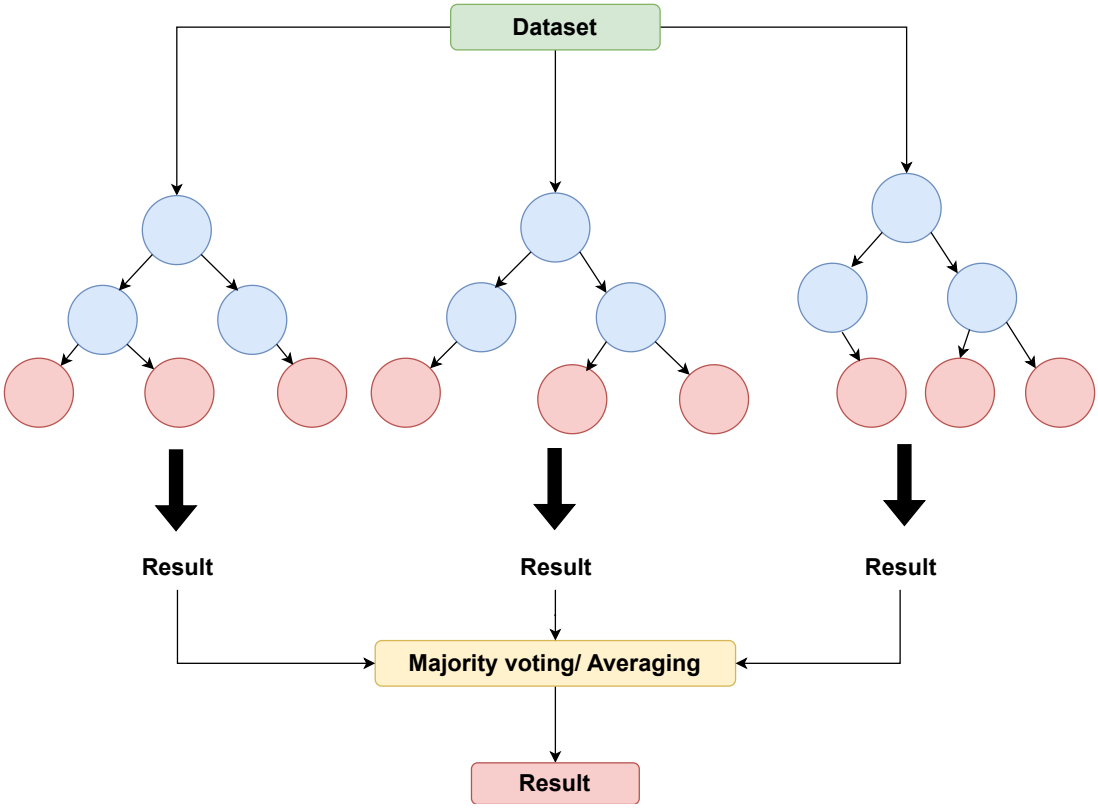

Figure 3.3: Random forest algorithm.

3.4 XGBoost

Algorithm XGBoost is an implementation of gradient-boosted decision trees [5]. Unlike bagging methods such as random forest, it uses trees with fewer splits instead of growing them to their full extent [48].

A gradient boosting ensemble method involves these steps. First, a model is defined to predict the desired variable. The first model will then be associated with a residual. This model is displayed in Figure 3.4 as M. Second, a newly created model fits the residuals from the previously trained model in the first step, in Figure 3.4 represented by K. Lastly, models from the first and second steps are combined to create a new model, shown in Figure 3.4 as B. This new model is a boosted version of the model from the first step. The mean squared error from the boosted model will be lower than that from the original model. This process can be repeated for multiple iterations until residuals have been minimized as much as possible. [48]

$$B(variable) = M(variable) + K(variable)$$

Figure 3.4: Representation of gradient boosting

3.5 AdaBoost

The Adaptive Boosting algorithm is an ensemble boosting technique. In the beginning, this algorithm builds a model and assigns equal weights to the data points. AdaBoost then identifies the data points that are wrongly classified and assigns higher weights to these data points. In the next model, the algorithm gives more importance to the data points with higher weights. This process is repeated multiple times until the algorithm achieves a lower error. The process of weight assigning is displayed in Figure 3.5. [39]

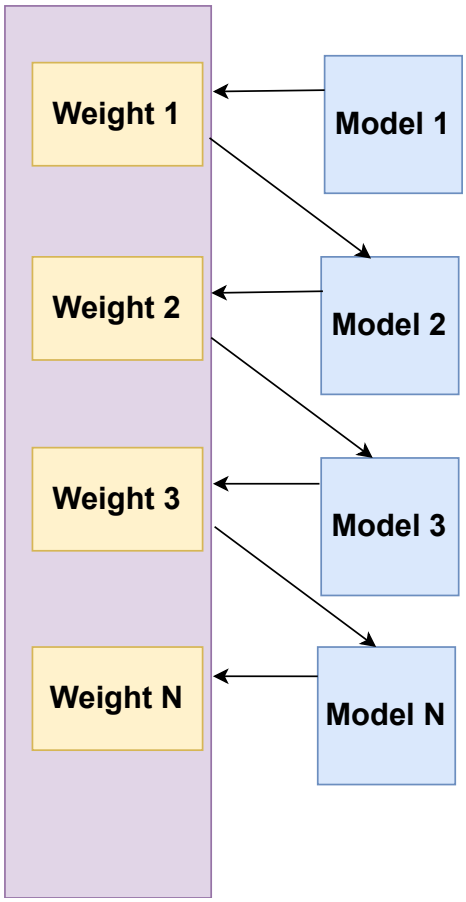

Figure 3.5: AdaBoost weight assigning.

3.6 LightGBM

LightGBM is a machine learning algorithm that implements gradient-boosted decision trees. Similarly to the algorithm, XGboost mentioned in Section 3.4, decision trees are combined

in a manner that every new model fits the residuals from the previous one, resulting in an improved model. The final model is created by aggregating the results from each step. The LightGBM algorithm searches for the best split of data instances that maximizes the information gained from each split. Information gain is the difference between entropy before and after the split, where entropy is a measure of randomness. The splits are performed in a manner that minimizes the randomness. [50]

On top of that, Light GBM produces the decision trees leaf-wise while other algorithms, for example, XGBoost, grow these trees level-wise. The LightGBM algorithm will choose the leaf with the max delta loss to grow. The comparison of these growing techniques is displayed in Figure 3.6. [29]

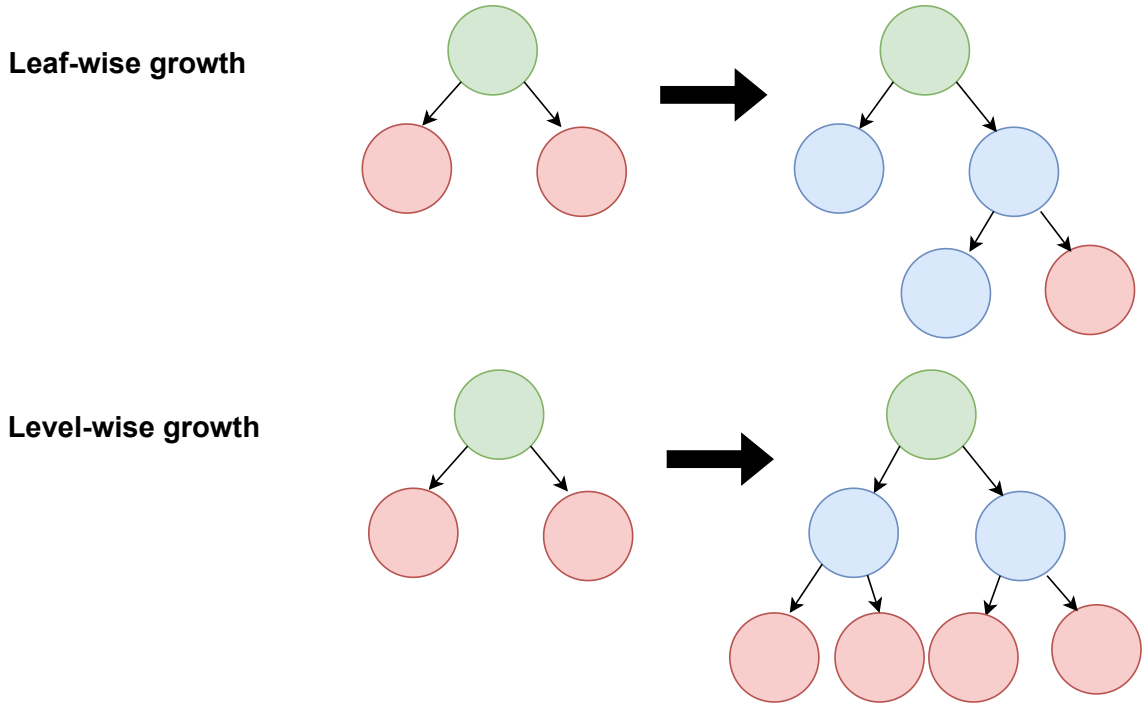

Figure 3.6: Leaf-wise growth and level-wise growth.

3.7 Neural Networks

A neural network is a machine learning algorithm that mimics the function of the human brain. The network consists of interconnected nodes that are known as artificial neurons. The neurons are arranged in layers. Every neuron receives input, performs calculations, and passes the output to the next layer. Connections between nodes in a network get stronger or weaker based on patterns in data, which enables the network to learn and make decisions. The first layer of a neural network is called the input layer. This layer converts received data into a format that the rest of the network is capable of processing. On the other hand, the last layer of the neural network is referred to as the output layer, and the number of outputs is determined by the creator of the network and its intended purpose. Hidden layers between input and output layers perform non-linear transformations to extract higher-level features from recieved data. As shown in Figure 3.7, every single neuron in the hidden layer gets input data from neurons located in the previous layer. Then, the hidden layer neuron

assigns weights and biases to these inputs and hands the result to a non-linear activation function. This is repeated until the output layer is reached. [40]

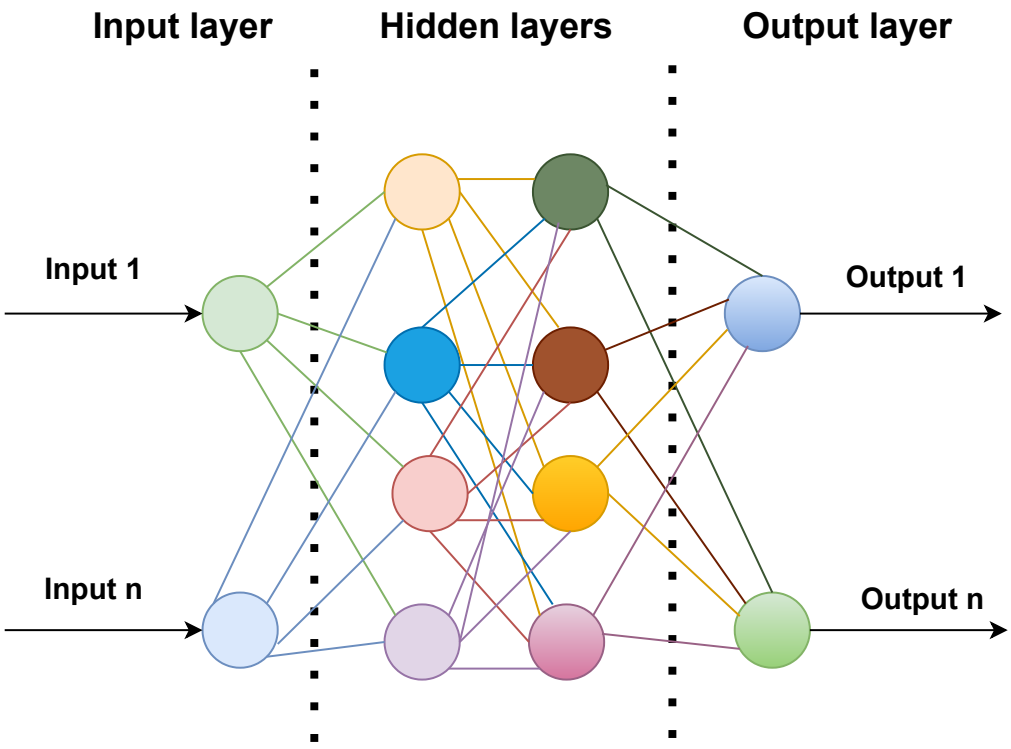

Figure 3.7: Example of an architecture of Neural Network.

Chapter 4

Collecting and storing data

This chapter introduces the challenges of collecting data and establishes their solutions. Moreover, it also explains the gathering of data in code and mentions the importance of reliable sources. It presents technologies chosen for storing data. Additionally, it provides information about filtering the stored data.

4.1 Data sources

Data gathering is a crucial aspect of this program, as the results of machine learning are only as reliable as the dataset used for training. Therefore, it is essential to use only trustworthy sources of data. The trusted data sources, in this case, are OpenPhish[1], PhishTank[2], and a benign training dataset from [11]. These sources can be considered trustworthy due to their licensing or reputation. Organizations OpenPhish and PhishTank provide phishing URLs that they verify, and thanks to this, the dataset constructed by data collected from these URLs should be of great quality. However, it is essential to filter the data collected from these sources. For example, OpenPhish feeds contain URLs that are unique, but it is common for more URLs to refer to the same phishing page. Getting rid of these duplicates ensures a good variety of training data rather than just a large quantity of data. Also, among the URLs from the benign dataset, many URLs are dead and return error messages that need to be removed as well.

4.2 Collecting data as a part of this program

As shown in Figure 4.1, the whole program consists of four Python modules and a database. The schema shows all of the modules and their interactions. However, the parts necessary for data gathering are modules *downloader.py*, *parse.py*, and database. The program takes two inputs to initiate data gathering. These inputs are a csv file that contains URLs and the name of a MongoDB collection, where data will be saved. The module *parse.py* extracts the information needed from the received csv file, and the module *downloader.py* then scrapes and saves the obtained data in the database.

[1] https://openphish.com/index.html
[2] https://phishtank.org/

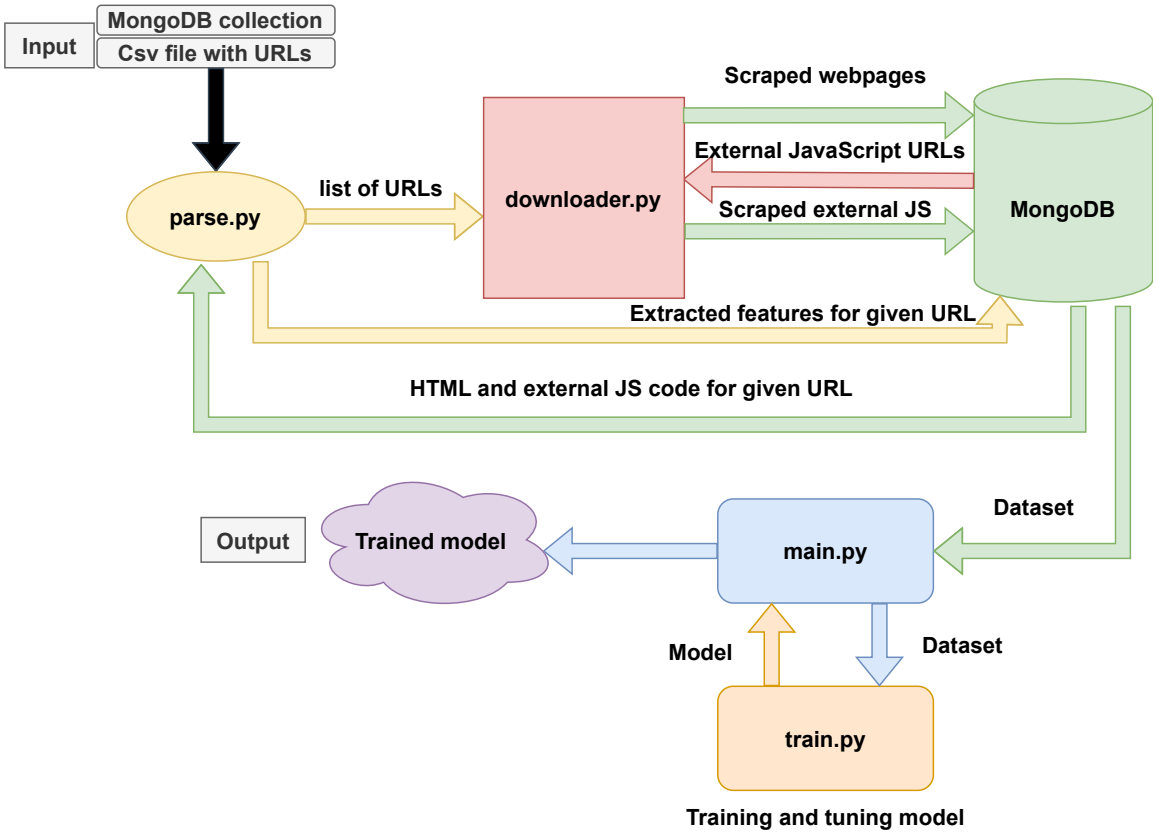

Figure 4.1: Program schema.

4.3 Command line arguments

The program supports three groups of arguments. First of all, the arguments that control the scraping process. These arguments include *scrape* and *collection*. The argument *scrape* has to be followed by the name of a csv file, which contains URLs to be scraped. This argument initiates the process of gathering and storing data. The argument *scrape* has to be paired with argument *collection* that specifies in what collection in the database the scraped data will be saved. Second of all, the arguments that train and save classifiers such as *trainLGB*, *trainXGB*, *trainNN*, and *trainSVM* these arguments need to be followed by the name of a file in which the trained classifier will be saved. Third of all, the arguments that start the process of tuning classifiers *tuneLGB*, *tuneXGB*, *tuneNN*, and *tuneSVM*. Using these arguments does raise a flag that begins the process of tuning the models.

4.4 Parsing input file

The parsed csv file has to contain two columns. The first column has to contain a URL that will be scraped. The second column must contain information on whether the URL is benign or malicious. These pieces of information are then extracted with the help of Python's package *csv*[3] and its method *reader*. Thanks to this method, it is possible to iterate through the csv file like through rows in a table. After the data has been extracted,

[3] https://docs.python.org/3/library/csv.html

the URLs and class labels are inserted into a list. If the first column contains a domain instead of a full URL, the domain will be modified by adding `http://www.` at its beginning, and then the domain is inserted into the list. This process is carried out in module *parse.py* by method *parse*. Figure 4.1 shows how parsing of the input file interacts with the rest of the program.

4.5 Scraping

It is necessary to create a web scraper to collect the actual data, like the HTML and JavaScript codes of a web page. One of the fastest scraping methods is to send a request to a URL, and the response will include the HTML of the web page hosted on that URL. However, as straightforward as it sounds, there are some things that could be improved with this method. Many websites have implemented defensive measures against web scraping, so not all of the responses contain data that are useable for machine learning. Naturally, these responses must be removed from the dataset. Another problem is that scraping can be time-consuming, particularly when scraping thousands of web pages to create a dataset. It is crucial to minimize the time by utilizing a concurrent approach. The concurrent approach enables the program to carry out multiple requests at once, thus making it significantly more time-efficient.

4.5.1 Asyncio

Asyncio[4] is a package that enables writing a concurrent code. Asyncio's coroutines can be scheduled concurrently but do not always have to work concurrently. Therefore, Asyncio is not parallelism. Asyncio is more similar to threading than multiprocessing [43]. It provides many tools for controlling the program flow or concurrent workers. One of the most significant aspects of Asyncio is that it runs asynchronously. Individual coroutines can halt while waiting for results and allow other routines to run [43]. Thanks to this unique ability, Asyncio is commonly used for web scraping.

4.5.2 Scraping the HTML

To address the challenges introduced in Section 4.5, creating a concurrent scraper with controlled request sending is important to avoid triggering the defense against web scraping. To accomplish that, I used Python's Asyncio. Scraping of HTML is handled by the method *run* in module *downloader.py*. To scrape the HTML, it is essential to pass the list of URLs mentioned in Section 4.4 and a MongoDB collection where data will be stored as parameters to method *run*. This method also employs Asyncio's semaphore, which monitors the flow of requests during scraping. This semaphore will permit only 30 workers to send out requests simultaneously to prevent any overload. Despite these precautions, if any web page is unavailable, recognizes the request is from a scraping bot, or returns a code that is bigger or equal to 400, this response will not be stored in the database, as responses like these do not contain any HTML code but only error messages and error codes so they would only cause disturbance while model training. It is crucial to mention that at this stage of the program, I not only scrape the HTML code of the web page, but I also isolate and store all sources of the external JavaScript used in the HTML code. The external JavaScript sources

[4] https://docs.python.org/3/library/asyncio.html

are later scraped as well, ensuring that all of the sources of features will be included in the dataset. Scraping of HTML is visualized in Figure 4.2.

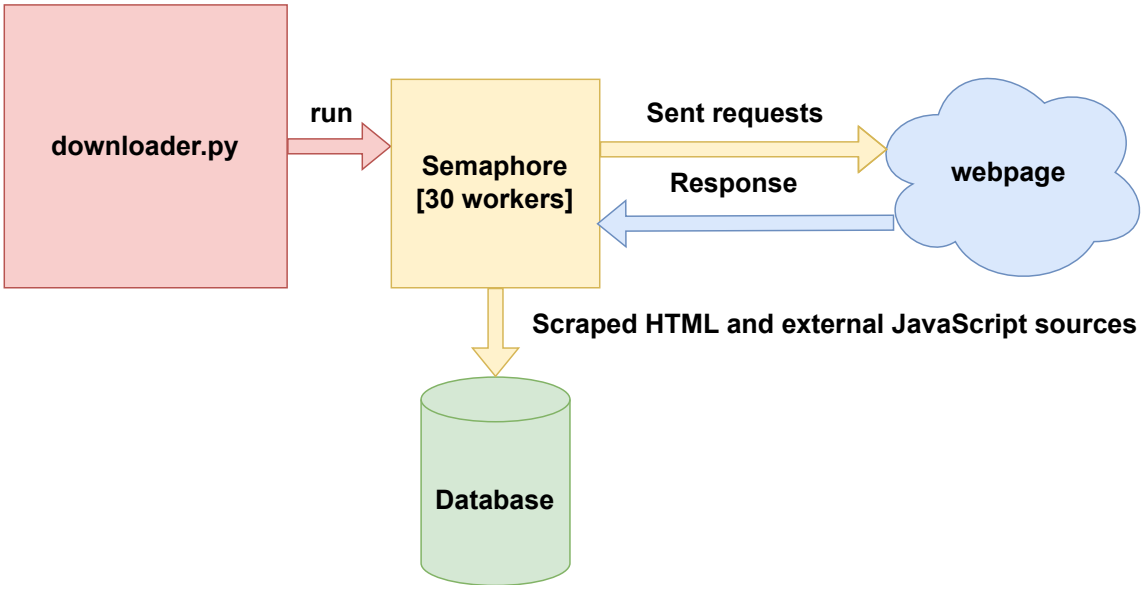

Figure 4.2: Scraping HTML schema.

4.5.3 Scraping external JavaScript

External JavaScript is scraped by the method *GetExternal* in module *downloader.py*. This method accepts two parameters, a list of URLs that will be scraped and a MongoDB collection used as a storage of scraped data. Similarly to Subsection 4.5.2, requests are sent out concurrently with Python's Asyncio and its semaphore. This semaphore only permits 20 workers at once. However, before any requests are sent out, it is necessary to extract external JavaScript sources from the database for every URL given as input. Usually, there are several sources of external JavaScript for a particular HTML file. Naturally, having multiple sources for one web page makes sending out more requests necessary, making this process even more time-consuming than scraping the HTML. The obtained JavaScript code is saved in the database.

4.6 Database

I combined three technologies to store data: Docker[5], Python, and MongoDB[6]. The docker container was built by file *docker-compose.yml*, obtained from [28].

MongoDB is a document-oriented database that stores data utilizing BSON, which is short for binary JSON. Thanks to using BSON instead of JSON[7], MongoDb is faster and has more features, including several extended types for numeric data like int32 and int64 [37]. This works well with Python's type dictionary, which I will use as a placeholder for extracted features and for scraped HTML and JavaScript code. As shown in Figure 4.3,

[5] https://www.docker.com/
[6] https://www.mongodb.com/
[7] https://www.json.org/json-en.html

Docker serves as a hosting service for MongoDB, which the Python application uses as a storage and data source.

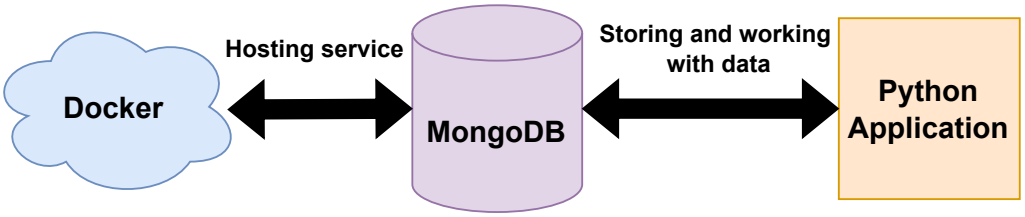

Figure 4.3: Relations between Docker, MongoDB and Python

4.7 Data structure

The saved data have a prescribed form, which is always followed. This form is shown in Figure 4.4. The item domain is always unique because it prevents data from duplicating, which could cause difficulties while training the model. When storing the data, the only item that may be missing is scraped external JavaScript. If a web page has no external JavaScript, the item external javascript urls will be an empty list. A record will not be saved to the database if the item html is an empty string or missing, as this record would not be usable for model training because there would be no features to extract from it.

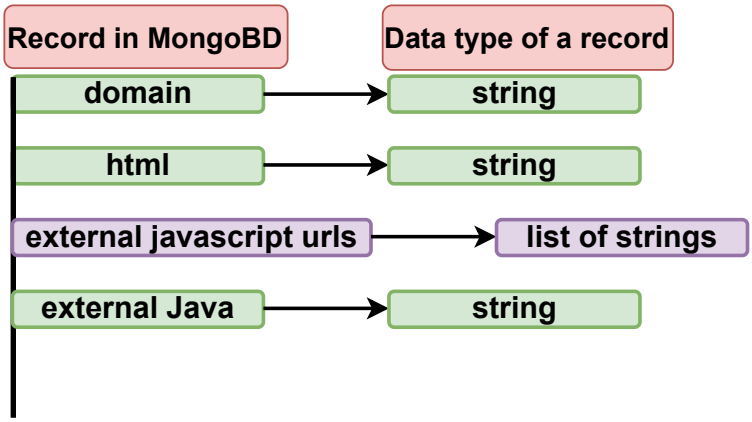

Figure 4.4: Example of how data is stored in database.

4.8 Data filtering

The first round of data filtering is carried out during scraping. Undesired data, such as empty or error responses, are not stored in a database. Automated responses to web scrapers are also included in filtering. These data could confuse the model during training because similar data could be marked as phishing and benign. This filtering is fully automated and does not require any human supervision. The second round of data filtering occurs after the features are extracted from scraped data. Some database entries have all feature values the same despite being scraped from different URLs, and it is vital to remove these duplicates and keep only one such entry to ensure the variety of the data being taught to the model.

Chapter 5

Data analysis

This chapter provides an analysis of scraped HTML and JavaScript data. It includes examining the size of scraped data, analyzing the percentage of encrypted or obfuscated phishing data or exploring the top-level domains found in the scraped data.

5.1 Size of HTML and JavaScript data

As shown in Figure 5.1, the total size of HTML codes scraped from 31481 URLs is 9876 MB. Meanwhile, the size of external JavaScript codes utilized in these URLs is 41956 MB. The sizes were measured using Python's method *sys.getsizeof*, which is capable of determining the size of a Python object. These findings indicate that the external scripts are crucial to analyze as they constitute almost 81% of the code used by all scraped web pages.

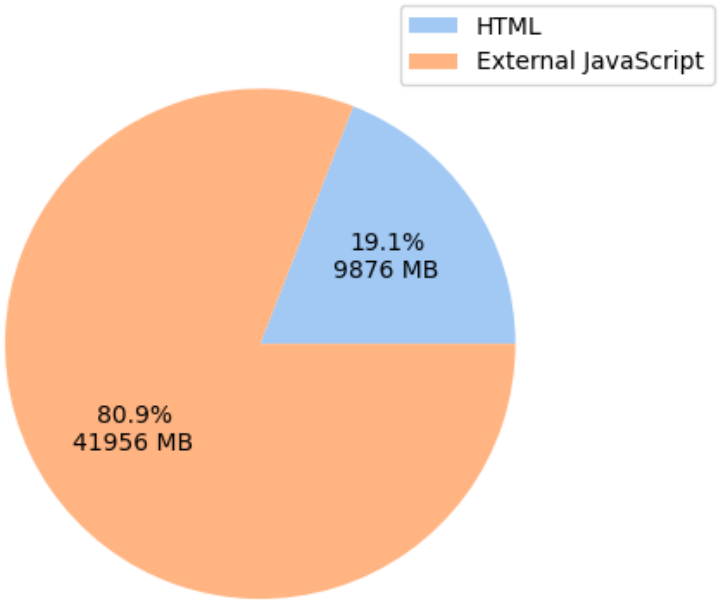

Figure 5.1: Size of scraped data.

5.2 Average number of characters in HTML

The graph in Figure 5.2 illustrates the number of characters detected in scraped HTML. The white circles represent the mean, revealing that benign web pages have, on average, 169847 characters in their code. In contrast, the mean of scraped phishing web pages is slightly lower, with an average of 163543 characters. However, the median, represented by the black lines, reveals that the outliers heavily impact the mean of phishing pages. These outliers are likely encrypted or obfuscated web pages, meaning that the HTML is composed of long strings that significantly increase the number of characters. These findings indicate that benign web pages among scraped data usually contain more characters. This conclusion is supported by a study [17], which identifies document length as one of the possible features capable of differentiating benign from phishing web pages.

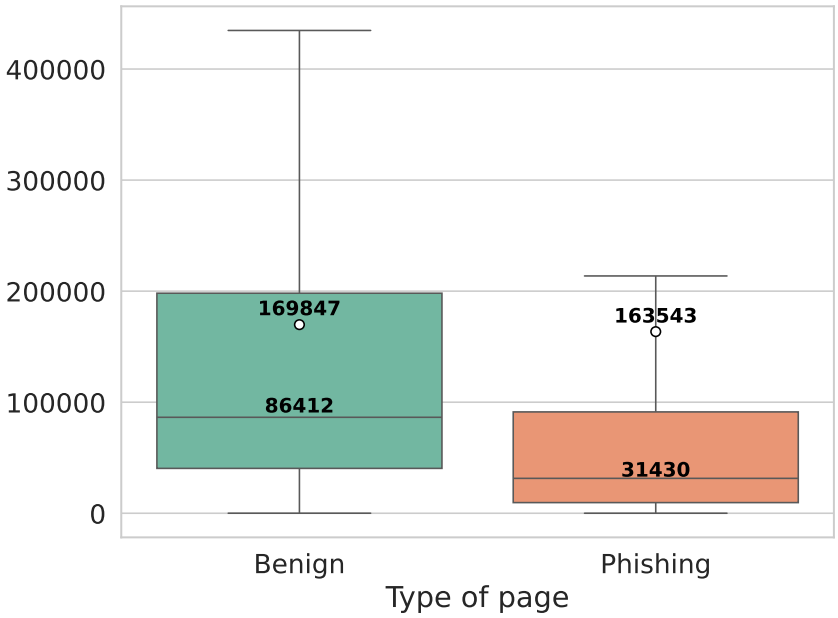

Figure 5.2: Number of characters in HTML code.

5.3 Encrypted and obfuscated data

One of the traits that can be found in malicious HTML code is the encryption of strings [17]. As stated in [31], the code is also being obfuscated. Obfuscation can also be used in benign pages to prevent copying of their code. However, attackers commonly use this method to make the code difficult to analyze, meaning that both encryption and obfuscation fulfill the same purpose of making it impossible to analyze the code that assembles the web page. I found that around 14% of the scraped phishing web pages contained encrypted or obfuscated components in their HTML code, as shown in Figure 5.3. I used the average length of words in an HTML code to detect such strings. If an HTML code had an average word length of more than 70 characters, it was considered to contain these elements. Although 70 characters may seem like a high limit, it is intentionally set up this way because these

pages contain numerous hyperlinks that refer to URLs, significantly increasing their average word length.

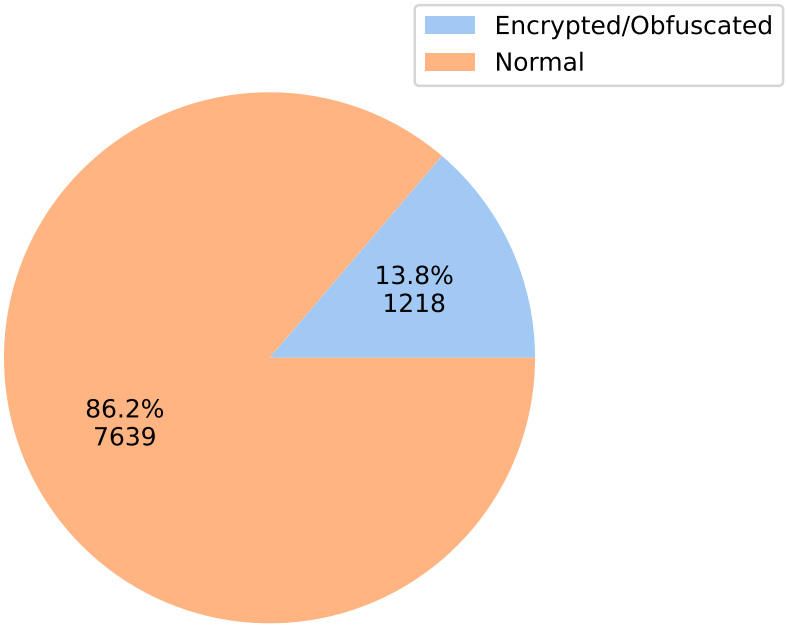

Figure 5.3: Number of phishing sites that contain encrypted or obfuscated elements.

5.4 Top-level domains of scraped URLs

The following findings are based on analyzing the six most common top-level domains, short TLDs, of scraped URLs, shown in Figures 5.4 and 5.5. The most commonly used TLD in both phishing and benign URLs is the *com* TLD, which is utilized by 46% of all websites according to statistics from [47]. Additionally, this statistic informs that TLDs like *org*, *net*, *edu*, *ru* are reasonably popular as well. The second most common TLD utilized in phishing URLs that were used to collect data is the *dev* TLD, which is operated by Google and is being abused by phishers on domains such as *pages.dev* and *workers.dev*, as mentioned in [2]. Many phishing emails containing URLs using these domains have been observed. Additionally, [7] provides a statistic that lists *com* and *net* among the 20 most utilized phishing TLDs. Finally, [36] mentions that attackers abuse free code repositories on GitHub, which enables them to host phishing websites on the *github.io* domain. All of these popular phishing domains are present in the collected data.

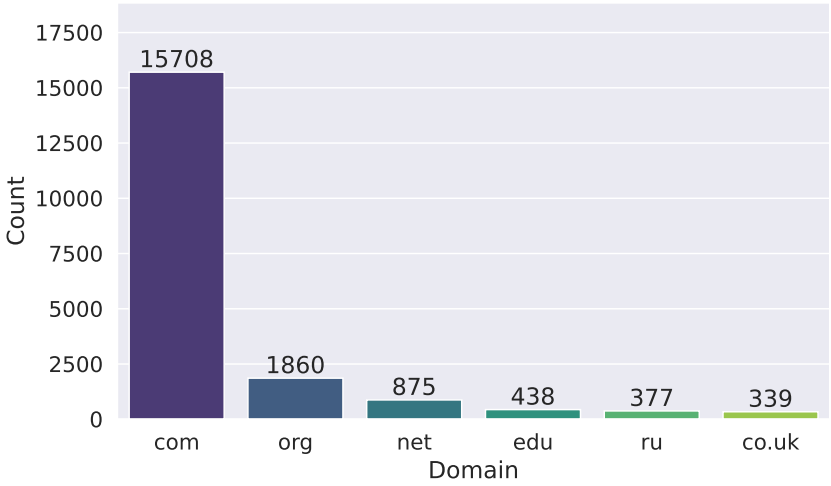

Figure 5.4: Six most common benign TLDs from scraped URLs.

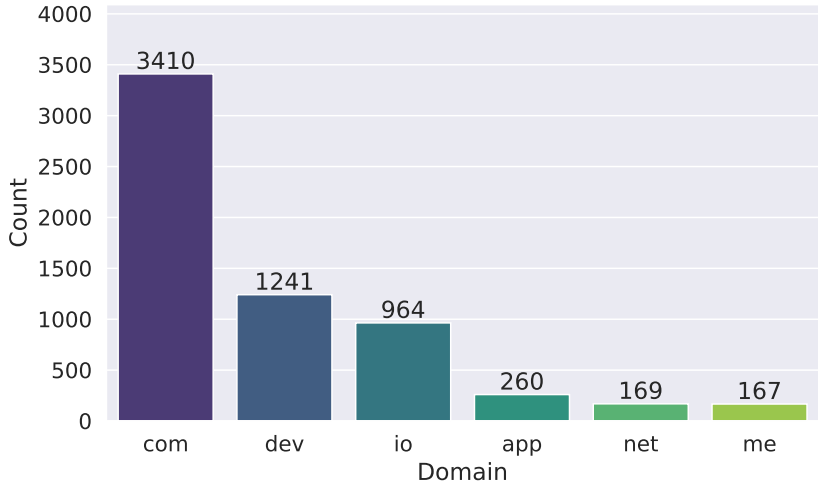

Figure 5.5: Six most common phishing TLDs from scraped URLs.

5.5 Targeted brands

Statistics from [38] and [33] contain information about brands that were most commonly targeted by phishing attacks in the years 2022 and 2023. I searched through the collected phishing data and inspected the titles of scraped phishing web pages to determine the most frequently targeted brands using a list of 37 possible targets. The information I found aligns with the statistics. Of the 37 possible targets, 31 were among the scraped phishing web pages. As shown in Figure 5.6, Netflix is the most targeted brand among the scraped phishing data, followed by ING and Facebook.

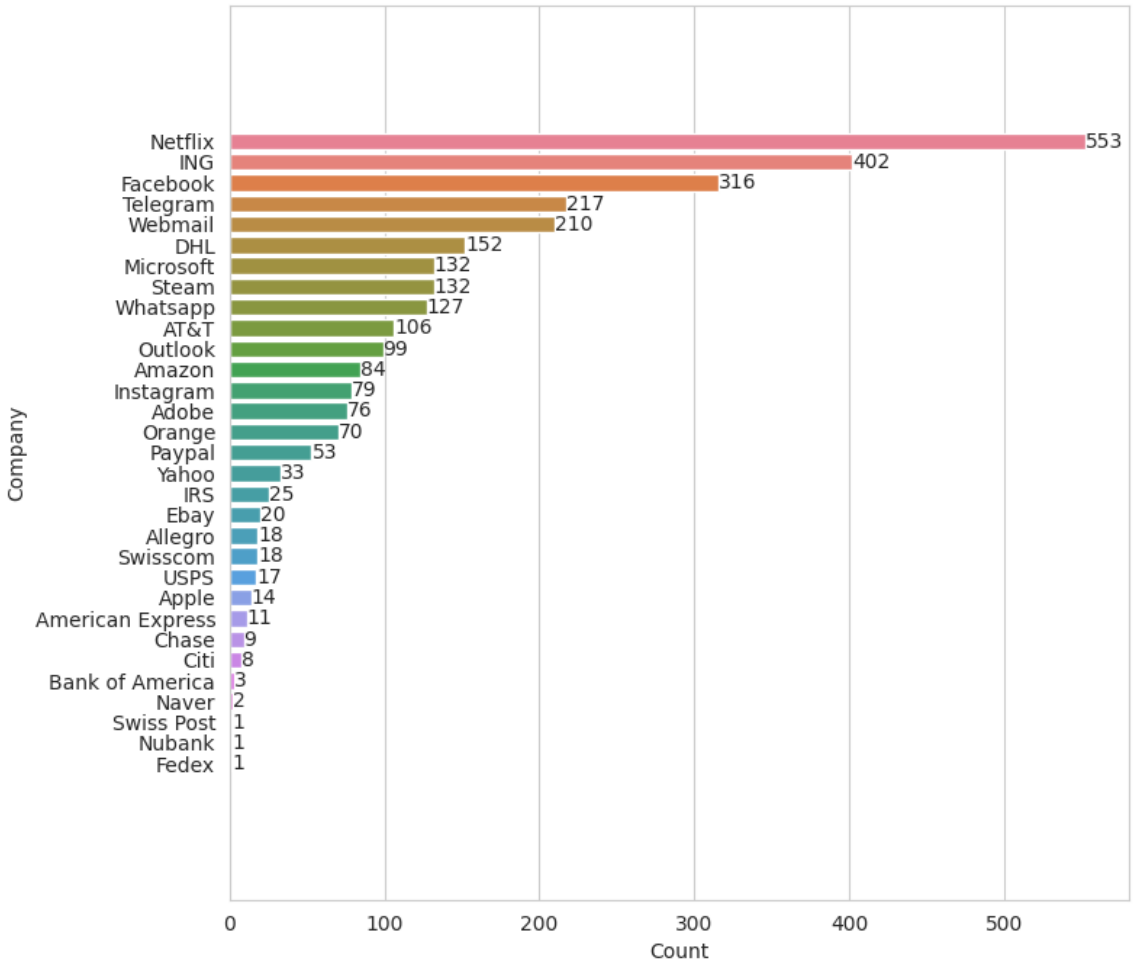

Figure 5.6: Brands targeted by attackers from scraped phishing web pages.

Chapter 6

Feature engineering

This chapter introduces the extraction of features and explains how it is carried out in the program. It clarifies why certain types of features are used and mentions all features that form the feature vector.

6.1 Feature extraction

Janiesch [18] states that the process of feature extraction involves extracting features from the input data, which can be used for building models. A feature is a property that is derived from the input data and provides a proper representation. Shallow machine learning heavily depends on well-defined features, and the performance of these algorithms depends on a successful extraction, which makes this process crucial. On the other hand, deep neural networks use a sophisticated architecture that allows them to automate feature learning. Therefore, deep learning works better with unstructured and enormous data. The feature learning generally moves hierarchically, with high-level features being constructed by the simpler ones. However, different mechanisms of feature learning are employed depending on the type of data and the deep learning architecture selected for the construction of the model.

6.2 Extracting features

To create a list of features, I combined approaches from multiple pieces of research. To be precise, the utilized feature vector is assembled from these studies [25], [31], [34], [10], and [17]. The extraction of features involves three approaches: one for HTML extraction, one for JavaScript extraction, and one for the extraction of the document level features. After successful extraction, the features are stored in a database. This process is illustrated in Figure 6.1. To achieve optimal program performance, the feature extraction process is executed concurrently, similarly to the method described in Section 4.5.

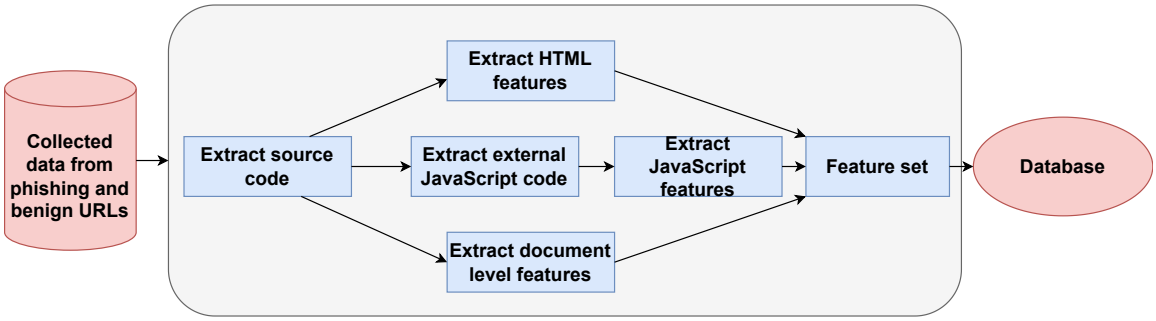

Figure 6.1: Extraction of features in program.

6.3 HTML features

HTML features play a significant role in determining whether a website is benign or malicious. The feature vector includes the HTML features obtained from previous research. To be precise, these researches are: [31], [10], [34], [25]. This thesis combines approaches used in these studies to determine how the combined features will perform.

The HTML features are very powerful. For example, the HTML tags can determine if the page is mainly assembled by parts obtained from external sources by analyzing the hyperlinks [10]. The tags in HTML code can also determine if the webpages are assembled by the code in the HTML document or by JavaScript methods by analyzing the number of utilized tags in the HTML code [31]. Table 6.1 shows all the HTML features that are used for model training.

6.3.1 Hyperlinks

Phishing websites often try to impersonate legitimate web pages to deceive people into sharing their personal information. To keep track of these targeted web pages, phishing attackers often use external hyperlinks to reference foreign domains in order to obtain CSS or icons from the sites they are attempting to copy [10]. Legitimate web pages may also use external hyperlinks, but they typically use internal hyperlinks that point to their local domain. HTML tags related to external information provide beneficial insight into issues related to the usage of external and internal hyperlinks. Listings 6.1 and 6.2 show both legitimate and suspicious methods of loading icons.

```
<link rel="shortcut icon" href="assets/images/favicon.ico"/>
```

Listing 6.1: Benign icon example

```
<link rel="shortcut icon" href="https://cdn-jm-tools.web.app/d..p/others/favicon.ico">
```

Listing 6.2: Suspicious icon example

More than merely the presence of these hyperlinks is required to determine whether the suspected site is actually a phishing web page. According to [10], calculating the ratio of external and internal hyperlinks is a reliable feature that can help identify phishing websites. The computation of these ratios is shown in Figures 6.2 and 6.3. After the computation, the features are assigned binary values: 0 for benign and 1 for malicious.

$$\text{Ratio of internal link} = \begin{cases} \frac{\text{total internal hyperlink}}{\text{total hyperlink}}, & \text{if total hyperlink} > 0 \\ 0, & \text{total hyperlink} = 0 \end{cases}$$

$$\text{Internal href ratio feature} = \begin{cases} 0, & \text{if Ratio of internal link} > 0.5 \\ 1, & \text{otherwise} \end{cases}$$

Figure 6.2: Internal hyperlinks ratio [10]

$$\text{Ratio of external link} = \begin{cases} \frac{\text{total external hyperlink}}{\text{total hyperlink}}, & \text{if total hyperlink} > 0 \\ 0, & \text{total hyperlink} = 0 \end{cases}$$

$$\text{External href ratio feature} = \begin{cases} 1, & \text{if Ratio of external link} > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

Figure 6.3: External hyperlinks ratio [10]

6.3.2 Malicious form

Detecting phishing attacks can be tricky, especially when distinguishing between a legitimate and malicious login form. To help with this issue, I used the malicious form feature.

Generally, the action tag in login or sign-up forms contains an internal hyperlink, indicating that the website is safe to use [10]. However, if the action field of a login form contains external links, a PHP file, or the character # or javascript:void(), then such action tags are considered malicious.

6.3.3 Most common anchor link

To mimic targeted web pages perfectly, attackers need to populate the phishing web pages with links and buttons pointing to other pages. However, it is common practice to populate all of the buttons and links with the same anchor links [10]. This feature helps to track such behavior by finding the most common anchor link, calculating its frequency, and comparing it to the number of anchor links used in the HTML code.

$$\text{Ratio} = \begin{cases} \frac{\text{frequency of most common anchor link}}{\text{total anchores}}, & \text{if total anchores} > 0 \\ 0, & \text{Ratio} = 0 \end{cases}$$

Figure 6.4: Ratio of most common anchor link [10]

Name	Feature description
titles	Title of the HTML document
scripts	<script> tag without set source
objects	Container for an external resource
inputs	Data entrance for user input
strong	Displays text in bold
meta	Metadata in the HTML code
embeds	Container for an external resource
divs	Division or a section in the HTML code
center	Center aligns the text
paragraph	Paragraph in HTML code
imgs	Inserts an image to an HTML code
frame	Inserts another document to the HTML
anchors	Hyperlink or is a placeholder for one
iframe	Inline frame
links	Relation of the HTML and external source
allTags	All tags in the HTML code
mForm	<form> tag refers to an external link, file or placeholder
intHrefs	Attribute refers to an internal source
aToHttps	<a> tag refers to URL starting with https
iframeSrc	<iframe> tag with set location of the external source
formAct	<form> tag with set location where it sends data
aToCom	<a> tag refers to URL ending with .com
extHrefs	Attribute refers to an external source
linkHCss	<link> tag refers to .css source
anchTo#Cont	<a> tag refers to #content
mostCom	Frequency of most utilized hyperlink in <a> divided by all anchors
anchRat	<a> tags that serve as a placeholder divided by all anchors
cssExternal	<link> tag with stylesheets located in an external source
icon	<link> tag refers to shortcut icon
formHttp	<form> tag refers to an external source
allHrefs	Hyperlinks utilized in the HTML
imgsSrc	<img> tag with set source
linksHref	<link> tag with set hyperlink
formPhp	<form> tag refers to a .php file
externalJS	<script> tag with set source
anchorsToHash	<a> tag refering to character #
formJS	<form> tag refers to an empty JavaScript code
formHash	<form> tag refers to character #
iconHttp	<link> tag refers to external shortcut icon
cssInternal	<link> tag with stylesheets located in an internal source
aToVoid	<a> tag refers to an empty JavaScript code
inputPass	<input> tag with type password
scriptAsync	<script> tag with set async
linksType	<link> tag with type set to text/css

Table 6.1: HTML features.

Name	Feature description
linkTypeApp	<link> tag with type set to application/rss+xml
scriptsType	<script> tag with type set to text/javascript
hiddenEl	HTML element has set the hidden attribute
inpHidden	<input> tag has type set to hidden

Table 6.1: HTML features.

6.4 JavaScript features

These are mainly JavaScript methods that are known to be associated with some malicious activities assembled from studies [31, 17]. For example, if an immense average word length is accompanied by string concatenation or methods like escape or unescape, it could indicate that attackers hide the code, which is a highly suspicious activity [17]. Another example is the combination of the methods unescape and write, which is typically used to decode and execute thousands of encrypted characters and generate a website without having a single HTML tag visible in the code of a web page. This is a common trait of malicious websites. The JavaScript features are shown in Table 6.2. All of the JavaScript features represent a number of callings of a specific method.

Name	Feature description
createElement	Modifies the web page
write	Modifies the web page
charCodeAt	Obtains the unicode of the character
concat	Concatenates string values
escape	Creates string where certain characters have been escaped
eval	Evaluates string and returns its completion value
exec	Searches with regular expression for a match in a string
fromCharCode	Returns a string from a unicode value
link	Method wraps a string in an <a> tag
parseInt	Returns an integer parsed from the given string
replace	Replaces match in a string by the given value
search	Searches with regular expression for a match in a string
substring	Returns string that includes the set part of the input
unescape	Creates string where escaped characters are unescaped
addEventListener	Sets up an event attachment
setInterval	Executes code with a delay between calls
setTimeout	Executes code after the timer expires
push	Adds a specified element to the end of an array
indexOf	Returns the index where the element is found in an array
document.write	Modifies the web page
get	Binds an object to a function that will be called
find	Returns the element from an array that matches the test
document.CreateElement	Modifies the web page
window.setTimeout	Executes code after the timer expires
window.setInterval	Executes a code with a delay between calls

Table 6.2: JavaScript features.

6.5 Document level features

Analysis of document level features helps determine if a code is written naturally. These features can also indicate if the code or a part of the code is encrypted or obfuscated. Document level features are very effective as they provide insight into how the HTML code is composed. For example, these features analyze the length of code from different perspectives, and as already mentioned in the analysis of scraped data in Section 5.2, the length of benign and phishing HTML code is polarizing, which makes it a great feature. These features were included from study [17]. Document level features are displayed in Table 6.3.

Name	Feature description
AllLines	Number of lines in an HTML document
AverageWordLength	Average word length in an HTML document
UniqueWords	Number of unique words in an HTML document
AllWords	Number of words in an HTML document

Table 6.3: Document level features.

6.6 Extracting HTML features

To find and store desired features from code, I created method *parseHtml* in module *parse.py*. This method takes three parameters: scraped HTML code in the form of Beautiful Soup constructor, external JavaScript code, and raw HTML. The method utilizes the python package Beautiful Soup[1], which can be used to extract data from HTML and XML files. Beautiful Soup creates a parse tree based on specific criteria that can be used to navigate, extract, and search exact data from HTML. With the help of this package, the *parseHtml* method can extract all of the features mentioned in Table 6.1 and store them in the database.

6.7 Extracting JavaScript features

Figure 6.5 displays how the extraction of features is carried out in the program. The figure shows that the program identifies two types of JavaScript code: inline and external. To avoid additional callings of method *parseJs*, which is a method used to extract JavaScript features, I have combined these two types of JavaScript in the method *parseHtml*, previously discussed in Section 6.6. By using the Beautiful Soup package, I can extract a list of all the inline JavaScript code in an HTML file. Additionally, as mentioned in the previous section, method *parseHtml* has access to the external JavaScript of the page. By combining these two lists, I have created a comprehensive list of all the JavaScript code used in the HTML file. This combined list makes it easier for regular expressions to look for JavaScript features and keep count of them. These pieces of information are then returned to method *parseHtml* and later stored in the database.

[1] https://pypi.org/project/beautifulsoup4/

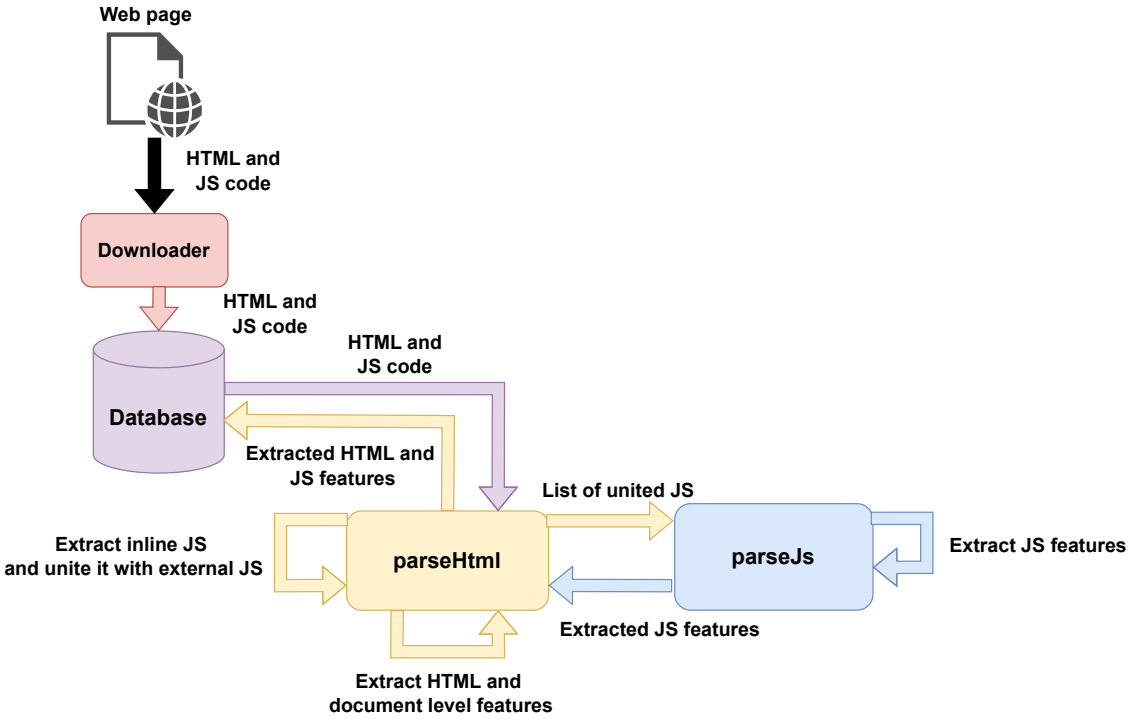

Figure 6.5: Extraction of features in program.

6.8 Extracting document level features

With the usage of basic Python string methods like *split* or *splitlines*, method *parseHtml* is able to quickly extract desired information about the HTML code of the web page and later store these data in the database.

Chapter 7

Dataset

This chapter explains the nature of the created dataset and analyzes the extracted features.

7.1 Information about the dataset

This dataset is created by data scraped from 31481 URLs. As shown in Figure 7.1, the dataset is imbalanced, consisting of 22624 benign and 8857 malicious web pages. As shown in Figure 7.1, the dataset is imbalanced, consisting of 22624 benign and 8857 malicious web pages. This dataset comprises features extracted from HTML and JavaScript code, which are all numerical. The dataset is stored in file *dataset.csv*.

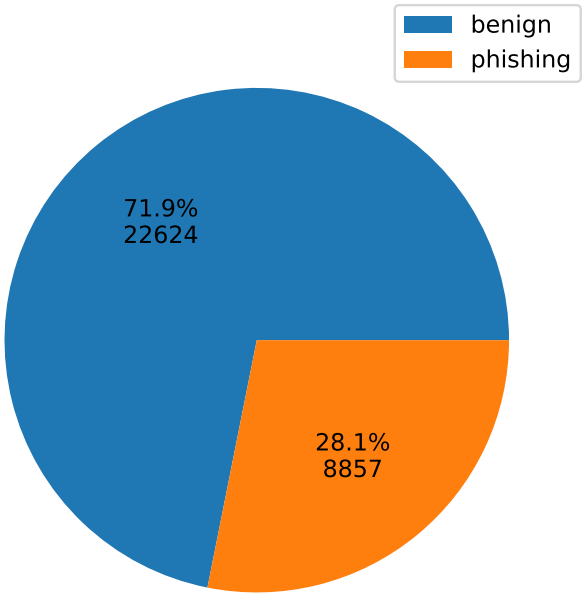

Figure 7.1: Dataset contents.

7.2 Feature correlation

The correlation matrix is utilized in machine learning during feature selection. It reveals the features that provide duplicate information. However, a high correlation does not always indicate redundant information.

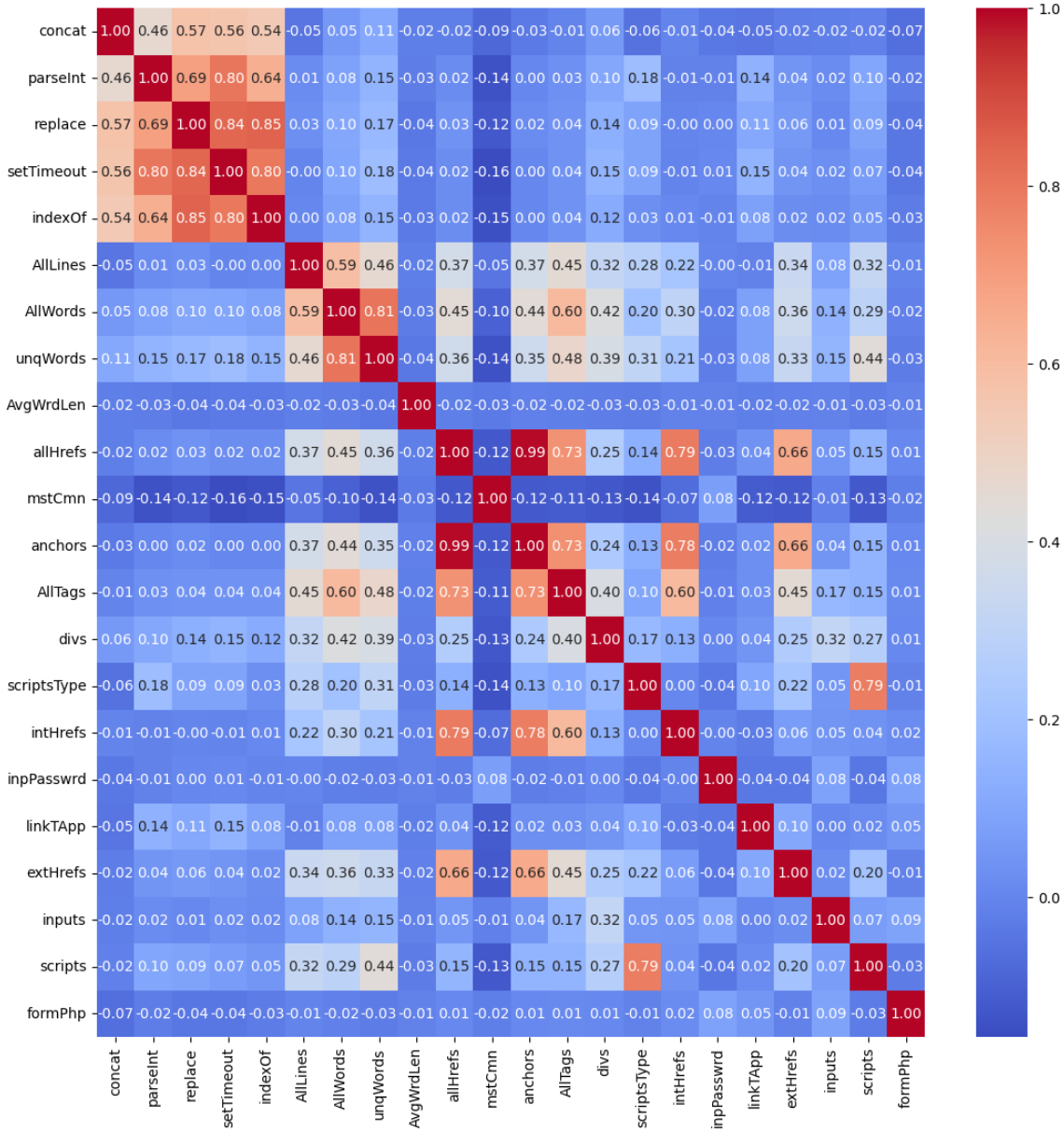

Figure 7.2: Correlation heatmap of 22 features.

Figure 7.2 displays that the highest correlation is between feature anchors and allHrefs. Feature anchors represent the number of all anchor tags and feature allHrefs represent the number of all hyperlinks utilized in the HTML document. The correlation is a result of the HTML tag anchor usually being used with hyperlinks, but hyperlinks can be used with other tags as well. So, this correlation does not indicate redundant information. Other higher correlations can be found between JavaScript methods used in work with strings,

such as replace and indexOf. This correlation does not indicate redundant information, as these methods are usually combined but not always. Removing these features only based on correlation would cause information loss during model training. The correlation matrix of all features can be found in Appendix B.

7.3 Analyzing document level features

Table 7.1 displays the average values of document level features extracted from scraped HTML. These values indicate that benign websites, on average, tend to have more lines of code, more words, and more unique words in their code, which is in line with expectations as phishing web pages tend to use hyperlinks to load content from legitimate web pages. Another expected difference was the average word length, which was found to be significantly longer in phishing pages. As noted in [17], it is widespread for phishing pages to contain long encrypted strings. This statement also applied to scraped phishing data in this dataset, as discussed in Section 5.3. Observing the average word length makes it possible to detect phishing web pages that hide their suspicious parts of code. Some phishing pages even encrypt or obfuscate their entire code, making the extraction of other features impossible.

Feature	Benign	Phishing
All lines	1860	753
All words	8140	2810
Unique words	2713	1071
Average word length	18	4982

Table 7.1: Average values of document level features.

7.4 Analyzing HTML features

Table 7.2 shows the average values of 15 selected HTML features from the dataset. Considering that data from Section 7.3 reveal that benign web pages, on average, contain more HTML code, it is not surprising that they contain more HTML tags than phishing web pages. The statistics about anchors and hyperlinks show an enormous gap between phishing anchors and hyperlinks and benign anchors and hyperlinks. Study [10] presents that legitimate websites have many other web pages connected via links so you can browse them. On the other hand, phishing websites have a limited number of web pages. This explains why this gap is so huge. However, what might come up as a surprise is that, on average, phishing web pages from this dataset use more internal than external hyperlinks. This contradicts the information about phishing pages duplicating most of their code from benign web pages mentioned in Subsection 6.3.1. Nevertheless, the data shown in Figure 7.3 can explain this. Approximately 20% of the internal hyperlinks found in phishing web pages from this dataset do not refer to any file, image, or script. They either refer to the top of the page itself by character #, to an empty string or trigger empty JavaScript code. These hyperlinks are usually used as fillers for buttons that trigger action on legitimate pages. This is connected to the previously mentioned statement that legitimate websites have many other web pages connected via links, but phishing websites have a limited number of web pages. However, using these internal hyperlinks as fillers means that most of the loaded content comes from external sources, which was originally expected. Despite

phishing pages utilizing fewer input tags, these pages ask for more passwords on average. Another essential difference between benign and phishing HTML code is reusing the same link in anchor tags. On average, 40% of all hyperlinks in anchor tags from phishing web pages refer to the same link, which serves as another page populating tacting similar to the usage of characters like #.

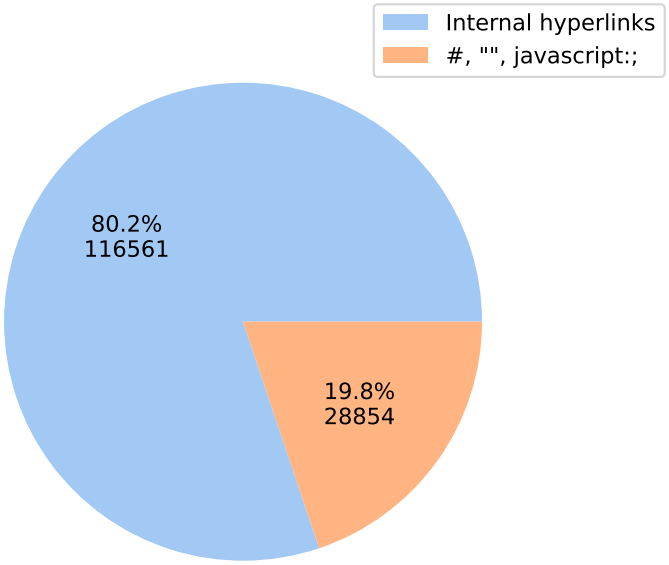

Figure 7.3: Percantage of internal hyperlinks that do not point to any content.

Feature	Benign	Phishing
All hyperlinks	205	24
Ratio of most commonly referred URL	0.1	0.4
Anchors	193	17
All tags	990	180
Divs	186	53
Scripts with type „text/javascript“	14	3
Internal hyperlinks	87	13
Inputs with type „password“	0.1	0.3
Links with type „application/rss+xml“	0.6	0.1
External hyperlinks	118	11
Inputs	8	4
Inline JavaScripts	17	5
Forms with action to .php file	0.2	0.2
Anchors to .com	24	2
Links that reffer to HTTPs	11	3

Table 7.2: Average values of 15 HTML features.

Figure 7.4 compares the values of features allHrefs, inputPassword, and anchors in phishing and benign HTML documents. The analysis shows that the combination of heavy usage of anchors and hyperlinks indicates that the web pages are benign. On the other hand, using fewer hyperlinks and having multiple inputs marked as passwords is more typical for phishing web pages. A similar relation is between the input password and a number of anchor tags. Using fewer anchors and asking for more passwords signifies a phishing web page. These relations are very polarizing, as apparent differences between phishing and benign web pages can be found in the plots. These features seem to be very consistent, making them reliable to utilize for classifying web pages.

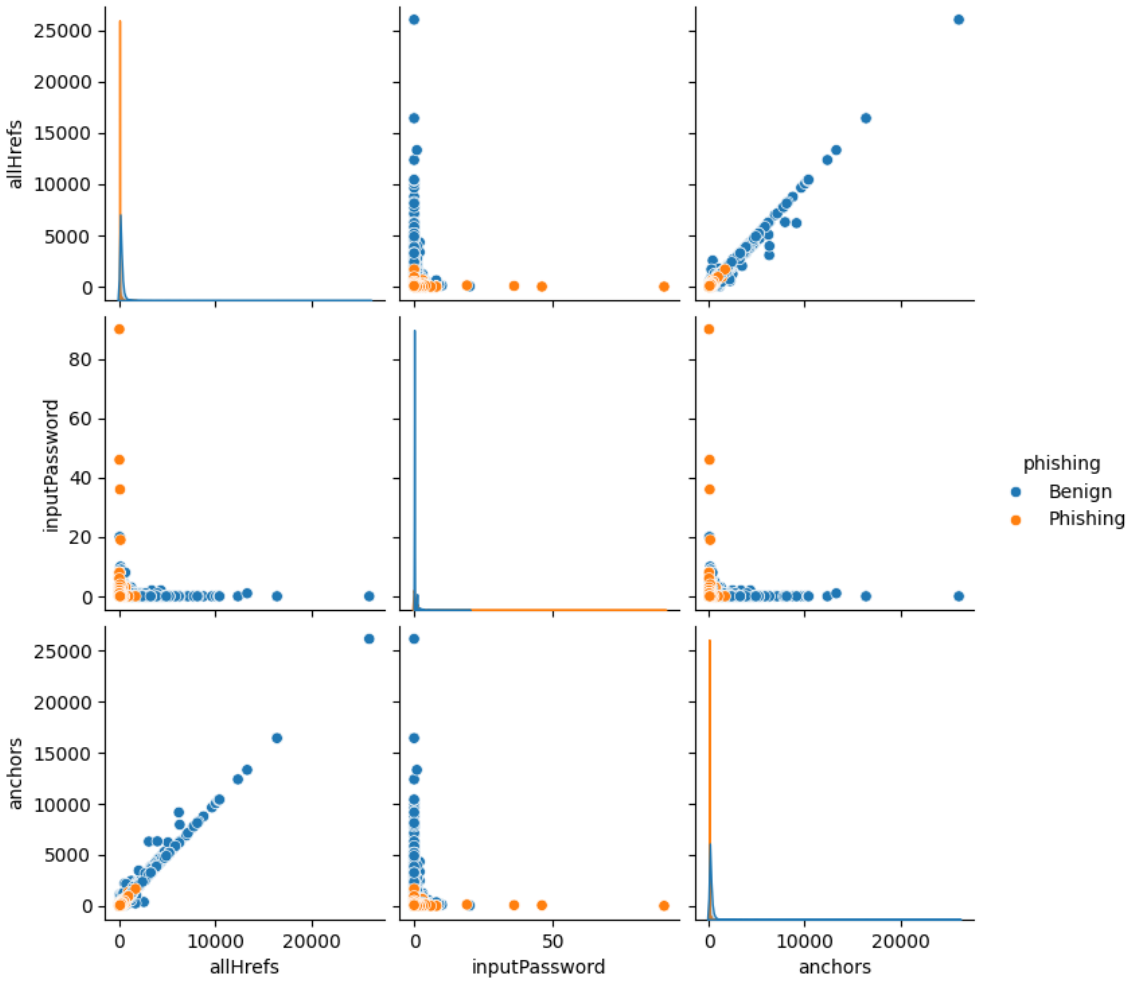

Figure 7.4: Relation between HTML features.

7.5 Analyzing JavaScript features

Table 7.3 displays the average values of 15 selected JavaScript features from the dataset. These values suggest that benign web pages, on average, utilize slightly more JavaScript methods. This is interesting because according to data in Table 7.1, benign web pages have two times more lines of code in them, which suggests that phishing web pages are assembled mainly by JavaScript methods, which generate pages and methods used for code obfuscation or encryption and decryption. However, the presence of these methods alone does not indicate that the site is benign or malicious. These features provide more valuable information when incorporated with the other features. For example, [17] states that the combination of immense average word length and usage of function eval indicates malicious activity.

Feature	Benign	Phishing
exec	19	21
escape	2	1
push	198	153
substring	22	12
document.CreateElement	19	9
search	2	1
find	59	24
createElement	89	64
get	94	62
window.SetTimeout	2	2
concat	101	102
parseInt	36	20
replace	88	66
setTimeout	35	22
indexOf	95	58

Table 7.3: Average values of 15 JavaScript features.

Figure 7.5 also supports the statement from [17]. The figure displays relations between JavaScript methods escape, concat, and a document level feature average word length in phishing and benign web pages. The relation of average word length and method escape shows that higher average word length and few uses of method escape, often used in code obfuscation, indicate a phishing web page. However, a lower average word length indicates that the code is not obfuscated or encrypted. Even when combined with higher usage of method escape, this information still indicates a benign web page.

Similarly, method concat is associated with obfuscation, and the plot displays the same pattern. The relation of the two JavaScript methods indicates that the heavy usage of these methods is not problematic as the benign pages utilize these methods more, but benign pages do not have such an enormous average word length. These plots confirm that the JavaScript features provide more helpful insight when accompanied by other features.

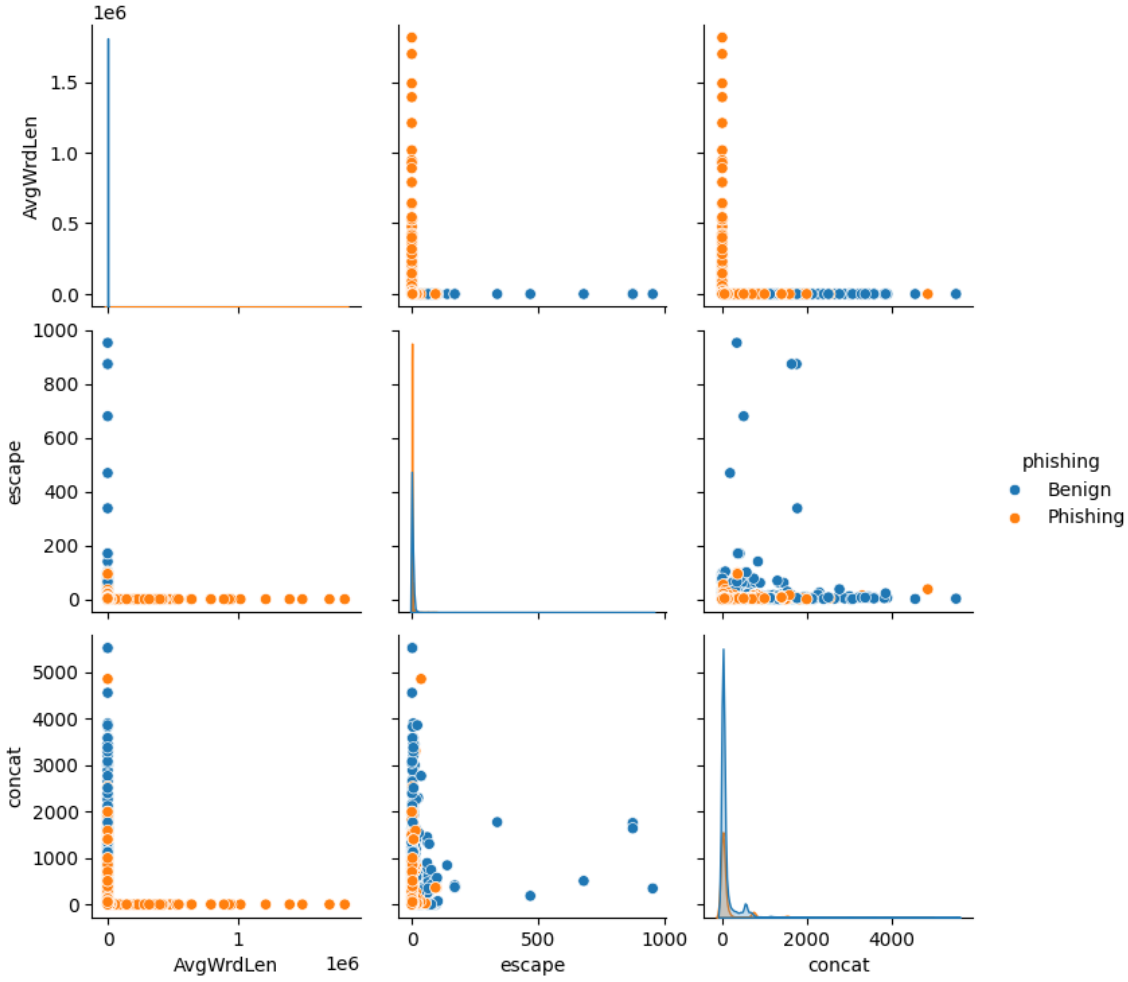

Figure 7.5: Relation between JavaScript features and Average word length.

Chapter 8

Training and tuning of models

This chapter provides information on creating and fine-tuning various phishing detection models trained by using different machine learning algorithms. It also introduces utilized evaluation metrics and tuning of hyperparameters and compares the performance of each untuned and tuned classifier.

8.1 Evaluation metrics

These are metrics used to evaluate created models. In my program, I utilize accuracy score, balanced accuracy score, precision, recall, ROC-AUC curve, false positive rate, false negative rate, and F1 score.

As stated in [21], accuracy score is a measure that informs how many of all the made predictions were correct. Figure 8.1 shows how the accuracy of the model is calculated. All accurate predictions of class one called true positives, and true negatives, which are correct predictions of class two, are summed up. These accurate predictions are then divided by a summary of all predictions. The summary comprises true positives, true negatives, false negatives, predictions that inaccurately labeled class one, and false positives, which inaccurately marked class two.

This metric can be misleading when the evaluated data contains more samples of one class. When evaluating such data, the accuracy score fails to punish the misclassification of underrepresented data, which is why I only utilize the accuracy score in the controlling mechanism described in Subsection 8.5.2.

$$\text{Accuracy} = \frac{\text{true positives} + \text{true negatives}}{\text{true positives} + \text{true negatives} + \text{false negatives} + \text{false positives}}$$

Figure 8.1: Calculating accuracy

On the other hand, a balanced accuracy score is a measure that is utilized to evaluate the performance of classification models on imbalanced data, according to [12]. The balanced accuracy is calculated using true positive and negative rates. The true positive rate, also known as recall, is a metric that evaluates if the model correctly identifies true positives. On the other hand, the true negative rate is the number of negative samples correctly predicted by the model. Figure 8.2 displays how these metrics are calculated. The balanced accuracy considers both minority and majority classes. It provides a fair description of the

performance that the evaluated model achieved, making it more trustworthy when working with imbalanced data.

$$\text{Balanced accuracy} = \frac{\text{TPR} + \text{TNR}}{2}$$

$$\text{TPR} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

$$\text{TNR} = \frac{\text{true negatives}}{\text{true negatives} + \text{false positives}}$$

Figure 8.2: Balanced accuracy

Figure 8.3 displays how precision is calculated. Precision defines how many predicted positives were labeled correctly.

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

Figure 8.3: Precision

As shown in Figure 8.4, thanks to combining precision and recall, the F1 score is better at evaluating models trained on imbalanced datasets than accuracy. If the trained model has low precision and high recall, the F1 score will punish this. For example, if the precision value was 40% and recall was 80%, the F1 score of this model would be only 53%.

$$\text{F1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Figure 8.4: Calculating F1

Figure 8.5 displays how false positive and negative rates are calculated. The false positive rate is the probability that a true negative will be misclassified. Similarly, the false negative rate is the probability that a true positive will be misclassified. It is beneficial to evaluate these properties of classifiers as they can indicate how reliable the classifier is.

$$\text{FPR} = \frac{\text{false positives}}{\text{false positives} + \text{true negatives}}$$

$$\text{FNR} = \frac{\text{false negatives}}{\text{false negatives} + \text{true positives}}$$

Figure 8.5: False positive rate and false negative rate

The last utilized evaluation metric is the ROC-AUC curve. This metric provides information about how much the classifier can distinguish between classes. The higher the values of the AUC score, the better the model is at distinguishing. The ROC curve is created by comparing the true positive rate and false positive rate of a classifier to a random assignment. The AUC value is a representation of the area under the ROC. The total area is equal to 1, which means a very good classifier should have a value of AUC close to 1.

8.2 Data preparation

The dataset used for training the model must be extracted and preprocessed before the training. The module *train.py* and method *getTrainData* were created to achieve this. This method takes two optional arguments *collection*, which contains MongoDB collection where the training dataset is stored, and *csvFile*, if collection with data is not specified, the dataset will be loaded from the given csv file. The dataset is then extracted and stored in pandas[1] data frame. The data frame is then split into two. One contains features, and one contains class labels. These data frames are then split into training and testing parts. 80% of these data frames is stored in variables *dX_train*, which contains features of web pages and *y_train*, which contains a label of these pages. These variables are used to train the model. The other 20% is stored in variables *X_test* and *y_test*. These variables are used to test the created model as the model previously never saw these data.

However, some machine learning algorithms, such as SVM, perform better when the training data are normalized [23]. For this case, the module *train.py* contains method *getNormalizedTrainData*, which normalizes the data before splitting them into training and testing samples. This normalization is carried out using the *MinMaxScaler* from package sklearn[2], which utilizes a Min-max method. As shown in Figure 8.6, this method scales each feature based on maximal and minimal values to a range from 0 to 1. After normalizing data frames, they are split into a training and testing set. The method *getNormalizedTrainData* takes four optional arguments *collection* and *csvFile*, which operate the same way as in method *getTrainData* and arguments *saveScaler* and *filename*. These arguments enable the saving of the scaler utilized to normalize training data, which makes it possible to apply the same scaler later while making predictions in a non-training environment, as the data will have to be normalized by the same scaler as the model that makes the prediction. The scaler that I utilized during training is saved in file `scaler.joblib`, which is later utilized during experiments to scale the input data to the format used during the training of the classifier. The scaler clips the data to a range of 0 and 1, meaning if in production some feature had a higher value than X_{max}, such feature will not be scaled to a value bigger than 1.

$$\mathrm{X} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Figure 8.6: Min-max normalization

8.3 Model training in program

The goal of creating the models is to achieve the most reliable and accurate predictions possible in a short time period so that predictions can be made in real-time. For these reasons, four machine learning algorithms are utilized in the program. These are LightGBM, XGBoost, Support vector machine, and Neural Networks. Models of these algorithms are tuned and trained by methods from module *train.py*.

Methods *GetLightGBMmodel* and *getXGBoostModel* function in a very similar way, and both take one argument *trainData*, which is a list of variables that is obtained by method *getTrainData* mentioned in Section 8.2. These methods create and return an instance of

[1] https://pandas.pydata.org/
[2] https://scikit-learn.org/stable/

LightGBM or XGBoost classifier, which both have set tuned hyperparameters obtained by tuning methods described in Section 8.5 and Section 8.8.

Method *getSVMmodel* operates similarly but takes argument *normalizedTrainData*, which can be obtained by utilizing method *getNormalizedTrainData*. It is crucial to use the normalized data because the algorithm Support vector machine performs best with such data. The created classifier is again trained with tuned hyperparameters obtained from the tuning method described in Section 8.11.

Finally, the neural network training is performed by method *getNNModel*. This method accepts one parameter *trainData*. The architecture of the neural network in my program is composed of several layers. The first layer is the normalization layer. This layer is responsible for the normalization of the input data. Thanks to this layer, it is possible to take the unnormalized data as input, making it more convenient to make predictions on data in a non-training environment. Because this scaler is already built into a model, there is no need to save it to normalize the previously unseen data. The normalization layer is followed by the input layer, dropout layer, hidden layer, another dropout layer, and finally, the output layer. To avoid overfitting, I employed a method that consists of inserting a dropout layer between the input and hidden layers and another dropout layer between the hidden and output layers. [16] states that this technique randomly disconnects some neurons in a neural network during training. These disconnects can be beneficial and prevent overfitting because they force the neural network to learn more robust features, as it can not rely on any particular neuron. These layers utilize hyperparameters, activational functions, and an optimizer determined during tuning described in Section 8.14.

8.4 LightGBM model training

The initial model trained before any tuning achieved promising results according to the evaluation metric F1 score. During training, this model was tested on data previously unseen by the model and achieved an F1 score of 92%. While this score is great for an untuned model, it is essential to analyze the behavior of this model further because a great score during training does not mean the model will perform well in a non-training environment.

The confusion matrix in Figure 8.7 visualizes the performance of the trained model during testing. The confusion matrix shows that the model successfully identified 4369 benign web pages and 1646 phishing web pages. However, it identified 204 benign web pages as phishing pages and 78 phishing pages as benign web pages. Table 8.1 displays a more detailed analysis of the performance of the untuned model. The false positive and false negative rates are relatively similar, which means that the untuned model struggles with the misclassification of both classes similarly. The achieved precision also confirms that the untuned model struggles with predicting benign labels, as the achieved precision score is the worst of all metrics. Even though the achieved balanced accuracy is great, with a score over 95%, the model can still be improved by proper tuning, decreasing false positive and false negative rates, thus making the classifier more reliable.

F1 score	Recall	Precision	Balanced accuracy	FPR	FNR
92.11%	95.48%	88.97%	95.51%	4.46%	4.52%

Table 8.1: Results of untuned LightGBM model.

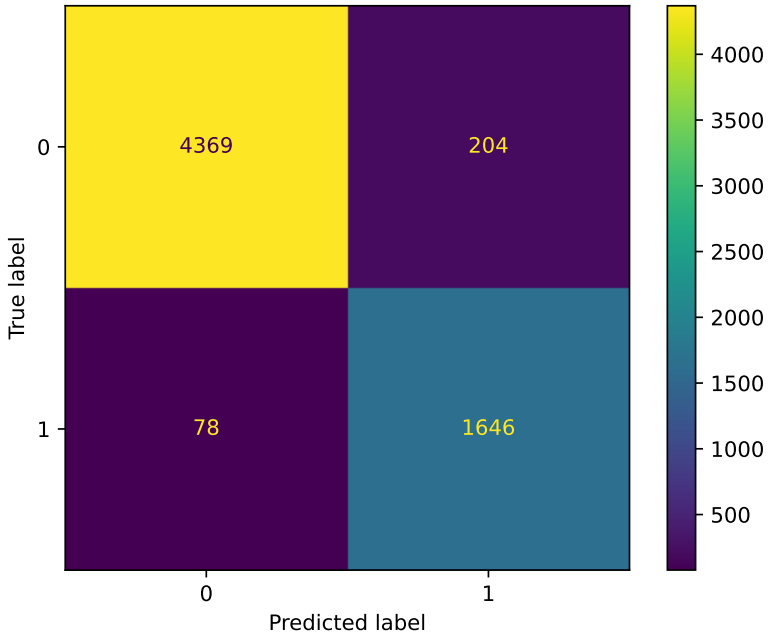

Figure 8.7: Confusion matrix of untuned LightGBM model.

8.5 LightGBM hyperparameter tuning

The tuning of hyperparameters is an automated process carried out by method *optimizeLightGBM* in module *train.py*. This method takes three arguments *trial*, *dataTrain* and *dataControl*. Trial is part of a greater study provided by package *optuna* and its class *Study*. This trial can suggest multiple possible values to different hyperparameters of a classifier and test predictions of these classifiers. It compares the results of each and tries to maximize the values of evaluation metrics in every trial. In the end, it picks the trial with the best results and returns its hyperparameters.

8.5.1 LightGBM hyperparameters

To efficiently and correctly use *optuna* and its studies, it is crucial to understand what parameters need to be tuned and approximately what values should be suggested to the classifier. Many parameters can cause overfitting if they are set incorrectly, and many other can help fight the overfitting of the model.

[24] states that to fight the problems with overfitting, it is needed to use smaller `num_leaves`, which is a value that sets the maximal number of leaves in one tree, then smaller `max_bin`, which is a parameter that is setting a maximal number of bins that feature values will be bucketed in and smaller `min_child_samples`, which is a value that is setting a minimal number of data in one leaf. Another suggested method to fight overfitting is utilizing bagging by setting `bagging_fraction` and `bagging_freq`, and also employing feature subsampling by setting `colsample_bytree`. Using regularization by trying parameters like `lambda_l1`, `lambda_l2` and `min_split_gain` can also lower the chances of overfitting. Avoiding deeper trees by setting lower `max_depth` can also be beneficial.

[24] also provides information on achieving higher prediction accuracy. It is advised to use large `max_bin`, small `learning_rate` with large `n_estimators`, which is a parameter setting a number of boosting iterations, or use large `num_leaves`.

8.5.2 Avoiding overfitting

The dangerous part of optimizing hyperparameters with *optuna* studies is that the trials suggest values of hyperparameters at random with the intent to achieve the best score possible. This intent can easily cause an overfitting issue when the model performs very well on the training and testing data, but in the real environment, the model will fail. In other words, the information gain of the model will no longer bring any benefit to it.

To prevent this issue, I implemented a controlling mechanism. I created a small dataset that includes contents of scraped benign web pages that phishing attackers very commonly impersonate. Among these pages are Facebook, Instagram, PayPal, Netflix and Microsoft. The dataset also contains phishing web pages that target these benign pages. Every created model in a study will then be tested by making predictions on the testing part of the dataset and also on predicting values of pages from this small control dataset. This testing ensures that the trained model is capable of spotting a difference between a benign login page and a phishing web page. This method works as a kind of cross-validation, but instead of randomly choosing a part of the training dataset, I collected these pages that may be difficult to classify. This control dataset is represented in method *optimizeLightGBM* by argument *dataControl* and is stored in file `controlData.csv`. After the *optuna* study ends, it returns a trial that achieved the best F1 score on testing data and the best accuracy on the control dataset.

Listing 8.1 shows how this process is implemented in code. The method uses a trial to suggest possible parameter values for the LightGBM classifier. After the suggestion of values, the model is trained and makes predictions on testing samples. Then, the model makes predictions on the control dataset. These predictions are evaluated by metrics F1 score and accuracy score. The accuracy score is used only to evaluate the predictions made on the control dataset because I only care about models that achieve 100% accuracy on this dataset, as tuned models must be able to differentiate phishing web pages from benign login pages. These evaluations are returned and used later to find the best-performing tuned model.

```
def optimizeLightGBM(
        self,
        trial: optuna.Trial,
        dataTrain: list[pd.DataFrame, pd.DataFrame, pd.DataFrame, pd.DataFrame, float],
        dataControl: list[numpy.ndarray, list],
    ) -> list:
    lightGBM = lightgbm.LGBMClassifier(
        max_depth=trial.suggest_int("max_depth", 1, 15),
        num_leaves=trial.suggest_int("num_leaves", 2, 1024),
        n_estimators=trial.suggest_int("n_estimators", 10, 4000),
        lambda_l1=trial.suggest_float("lambda_l1", 0.00001, 0.1),
        lambda_l2=trial.suggest_float("lambda_l2", 0.00001, 0.1),
        bagging_fraction=trial.suggest_float("bagging_fraction", 0.05, 1.0),
        bagging_freq=trial.suggest_categorical("bagging_freq", [0, 1]),
        max_bin=trial.suggest_int("max_bin", 256, 1024),
        learning_rate=trial.suggest_float("learning_rate", 0.01, 0.3),
        min_child_samples=trial.suggest_int("min_child_samples", 2, 100),
        min_split_gain=trial.suggest_float("min_split_gain", 0.00001, 2),
        colsample_bytree=trial.suggest_float("colsample_bytree", 0.05, 1.0),
    ) # example of parameter tuning
    model = lightGBM.fit(dataTrain[0], dataTrain[2]) # fitting X_train and y_train
    predictions = model.predict(dataTrain[1]) # predictions on testing part of dataset
    predictionsC= model.predict(dataControl[0]) # predictions on control dataset
    return [f1_score(dataTrain[3], predictions),accuracy_score(dataControl[1], predictionsC)]
```

Listing 8.1: Tuning of hyperparameters in code

8.6 Results of LightGBM tuning

The tuning of hyperparameters caused an increase in the F1 score of the model. The score of the tuned model is 94.7%. As shown in Figure 8.8, the number of benign pages classified as phishing pages is drastically lower than on the untuned model. This improvement implies that the tuned model can more reliably discover differences between legitimate login and phishing pages. The tuning caused a slight increase in the number of phishing pages classified as benign. These results are also confirmed by Table 8.2. The tuned model managed to achieve a very good false positive ratio of 2%. The precision of the model also drastically increased, resulting in an improved F1 score. A balanced accuracy score also indicates that the model is more reliable than the untuned model. The false negative rate increased, but overall, the tuning positively impacted the model.

F1 score	Recall	Precision	Balanced accuracy	FPR	FNR
94.7%	94.78%	94.61%	96.37%	2.03%	5.22%

Table 8.2: Results of tuned LightGBM model.

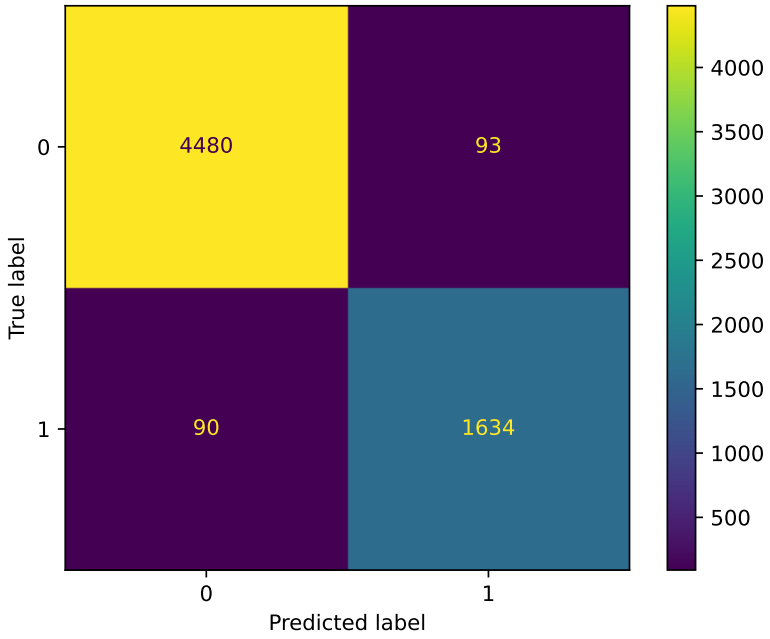

Figure 8.8: Confusion matrix of tuned LightGBM model.

8.7 XGBoost model training

The untuned model, which included only default values of hyperparameters, performed with a very good F1 score of 93.6%. Figure 8.9 presents a confusion matrix of the untuned XGBoost model. This confusion matrix presents a balanced number of false positives and false negatives. However, it is vital to keep the class imbalance in mind, as the false negative rate in Table 8.3 displays that the model struggles with classifying phishing web pages significantly more than benign pages. This imbalance could be caused by a slight overfitting of the classifier during training, which might cause this bias. Both precision and recall scores indicate that the untuned classifier is good at predicting benign pages. The achieved balanced accuracy score is also good. The main goal of tuning the XGBoost model is to decrease the false negative rate value by solving the slight overfitting problem, thus achieving a more reliable classifier.

F1 score	Recall	Precision	Balanced accuracy	FPR	FNR
93.62%	94.08%	93.16%	95.74%	2.60%	5.92%

Table 8.3: Results of untuned XGBoost model.

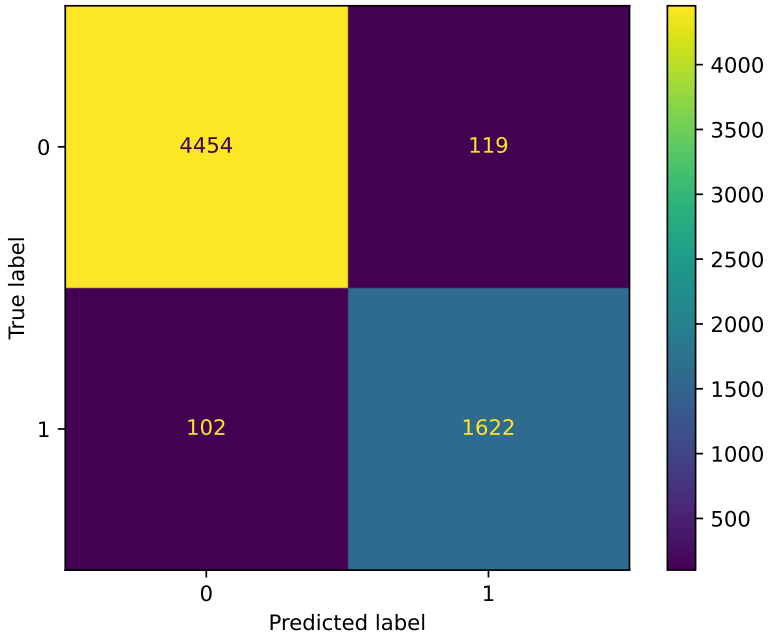

Figure 8.9: Confusion matrix of untuned XGBoost model.

8.8 XGBoost hyperparameter tuning

Method *optimizeXGBoost* is responsible for tuning the XGBoost model. This method takes three arguments. The first one is *trial* followed by *dataTrain* and *dataControl*. The arguments are the same as those described in Section 8.5, as both optimizing methods are based on the package *optuna* and its study. Similarly, the mechanism that prevents the overfitting of the trained module is the same as described in Subsection 8.5.2. The hyperparameters of the XGBoost classifier are similar to the LightGBM as these machine learning algorithms both utilize gradient-boosted decision trees. This means that parameters like `max_depth`, `learning_rate`, `max_bin`, `n_estimators` are all present and helpful both with overfitting issues and increasing the knowledge of the model. However, XGBoost handles imbalanced data differently. The algorithm takes parameter `scale_pos_weight` to handle imbalanced data properly. This parameter is a ratio of the number of negative instances and the number of positive instances. This ratio is included in the argument *dataTrain*.

8.9 Results of XGBoost tuning

The tuned XGBoost model finished with an F1 score of 94.49%, which is a slight increase compared to the untuned version. As shown in Figure 8.10, the tuned model has improved in classifying phishing and benign web pages and is now more consistent in making predictions. These results of tuning are also indicated by evaluation metrics displayed in Table 8.4. The achieved values of evaluation metrics have improved by model tuning, resulting in a more reliable and accurate classifier. The XGBoost model is on par with the tuned LigthGBM

model. However, the XGBoost model is slightly worse at classifying benign web pages but has a lower false negative rate than the tuned LightGBM classifier.

F1 score	Recall	Precision	Balanced accuracy	FPR	FNR
94.49%	95.01%	93.98%	96.36%	2.30%	4.99%

Table 8.4: Results of tuned XGBoost model.

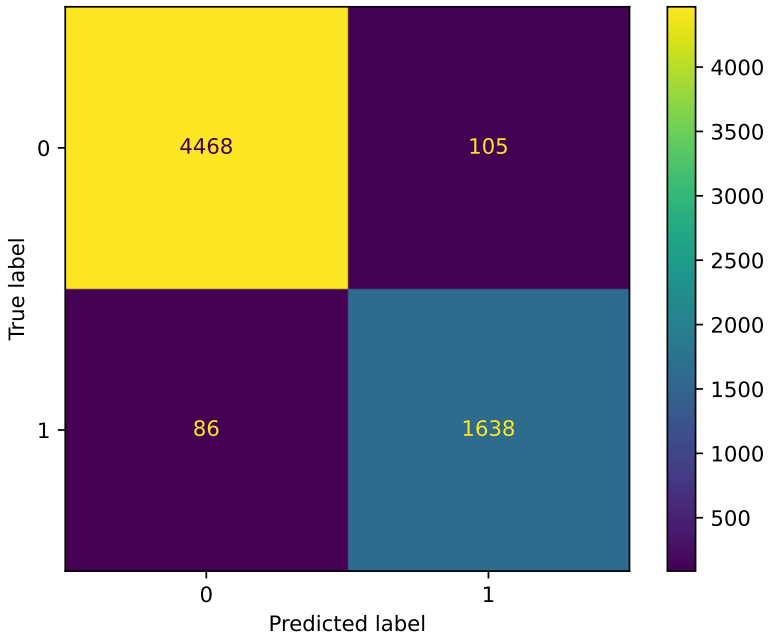

Figure 8.10: Confusion matrix of tuned XGBoost model.

8.10 Support vector machine model training

The untuned SVM classifier performed much worse than the previous ones, with an F1 score of 76%. The confusion matrix shown in Figure 8.11 confirms this. The number of false positives and false negatives is much higher than in previous algorithms. Table 8.5 also displays much worse results than the results achieved by previous untuned classifiers. The model fails to classify both classes reliably, but the false negative rate is higher than the false positive rate, which indicates that the classification of phishing pages is more problematic than the classification of benign pages. The achieved precision score shows that only 71% positive predictions were correct, confirming that the classification of benign pages is also troublesome. The hyperparameter tuning should be helpful, as the untuned model seems to have issues with underfitting.

F1 score	Recall	Precision	Balanced accuracy	FPR	FNR
76.06%	81.90%	70.99%	84.64%	12.61%	18.10%

Table 8.5: Results of untuned SVM model.

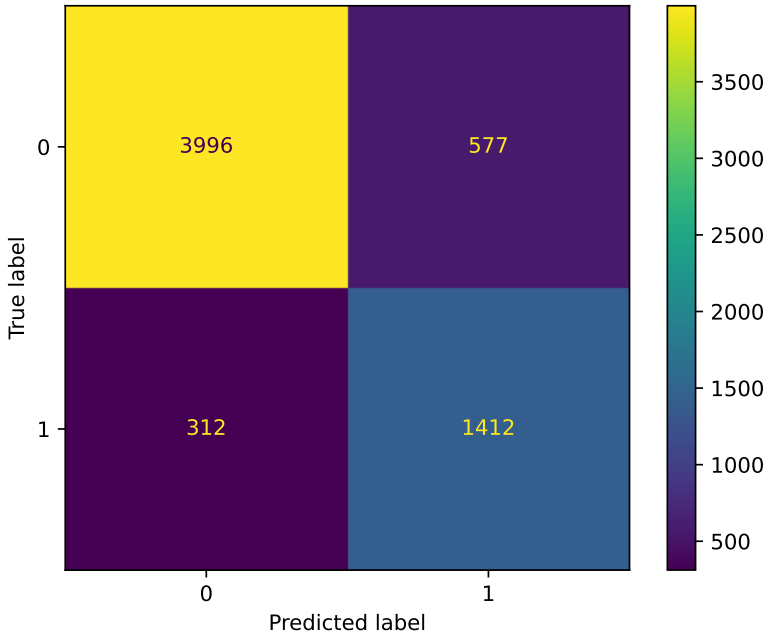

Figure 8.11: Confusion matrix of untuned Support vector machine model.

8.11 SVM hyperparameter tuning

The tuning of the Support vector machine model is carried out by method *optimizeSVM*. The tuning is performed by utilizing class *GridSearchCV* from package *sklearn*. This class performs the model training with previously given parameters, which should be tried. When it finishes training all models, it chooses the one that achieved the best score on specified evaluation metrics. Support vector machine offers much less variety of hyperparameters than LightGBM or XGBoost. However, some crucial parameters must be tuned to achieve optimal results.

One such is parameter `C`. As stated in [1], it is a regularization parameter that influences the trade-off between maximizing the margin and minimizing the classification error, which means that a smaller value of `C` allows for a broader margin and, therefore, more misclassifications. In comparison, a more significant value of `C` punishes misclassifications more heavily, leading to a narrower margin. In other words, a higher value of `C` allows for more flexibility in the decision boundary, potentially leading to overfitting. Because the algorithm penalizes the errors while classifying more heavily. On the other hand, a lower value of `C` imposes a smoother decision boundary and may lead to underfitting. A lower `C` value also leads to a better generalization of unseen data.

The following significant parameter is `gamma`. According to [1] `gamma` is used to determine how powerful will be the influence of the respective data points on the decision boundary. In other words, the higher the value, the fewer data points will impact the decision boundary, which can lead to overfitting. On the other hand, smaller values allow more data points to influence the decision boundary, which makes the decision boundary more generic.

The last parameter is `kernel`. This parameter allows one to choose from several kernels, such as linear, RBF, or sigmoid. Choosing an appropriate kernel is very important to get great results. In my program, I chose between kernels RBF, poly, and sigmoid as they are non-linear kernels, and the training data are non-linear. I determined the linearity of data by the method described in [26]. Determining whether the dataset is linear is possible by utilizing Linear regression and evaluation metric R-squared. If the evaluation metric R squared shows a value close to 1, it implies that the dataset is linear. The training dataset had an R-squared value of 0.43, which implies that the dataset is non-linear.

8.12 Results of SVM tuning

After tuning the hyperparameters, the F1 score of the SVM model is 87.43%. This improvement is also displayed in Figure 8.12. The number of false positives and negatives has significantly dropped. However, the confusion matrix also reveals that the issue with labeling benign web pages as phishing pages prevails even after tuning, suggesting that the model might not be as reliable in spotting differences between benign login pages and phishing pages. Table 8.6 displays a detailed analysis of the performance of the tuned SVM classifier. Thanks to tuning, the model got better in every displayed metric. The model is more reliable than the untuned counterpart, as the false positive and false negative rates dropped significantly. However, these rates are relatively high, and the model is less reliable and accurate than the previous models. Even though the tuning was successful, the classifier performs much worse than the untuned versions of LightGBM and XGBoost models.

F1 score	Recall	Precision	Balanced accuracy	FPR	FNR
87.43%	93.97%	81.74%	93.03%	7.92%	6.03%

Table 8.6: Results of tuned SVM model.

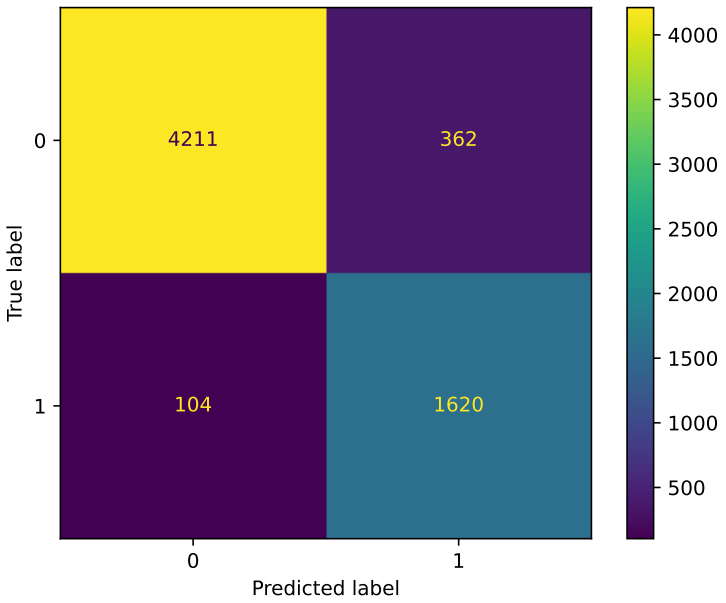

Figure 8.12: Confusion matrix of tuned Support vector machine model.

8.13 Neural network model training

The F1 score of an untuned neural network model is almost 80%. Moreover, the untuned classifier performs slightly better than the untuned SVM model. However, Figure 8.13 displays a much different problem than any previous classifier struggled with. The untuned model heavily struggles with marking benign pages as phishing web pages. The precision score in Table 8.7 also confirms this. The achieved precision is only 69%. However, the untuned classifier lacks activation functions and an optimizer. The classifier should perform significantly better after fine-tuning these significant parts of the neural network.

F1 score	Recall	Precision	Balanced accuracy	FPR	FNR
79.74%	93.74%	69.39%	89.07%	15.59%	6.26%

Table 8.7: Results of untuned Neural network model.

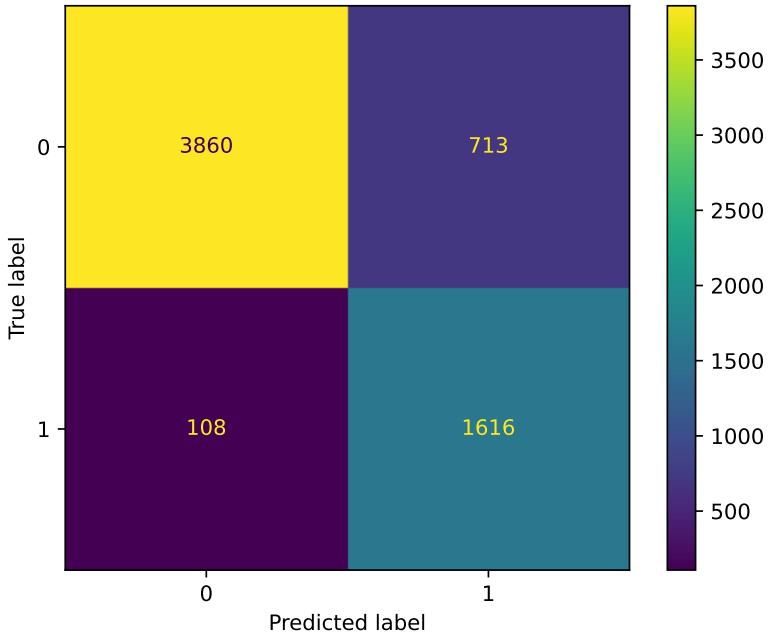

Figure 8.13: Confusion matrix of untuned Neural network model.

8.14 Neural network hyperparameter tuning

The hyperparameters are tuned by method *optimizeNn*, which takes two parameters *tuner* and *dataTrain*. The tuner parameter suggests values for the training of models. The dataTrain parameter contains the training part of a dataset and will be normalized by the normalization layer before tuning. The training of the model is carried out by *Hyperband* from package *keras_tuner*. This tuner uses a sports championship-style bracket. The algorithm trains many models for a few epochs and carries only the best-performing half of the models to the next round until it finds the best-performing classifier. This style saves a lot of time and resources. It evaluates the performance of models by given evaluation metrics, in this case, precision and recall.

As mentioned in Section 8.3, the neural network has an architecture that consists of layers. The input and hidden layers have certain parameters which need to be tuned. One such is `activation`. This parameter chooses an activation function for a layer. [6] suggest three activation functions to try out during model tuning. These are `relu`, `sigmoid` and `tanh`. While using these activation functions, it is advised to use normalized input data. The next parameter to tune in layers is called `units`. This parameter specifies how many neurons will be employed in a particular layer. The last parameter for layers is `rate`. This parameter is used in dropout layers to determine what fraction of neurons will be disconnected during training. The only layer with a set activation function and a number of units is the output layer because the model needs to perform a binary classification, which is achieved by utilizing a sigmoid activation function and one unit.

While compiling a neural network model, the algorithm uses an optimizer. This optimizer is employed to change the parameters of the neural network, such as weights or

learning rate, to reduce the losses during training [8]. While tuning, the program chooses from three optimizers `adam`, `sgd`, and `rmsprop`. All of these optimizers require a learning rate, which is also suggested by the tuner.

8.15 Results of Neural network tuning

After tuning, the neural network model achieved a 91.41% F1 score, which is a significant improvement. The confusion matrix in Figure 8.14 shows that the tuned model can classify web pages much better than the untuned classifier. The number of web pages inaccurately classified as phishing pages has dropped from 713 to 203. This improvement suggests that the initial problem with classifying benign web pages as phishing pages that the untuned model had has been solved by tuning. The other metrics have all significantly improved, as shown in Table 8.8. The false positive rate and the precision score achieved significant improvements, resulting in a more reliable classifier. The performance of the tuned neural network model is much better than that of the tuned Support vector machine model. The tuned neural network model outperforms it in every metric. However, the tuned model still does not achieve the performance of LightGBM or XGBoost.

F1 score	Recall	Precision	Balanced accuracy	FPR	FNR
91.41%	94.08%	88.88%	94.82%	4.44%	5.92%

Table 8.8: Results of tuned Neural network model.

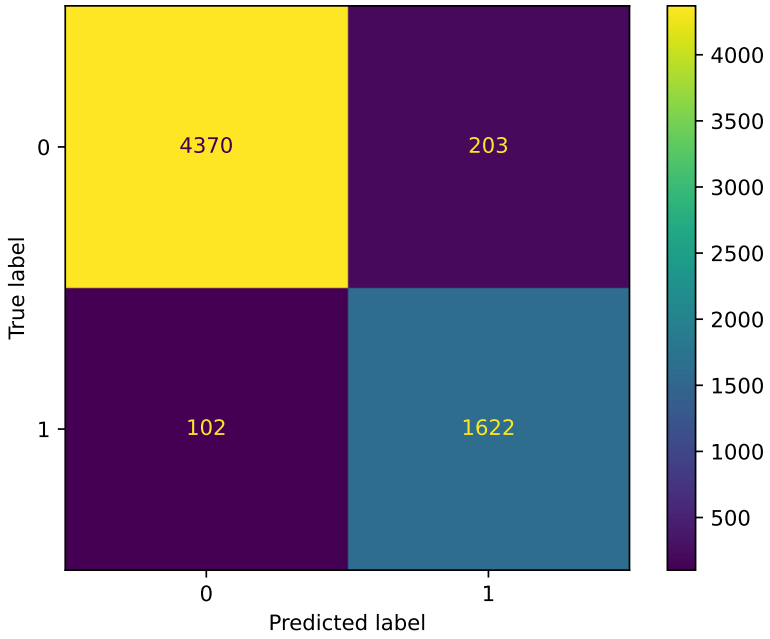

Figure 8.14: Confusion matrix of tuned Neural network model.

8.16 ROC-AUC analysis of tuned classifiers

The ability of classifiers to differentiate between benign and phishing pages is crucial for successfully detecting phishing web pages. That is why this analysis, which verifies the ability to distinguish between the two classes, is essential to determine if the classifiers are of great quality.

Figure 8.15 displays the ROC-AUC analysis of tuned classifiers. The x-axis displays values of false positive rate, which is the probability that a true negative will be misclassified. The y-axis consists of the true positive rate, a metric that evaluates if the model correctly classifies true positives. As stated in [45], the ROC curve, displayed as the orange line on the plots, represents the trade-off between the TPR and FPR while trying to increase TPR. The ROC curve compares the TPR and FPR of the classifier, while values of the true and false positives change as the threshold value changes. The threshold represents the value that determines how to convert the predicted values into class labels. As described in [32], a test with perfect discrimination has a ROC curve that passes through the upper left corner. That is why the closer the ROC curve is to the top left corner of the plot, the better the tested classifier is. The displayed plots show that the curves are close to the top left corner, indicating good quality. The AUC score in the ROC-AUC curve represents the ability to differentiate between the classes. The closer this value is to 1, the better the model differentiates. The achieved AUC score of all tuned classifiers is close to 1, meaning that all classifiers can spot differences between two classes well.

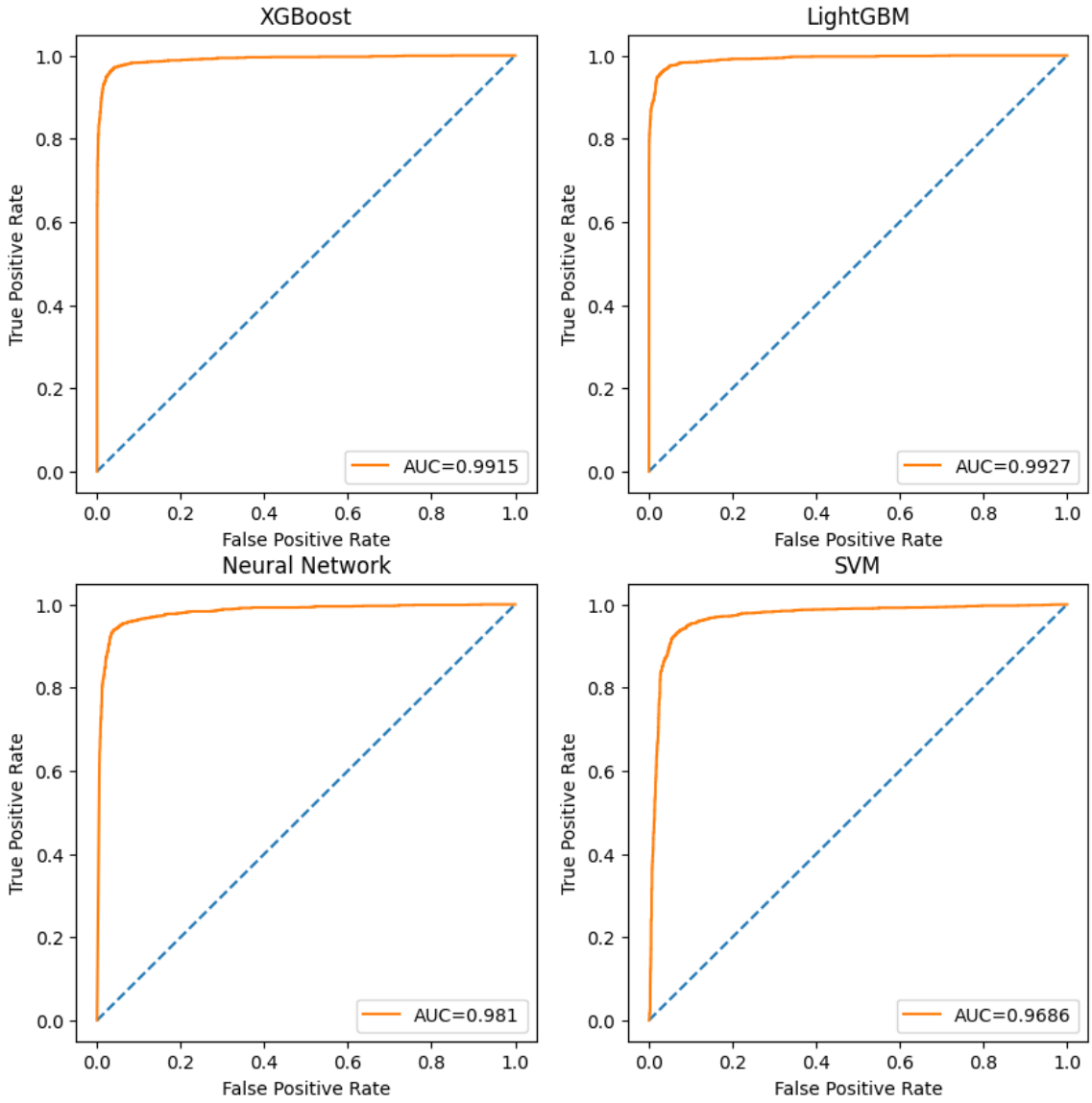

Figure 8.15: ROC-AUC of all tuned classifiers.

8.17 Feature importance

Another way to understand the classifier and relations between features and target variable is to analyze the feature importance of each classifier. To analyze the feature importance of the LightGBM and XGBoost classifiers, I utilized Python's package shap[3]. The shap values indicate the impact of each feature on the final prediction or the importance of every feature. These values are based on game theory, and each feature is assigned an importance value in the model.

[3] https://shap.readthedocs.io/en/latest/

8.17.1 LightGBM feature importance

Figure 8.16 shows the five most important features utilized in the LightGBM classifier according to shap.

The most important feature utilized by the LightGBM classifier is `allHrefs`, which is the number of all hyperlinks used in an HTML file. Study [10] states that legitimate websites have many other web pages connected via links, and in HTML, the destinations of these links are specified by hyperlinks, which means that the number of hyperlinks in a benign page should be higher on average. This statement was already confirmed to apply to the training dataset while analyzing HTML features in Section 7.4. This parameter proved very polarizing as the average number of hyperlinks used in a benign web page from the dataset is significantly higher than in a phishing web page. This significant difference might be why the LightGBM classifier relies on this feature the most.

The second most important feature is `mostCommon`, which is the frequency of the most common anchor link divided by the number of all anchors. This feature is based on the fact that phishing web pages usually feed their pages with the same link in several anchors. The detailed reasoning behind this feature was already discussed in Subsection 6.3.3. While analyzing the average values of HTML features in Section 7.4, it was discovered that, on average, 40% of anchors contain the same link in phishing web pages while only 10% of links are the same in benign pages. This significant difference might explain why the feature is so important.

The third most important feature is `anchors`, a number of all anchor tags utilized in the HTML. This feature is related to the usage of hyperlinks, as the anchor tag is usually paired with a hyperlink. The analysis of average values of HTML features in Section 7.4 shows massive differences in anchor usage in phishing and benign pages. Similar to hyperlinks, benign pages use anchor tags much more than phishing pages, which might be the reason why this feature is so beneficial.

The features `AllTags` and `divs` are similar. These features are very polarizing, as shown when analyzing the HTML features. On average, benign pages utilize many more div tags and HTML tags. It is tied to the fact stated in [10] that the phishing pages usually utilize hyperlinks to mimic other pages. This means that phishing pages utilize fewer HTML tags to assemble the page and contain fewer lines of code. The massive difference in the usage of divs and HTML tags might be the reason for the high importance of these two features.

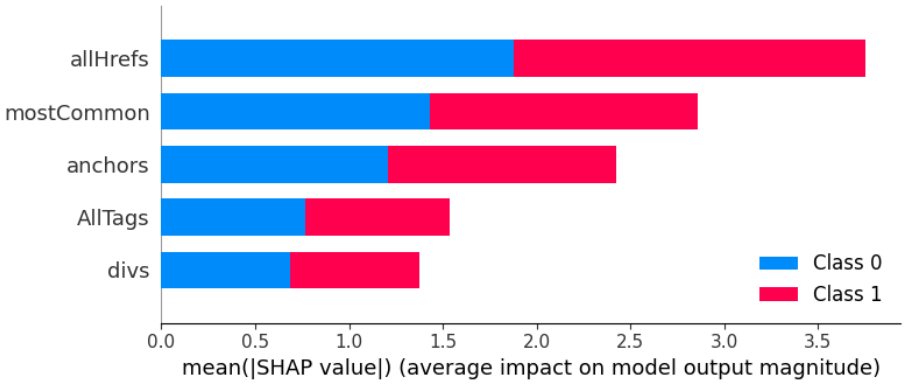

Figure 8.16: Five most important LightGBM features according to SHAP.

8.17.2 XGBoost feature importance

Very similar features are dominant in the XGBoost classifier, as shown in Figure 8.17. The only different feature is called `internalHrefs`, which keeps count of internal hyperlinks used in HTML. As mentioned in Subsection 6.3.1, this feature is included in the feature vector because phishing pages usually obtain their content from external sources, while benign pages utilize their internal sources. While analyzing the values of HTML features in Section 7.4, it was found that benign pages utilize much more internal hyperlinks than phishing pages. This vast difference may be why this feature is among the five most important.

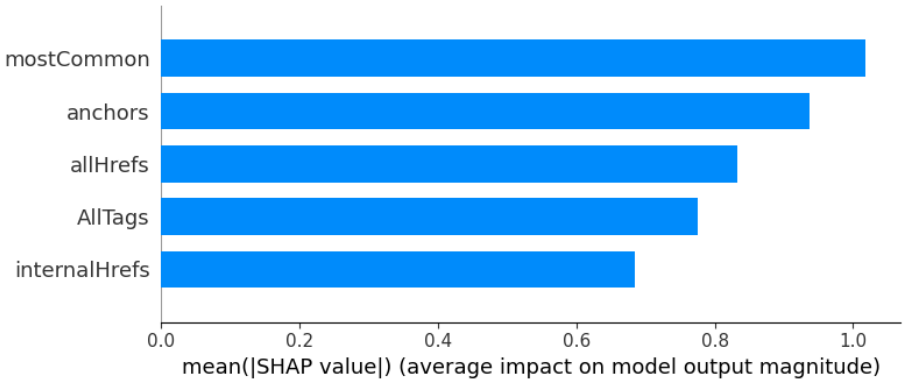

Figure 8.17: Five most important XGBoost features according to SHAP.

8.17.3 SVM feature importance

Because the SVM classifier does not use a linear kernel, the data are transformed to another space unrelated to the input space, so getting the feature importance is more complicated. Using the permutation feature importance technique makes it possible to obtain the feature importance. According to [22], this method is especially beneficial for non-linear classifiers. It randomly rearranges the values of a single feature and observes the degradation of the score achieved by the classifier. Breaking the relationship between the feature and the target variable makes it possible to determine how much the model relies on the particular feature. The results of the permutation importance of the SVM classifier are shown in Figure 8.18.

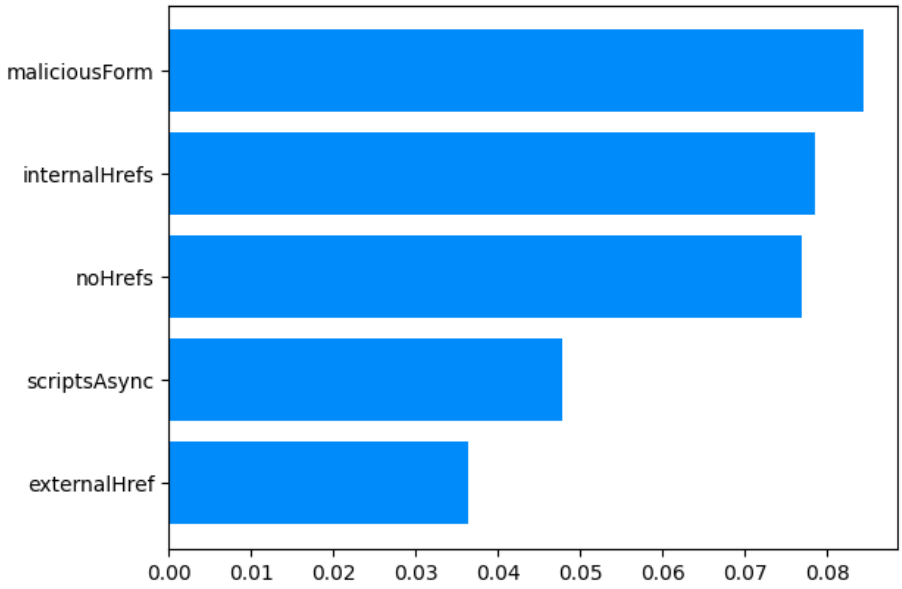

Figure 8.18: Five most important SVM features according to permutation importance.

The most important feature in the SVM classifier is called `maliciousForm`, which is an HTML form that refers to an external link, a PHP file, character #, or javascript:void(). As mentioned in Subsection 6.3.2, phishing attackers commonly utilize these forms to obtain data from their victims. This feature is used to differentiate benign and phishing login forms, which might be why the SVM classifier considers it the most important feature.

The SVM classifier also relies on the feature `noHrefs`. This binary feature is 1 when the HTML document contains no hyperlinks. Other times, it is 0. This feature is based on the information that benign web pages have many other links connecting them, and phishing web pages have only a limited number. Moreover, the phishing pages might hide or encrypt their HTML code, making the analysis of hyperlinks impossible, which makes this feature very useful in such cases. As mentioned in Section 5.3, almost 14% of scraped phishing pages contain encrypted or obfuscated elements, which might be why this feature is so important.

The feature `scriptAsync` represents a number of inline scripts with attribute async. The async scripts only work with external scripts. These scripts are usually used as scripts that run counters or display ads on web pages as these scripts run independently on other scripts and do not wait for DOM. The reason SVM considers this feature important might be the heavier usage by benign pages, as these web pages usually display ads.

The last of the most important features for the SVM classifier is feature `externalHref`, which represents the number of external hyperlinks in an HTML document. This feature is included in the feature vector because the phishing pages try to mimic other web pages and usually use external hyperlinks to achieve this. Another reason is the previously mentioned statement that benign pages have many other pages connected via links, which are usually external. This means that the external hyperlinks are heavily used by benign web pages, as shown in the analysis of HTML features in Section 7.4, which might be the reason why the feature is this important.

8.17.4 Neural network feature importance

I employed a gradient-based method to determine the importance of features in a neural network classifier. This method involves calculating the gradients of the network output concerning the input features to estimate their importance. A bigger absolute value of gradient indicates a more substantial impact on the output. Figure 8.19 shows the obtained feature importance.

The features `formHttp` and `formHash` are similar to the feature `maliciousForm`. These features represent the number of HTML forms that send the data from the form to an external link or character #. This action is considered malicious and might be important in determining if the login form is malicious or benign.

The following feature is called `escape`. This feature represents the number of callings of the JavaScript method escape. This method is considered suspicious as it can be used to either encrypt or obfuscate the source code of a web page. This tactic is supposed to hide the code and make the analysis of the code impossible. This feature might be important as using this method in combination with a large average word length can uncover such tactics.

The feature `cssInternal` represents the number of link tags with attribute rel set to stylesheet used to obtain style sheets from an internal source. This feature is included because benign pages tend to use their own style imported from the internal files. This might be the reason why this feature is so important.

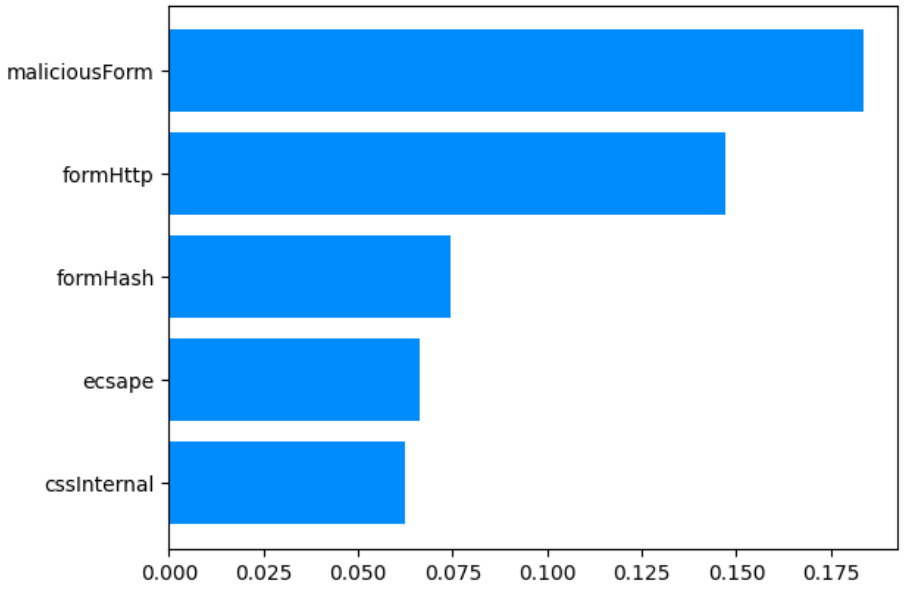

Figure 8.19: Five most important Neural network features according to gradient-based method.

Chapter 9

Experiments

This chapter introduces experiments that were carried out to find out more information about behavior and possible improvements of tuned classifiers. It also provides an analysis and comparison of the performance of each classifier.

In experiments, the classifiers were tested on 2174 previously unseen web pages. Figure 9.1 shows the content of this dataset. Some experiments also utilize the testing data from the training dataset. This testing sample is composed of 6297 pages and is imbalanced.

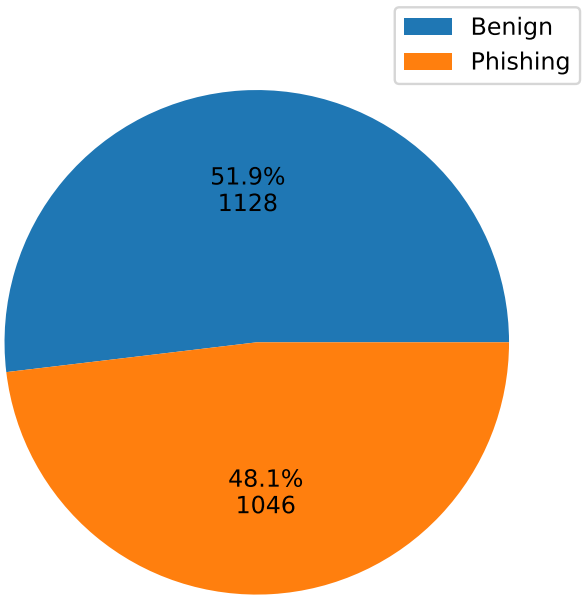

Figure 9.1: Dataset with unseen data.

9.1 Performance on unseen data

This experiment aimed to evaluate the performance of tuned models in a non-training environment to see if these classifiers are usable in a realistic environment. These metrics carry out the evaluation: balanced accuracy score, false positive rate, false negative rate, and ROC-AUC curve. This experiment also verifies if the goal of creating reliable and accurate classifiers was accomplished. In the experiment, the tuned classifiers made predictions on pages from the unseen dataset displayed in Figure 9.1. Table 9.1 shows the results of the experiment.

Algorithm	Balanced accuracy	False positive rate	False negative rate
XGBoost	97.03%	2.22%	3.73%
LightGBM	96.55%	2.30%	4.59%
SVM	90.74%	7.54%	10.99%
NN	95.40%	4.52%	4.68%

Table 9.1: Results of the experiment.

Although LightGBM was the best-performing classifier during training, it did not perform the best on the unseen data. In fact, the Neural Networks model managed to catch up to the LightGBM model despite having much worse performance during training. However, the XGBoost classifier outperformed all other classifiers, achieving a balanced accuracy score of 97.03%. On the other hand, the results of the SVM model were dissatisfactory. The classifier performed worse than during training and proved less reliable than other classifiers. The SVM classifier achieved a balanced accuracy score of only 90.74%. The false negative and false positive rates of this classifier were also much higher than during training. Except for the SVM classifier, all classifiers achieved a greater balanced accuracy score and lower false positive and false negative rates than during training.

Figure 9.2 displays the ROC curve of all classifiers calculated on unseen data. The AUC is yet again a value close to 1 in all classifiers. However, the SVM classifier achieved lower AUC than during training, which makes it less reliable. Nevertheless, the three best-performing classifiers can differentiate between the two classes very well, even while predicting previously unseen data.

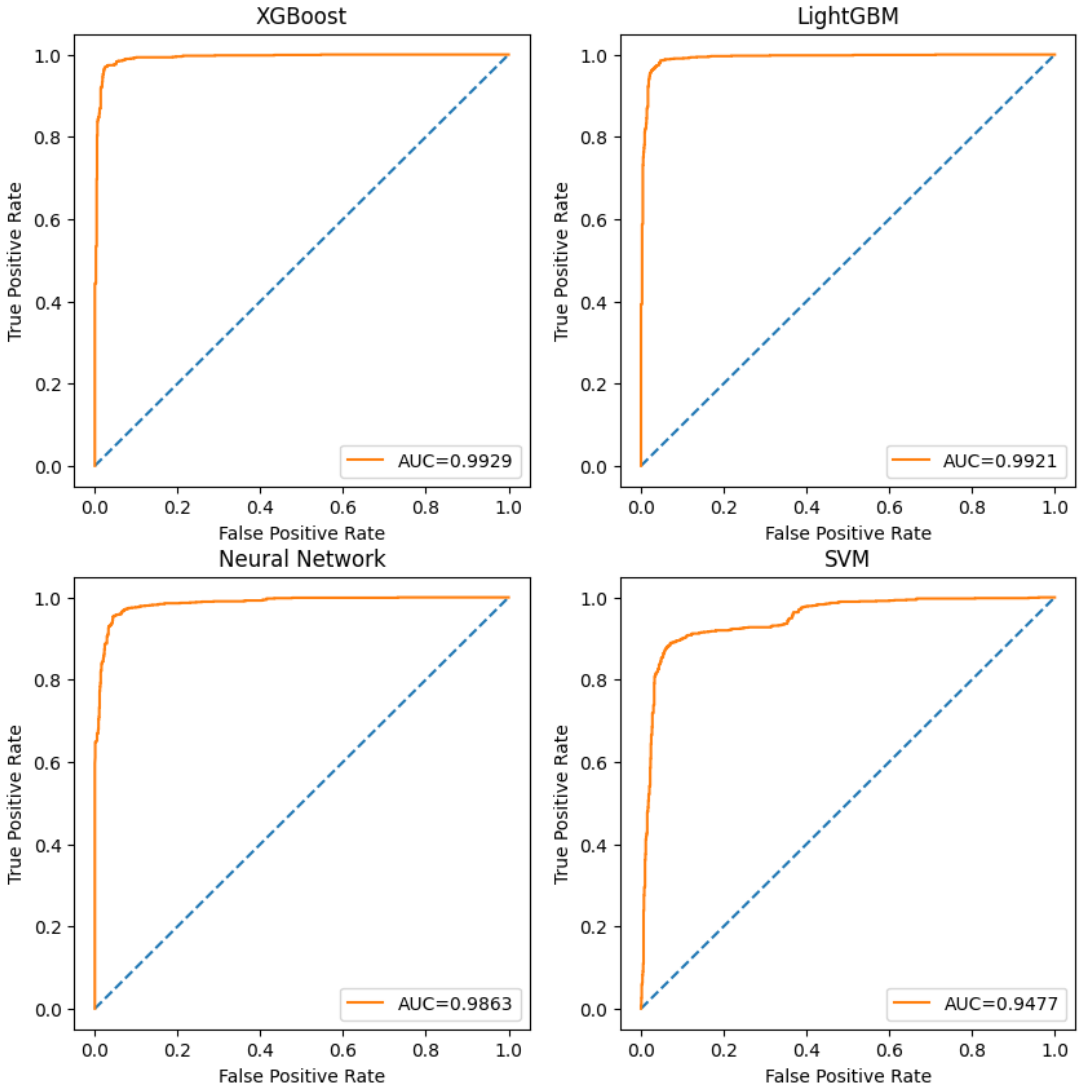

Figure 9.2: ROC-AUC curves on unseen data.

The results of LightGBM, XGBoost, and Neural network classifiers were even better than during training. Based on this experiment, it can be stated that these three classifiers are all capable of making reliable and consistent predictions in a non-training environment. On the contrary, the SVM classifier performed worse than during training and did not prove that it can make reliable and correct predictions in the non-training environment. This experiment established that the XGBoost, LightGBM, and Neural network models achieved the goal of making reliable and accurate predictions.

9.2 Performance with tuned threshold

In binary classification, the threshold is utilized to convert the model scoring into a prediction. [27] declares that using the default threshold value 0.5 for predicting outcomes might be problematic for some models, and it needs to be tuned as any other variable during model building.

This experiment aims to possibly improve the classifiers by uncovering the impact of such tuning on created classifiers while making predictions on training and unseen data introduced at the beginning of this chapter. [4] introduces a method that I will apply in this experiment. This method looks for an optimal threshold for the precision-recall curve. The method calculates class labels from probability predictions using different thresholds and calculates precision and recall based on these predictions. Knowing the precision and recall makes acquiring the threshold that produces the best F1 score possible. Figure 9.3 shows the calculated F1 values and thresholds used to calculate the class labels for different machine learning algorithms. The dots on the F1 values mark the spot where the F1 score is the highest, which also marks the threshold this method seeks.

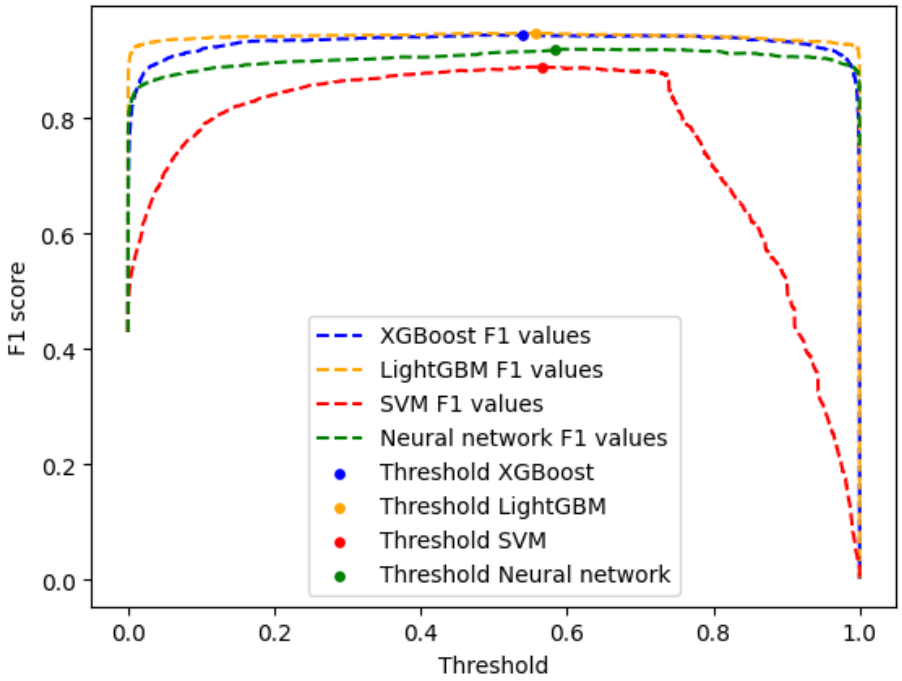

Figure 9.3: Threshold tuning.

Tables 9.2 and 9.3 display the effect that threshold tuning had on the testing sample of training data. Thanks to this tuning, every classifier managed to increase its F1 score while making predictions on the testing sample of training data. And in most cases, even the balanced accuracy score.

Algorithm	Threshold	F1 score	Balanced accuracy	Precision	Recall
XGBoost	0.5	94.49%	96.36%	93.98%	95.01%
LightGBM	0.5	94.70%	96.37%	94.61%	94.78%
SVM	0.5	87.43%	93.03%	81.74%	93.97%
NN	0.5	91.41%	94.82%	88.88%	94.08%

Table 9.2: Values for training data with default threshold.

Algorithm	Threshold	F1 score	Balanced accuracy	Precision	Recall
XGBoost	0.539037	94.50%	96.28%	94.28%	94.72%
LightGBM	0.557782	94.79%	96.33%	95.10%	94.49%
SVM	0.566930	88.97%	93.13%	86.39%	91.71%
NN	0.5839758	92.05%	94.90%	90.76%	93.39%

Table 9.3: Values for training data with tuned threshold.

However, this change had the opposite impact on predictions made on previously unseen data, as displayed in Tables 9.4 and 9.5. The balanced accuracy score of models with tuned threshold got worse, and so did the F1 score, the only exception being the LightGBM classifier, which got slightly better.

Algorithm	Threshold	F1 score	Balanced accuracy	Precision	Recall
XGBoost	0.5	96.92%	97.03%	97.58%	96.27%
LightGBM	0.5	96.43%	96.55%	97.46%	95.41%
SVM	0.5	90.30%	90.74%	91.63%	89.01%
NN	0.5	95.22%	95.40%	95.13%	95.32%

Table 9.4: Values for unseen data with default threshold.

Algorithm	Threshold	F1 score	Balanced accuracy	Precision	Recall
XGBoost	0.539037	96.82%	96.93%	97.57%	96.08%
LightGBM	0.557782	96.47%	96.60%	97.56%	95.41%
SVM	0.566930	89.63%	90.27%	93.65 %	85.95%
NN	0.5839758	94.71%	94.91%	95.35%	94.07%

Table 9.5: Values for unseen data with tuned threshold.

This experiment shows that, while the threshold tuning proved to be beneficial during the training phase in the non-training environment, this change did not bring any benefit to most of the classifiers. Based on the results of this experiment, the classifiers will keep their default threshold values.

9.3 Majority voting

[3] introduces voting as the fundamental ensemble learning method. The voting technique is a powerful tool that harnesses the strengths of multiple classifiers to achieve superior performance and improve overall quality. This experiment aims to possibly improve the predictions by combining the three best-performing classifiers: XGBoost, LightGBM, and Neural networks. The experiment tries to achieve this by utilizing the hard voting method on training and unseen data. Hard voting or majority voting involves collecting the predictions made by each tuned model and selecting the class label that receives the most votes as the prediction [3]. Figure 9.4 compares the F1 scores of tuned classifiers and majority voting classifier applied to training data. The majority voting technique managed to improve the predictions of classifiers and overall increased the F1 score.

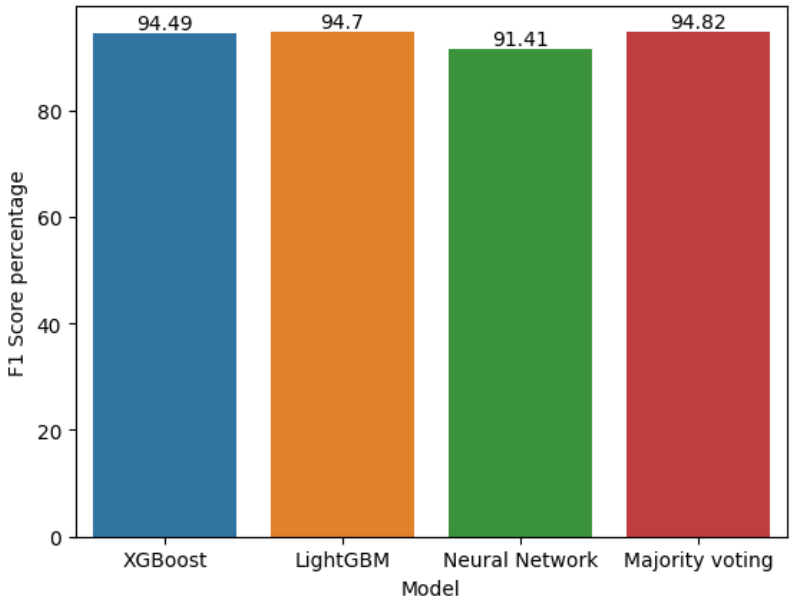

Figure 9.4: Majority voting results on training data.

Figure 9.5 displays the comparison of balanced accuracy scores of classifiers that made predictions on unseen data. The majority voting results are still great, but the XGBoost classifier outperformed this method. However, overall, the voting technique performed very well on both training and unseen data, and this method is an intriguing alternative to ordinary classifiers.

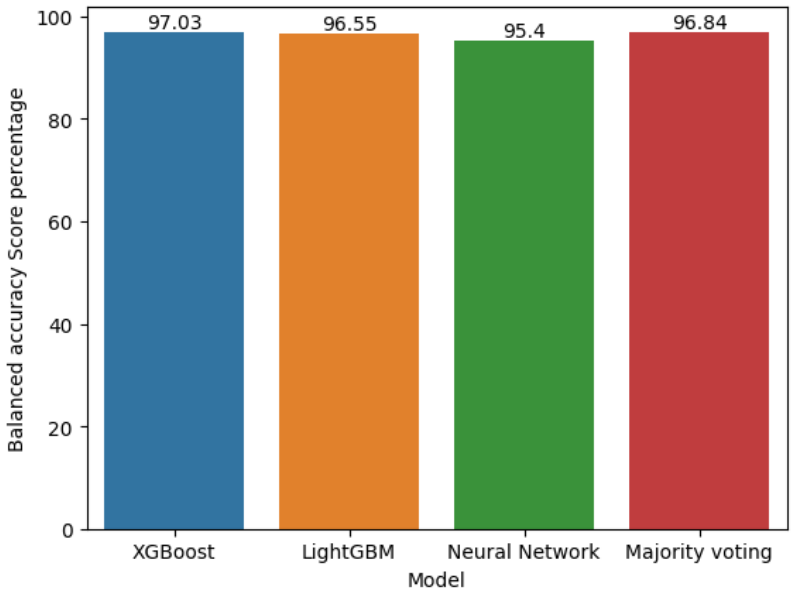

Figure 9.5: Majority voting results on unseen data.

9.4 Soft voting

Soft voting is a technique that takes into account the probabilities of predictions instead of class labels. Therefore, this technique should be more sensitive than hard voting. This experiment aims to apply soft voting to the training and unseen data and compare the results of this technique to the previously achieved scores of hard voting. The experiment tries to determine if there is a significant benefit to using probability predictions instead of class labeling predictions. The three best-performing classifiers once more carry out the voting in this experiment. Figure 9.6 displays the comparison of F1 scores achieved by the soft voting and hard voting techniques carried out on training data. In this case, the majority voting outperformed the soft voting.

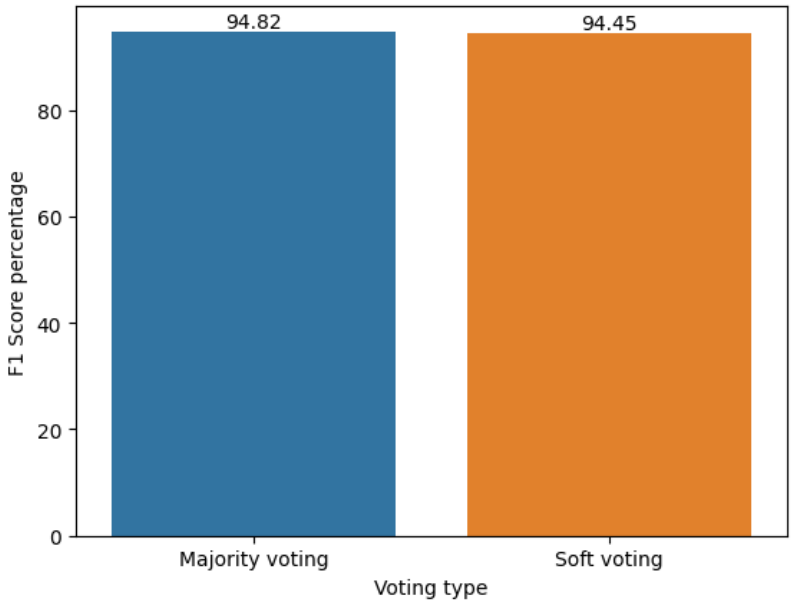

Figure 9.6: Voting types on training data.

However, Figure 9.7 presents the experiment on the unseen dataset. This time, taking the confidence of each classifier into account benefited the classification and improved the balanced accuracy score. However, both methods had very similar performance, and the results of experiments indicate that there is not any significant benefit in using probability predictions.

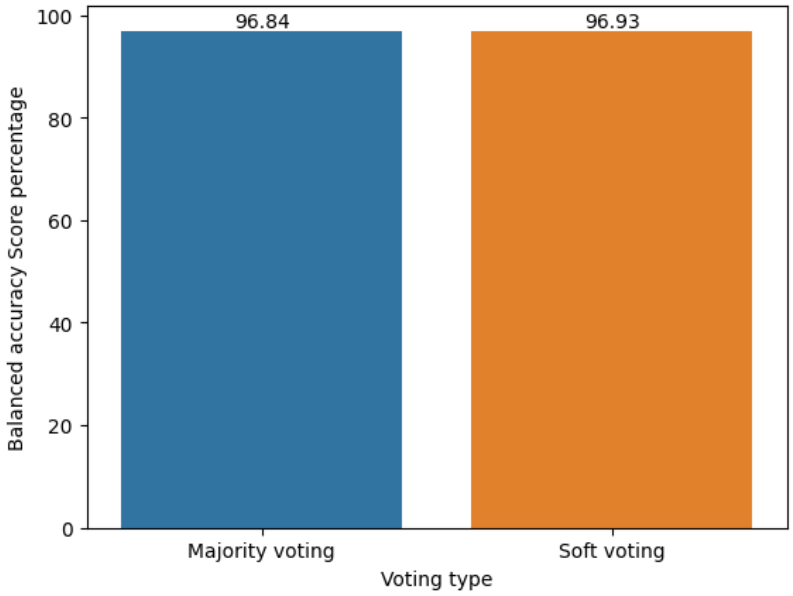

Figure 9.7: Voting types on unseen data.

9.5 Summary of achieved results

The first experiment tries to uncover any possible faults caused during the training and tuning of models by making predictions on previously unseen data, simulating a non-training environment. The results revealed that the XGBoost classifier performed best while predicting the previously unseen data, achieving a balanced accuracy score of 97.03% and a low false positive rate of 2.22%. LightGBM and Neural network classifiers also performed even better than during the training phase. This experiment concluded that these classifiers proved reliable and accurate. On the other hand, the SVM classifier performed worse than during training and failed this test, making it unreliable. This test established which tuned classifiers will be utilized in the web extension to detect phishing web pages.

The second experiment introduced the concept of threshold tuning and analyzed the impact this tuning has on models. This experiment tries to improve the classifiers. However, the results proved drastically different. While the threshold tuning was beneficial for predictions made on training data in the non-training environment, it had the opposite effect. The experiment concluded that this tuning was not beneficial for the classifiers. I found this surprising as I expected some classifiers to benefit from this tuning significantly.

The final two experiments explore the concept of voting in binary classification. These experiments aim to improve the quality of predictions by combining the strength of the best-performing classifiers. Both approaches to voting proved to be beneficial. However, I expected soft voting to outperform the majority voting approach completely. This was not the case, as both approaches had very similar results. These experiments improved the quality of predictions, and the majority voting technique performed well enough to be included in the web extension that detects phishing web pages.

Chapter 10

Web extension

This chapter introduces a web extension for the Chrome browser and experimentally verifies the performance of classifiers in practice. This web extension is a proof-of-concept implementation that detects phishing web pages by utilizing the XGBoost model, established as the best-performing model in the non-training environment, or the majority voting technique, which employs the three best-performing classifiers. The assignment of this thesis did not require the creation of this extension, but I decided to implement it to demonstrate the usability of the created classifiers in practice.

10.1 Functionality

Figure 10.1 exhibits the functionality of the web extension. The extension is composed of two parts front end and back end. The front end interacts with the user and waits for a prompt that starts the process of analyzing the web page. After the prompt is given, the front end extracts the URL where the user is currently located. This URL is passed to the back end, which scrapes the given URL. After acquiring the HTML and JavaScript code of the given URL, the acquired code is then parsed, and discovered features are assembled and fed to the classifier. After obtaining the necessary feature vector, the classifier makes a prediction, and the back end sends it back to the front end. After receiving the information, the front end displays the prediction to the user.

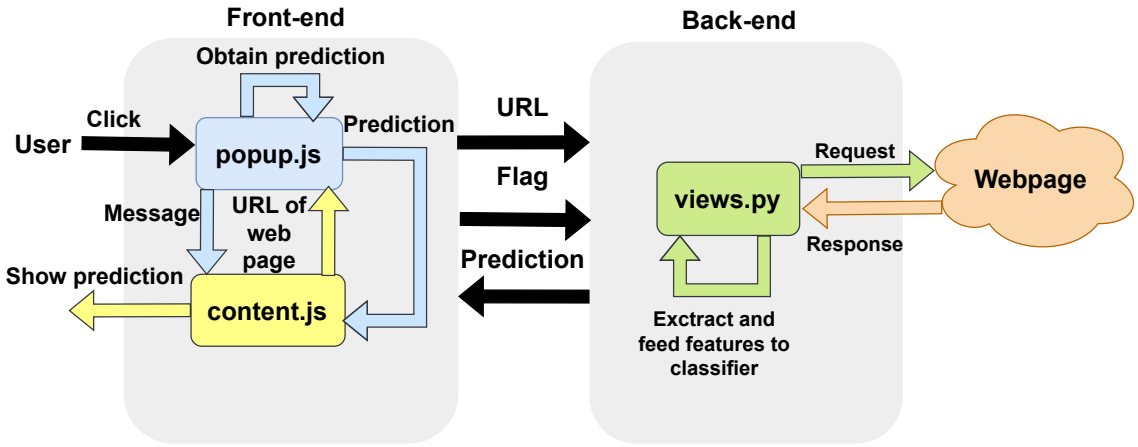

Figure 10.1: Schema of web extension.

10.2 Front end implementation

The front end consists of three files *popup.js*, *popup.html*, and *content.js*. File *popup.html* creates a user interface. The user interface is managed by file *popup.js*, which is responsible for catching the click of a user and sending a message that will be caught by the listener implemented in file *content.js*. *Content.js* can catch one of two messages either `getUrl` or `getUrlMulti`. These messages specify if the user chose the single classifier or majority voting option to make predictions. After getting the message, the listener will acquire the URL of the current web page. After this, it will send the URL and flag multi back to the listener in file *popup.js*. The flag multi is utilized to inform the back end about the type of classification to perform. The listener in file *popup.js* will then send the acquired URL and flag multi to the back end on the endpoint *'/req/?URL=&multi='*, which will trigger the scraping process. After all processes in the back end finish, the back end will return a prediction to the listener in *popup.js*. This listener will then return the prediction to the listener in *content.js* that will display this prediction to the user.

10.3 Back end implementation

After receiving the URL on the endpoint, the function *getUrl* in module *views.py* starts scraping the web page. HTML and JavaScript code is scraped similarly to methods described in Subsections 4.5.2 and 4.5.3. The concurrent approach that *Asyncio* provides makes the scraping very fast, even for web pages with lots of external JavaScript pages that need to be scraped as well. After obtaining the code of a web page, the back end parses the acquired data. The extraction of features is precisely the same as described in Chapter 6. When extraction is finished, the features are used by one of two classification methods depending on which method the user selected. If the user selects the single classifier prediction, the prediction is carried out by the XGBoost model. On the other hand, the multi-classifier prediction is made by LightGBM, XGBoost, and Neural network models. The models vote, and the most popular answer is returned to the front end.

10.4 Experiment

This small-scale experiment aims to show the practical use of created classifiers. By utilizing the web extension, I employed the XGBoost and majority voting classifiers to make predictions in a real environment. Table 10.1 shows that the predictions were made on benign login and phishing web pages. The predictions were made on popular phishing targets such as Facebook or Instagram, as well as on pages known only locally, such as Tatrabanka. The results demonstrate reliable and accurate predictions by both classifiers. The experiment concludes that the classifiers can function even in a real environment.

URL	XGBoost	Voting	Label
https://www.facebook.com/	0	0	0
https://www.instagram.com/	0	0	0
https://moja.tatrabanka.sk/html-tb/	0	0	0
https://github.com/login	0	0	0
https://important39.weebly.com/	1	1	1
https://ib.raiffeisen.sk/m/	0	0	0
https://web.telegram.org/a/	0	0	0
https://discord.com/login	0	0	0
https://vhanne-ranjeet.github.io/Netflix-Project/	1	1	1
https://www.netflix.com/sk/	0	0	0
https://sign-in-att-109479.weeblysite.com/	1	1	1
https://web.telegrenn.com/	1	1	1
https://im-token.ltd/	1	1	1
https://sp.t1skins.com/	1	1	1
https://www.dropbox.com/login	0	0	0
https://www.linkedin.com/login	0	0	0
https://mail.g00ggle.workers.dev/	1	1	1
https://service.qoll.workers.dev/	1	1	1
https://www.vut.cz/login/studis	0	0	0
https://yhgh.pages.dev/	1	1	1
https://online.mbank.sk/sk/Login	0	0	0
https://vcfs.pages.dev/	1	1	1
https://mail-104105.weeblysite.com/	1	1	1
http://www.drgebfish.com/M3mail@b.c	1	1	1
https://new.tollsk.info/	1	1	1

Table 10.1: Results of experiment.

Chapter 11

Conclusion

The purpose of this bachelor's thesis was to create a classifier capable of detecting phishing web pages based on the analysis of their HTML and JavaScript code to help fight phishing attacks and possibly decrease the damage that these attacks cause.

Within this work, I studied related research, different machine learning algorithms, and trustworthy data sources. Moreover, I have implemented a data gathering program, which collected HTML and JavaScript code of phishing and benign web pages. Later, I introduced a feature vector and extracted features from scraped HTML and JavaScript code. After this, I filtered the scraped data. This led to the creation of a dataset composed of features from 31481 web pages, ready to be utilized for model training. Then, I used this dataset to create four tuned classifiers, each one created by a different machine learning algorithm. Moreover, I experimentally measured and compared the performance of each tuned classifier by employing suitable evaluation metrics. The satisfactory performance of the tuned classifiers, where the best-performing classifier achieved a balanced accuracy score of 97.03% while making predictions on previously unseen data, indicates that the detection of phishing web pages by utilizing machine learning and analysis of HTML and JavaScript code is a successful strategy to fight phishing web pages. By experimenting, I also verified the benefits of methods utilized to improve the performance of classifiers, such as tuning the threshold or utilizing the majority and soft voting techniques. The majority and soft voting of classifiers proved to be beneficial methods, which successfully combined the strength of multiple classifiers to increase the quality of predictions.

Finally, I implemented a web extension capable of warning users about suspected phishing web pages, which employs the majority voting method and utilizes the best-performing models for practical use. This web extension also supports a standard classification carried out by the XGBoost model. The classifiers were also tested in a real environment by utilizing this web extension, proving that the classifiers can reliably differentiate between benign and phishing pages even in practice.

In the future, the work could be improved by utilizing the URL-based features obtained by analyzing the URL of a web page. Another possible improvement is the analysis of visual elements, such as screenshots of a web page, which could help classify the web page by trying to find a logo of brands frequently targeted by phishing attackers.

Bibliography

[1] Agarwal, D. *Introduction Support Vector Machines (SVM) with Python Implementation* [online]. 2024 [cit. 2024-3-22]. Available at: https://www.analyticsvidhya.com/blog/2021/04/insight-into-svm-support-vector-machine-along-with-code/.

[2] Agregado, K. *It's Raining Phish and Scams – How Cloudflare Pages.dev and Workers.dev Domains Get Abused* [online]. 2023 [cit. 2023-12-26]. Available at: https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/its-raining-phish-and-scams-how-cloudflare-pages-dev-and-workers-dev-domains-get-abused/.

[3] Awan Ur Rahman. *Understanding Soft Voting and Hard Voting: A Comparative Analysis of Ensemble Learning Methods* [online]. 2023 [cit. 2024-4-2]. Available at: https://medium.com/@awanurrahman.cse/understanding-soft-voting-and-hard-voting-a-comparative-analysis-of-ensemble-learning-methods-db0663d2c008/.

[4] Brownlee, J. *A Gentle Introduction to Threshold-Moving for Imbalanced Classification* [online]. 2021 [cit. 2024-3-28]. Available at: https://machinelearningmastery.com/threshold-moving-for-imbalanced-classification/.

[5] Brownlee, J. *A Gentle Introduction to XGBoost for Applied Machine Learning* [online]. 2021 [cit. 2023-12-22]. Available at: https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/.

[6] Brownlee, J. *How to Choose an Activation Function for Deep Learning* [online]. 2021 [cit. 2024-3-26]. Available at: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/.

[7] Center, C. I. *Phishing Activity in Top-level Domains (TLDs) November 1, 2022 - January 31, 2023* [online]. 2023 [cit. 2023-12-26]. Available at: https://www.cybercrimeinfocenter.org/phishing-activity-in-tlds-november-january-2023.

[8] Chauhan, N. S. *Optimization Algorithms in Neural Networks* [online]. 2020 [cit. 2024-3-26]. Available at: https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html.

[9] Chen, K.-T., Chen, J.-Y., Huang, C.-R. and Chen, C.-S. Fighting phishing with discriminative keypoint features. *IEEE Internet Computing*. IEEE. 2009, vol. 13, no. 3, p. 56–63.

[10] Das Guptta, S., Shahriar, K. T., Alqahtani, H., Alsalman, D. and Sarker, I. H. Modeling hybrid feature-based phishing websites detection using machine

learning techniques. *Annals of Data Science*. Springer. 2024, vol. 11, no. 1, p. 217–242. DOI: 10.1007/s40745-022-00379-8.

[11] Dung, P. T. *Phishing Website Detection Dataset*. 2022. DOI: 10.17632/kvpkc4j658.1.

[12] Educative, I. *What is balanced accuracy?* [online]. 2024 [cit. 2024-4-30]. Available at: https://www.educative.io/answers/what-is-balanced-accuracy.

[13] Egelman, S., Cranor, L. F. and Hong, J. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In: *Proceedings of the SIGCHI conference on human factors in computing systems*. 2008, p. 1065–1074.

[14] Guo, Y., Liu, Y., Oerlemans, A., Lao, S., Wu, S. et al. Deep learning for visual understanding: A review. *Neurocomputing*. 2016, vol. 187, p. 27–48. DOI: https://doi.org/10.1016/j.neucom.2015.09.116. ISSN 0925-2312. Recent Developments on Deep Big Vision. Available at: https://www.sciencedirect.com/science/article/pii/S0925231215017634.

[15] Hara, M., Yamada, A. and Miyake, Y. Visual similarity-based phishing detection without victim site information. In: IEEE. *2009 IEEE Symposium on Computational Intelligence in Cyber Security*. 2009, p. 30–36.

[16] Hernandez, D. *Binary Classification with Neural Networks using Tensorflow & Keras Pt.2* [online]. 2024 [cit. 2024-3-25]. Available at: https://python.plainenglish.io/binary-classification-with-neural-networks-using-tensorflow-keras-%EF%B8%8F-pt-2-6831978765cb.

[17] Hou, Y.-T., Chang, Y., Chen, T., Laih, C.-S. and Chen, C.-M. Malicious web content detection by machine learning. *Expert systems with applications*. Elsevier. 2010, vol. 37, no. 1, p. 55–60. DOI: https://doi.org/10.1016/j.eswa.2009.05.023. ISSN 0957-4174.

[18] Janiesch, C., Zschech, P. and Heinrich, K. Machine learning and deep learning. *Electron Markets*. 2021, vol. 31, p. 685–695. DOI: https://doi.org/10.1007/s12525-021-00475-2.

[19] Khonji, M., Iraqi, Y. and Jones, A. Phishing detection: a literature survey. *IEEE Communications Surveys & Tutorials*. IEEE. 2013, vol. 15, no. 4, p. 2091–2121.

[20] Kumaraguru, P., Rhee, Y., Acquisti, A., Cranor, L. F., Hong, J. et al. Protecting people from phishing: the design and evaluation of an embedded training email system. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. 2007, p. 905–914.

[21] Labelf. *What is Accuracy, Precision, Recall and F1 Score?* [online]. 2022 [cit. 2024-3-20]. Available at: https://www.labelf.ai/blog/what-is-accuracy-precision-recall-and-f1-score.

[22] learn, S. *Permutation feature importance* [online]. 2024 [cit. 2024-4-25]. Available at: https://scikit-learn.org/stable/modules/permutation_importance.html.

[23] LI, W. and LIU, Z. A method of SVM with Normalization in Intrusion Detection. *Procedia Environmental Sciences*. 2011, vol. 11, p. 256–262. DOI: https://doi.org/10.1016/j.proenv.2011.12.040. ISSN 1878-0296. 2011 2nd International Conference on Challenges in Environmental Science and Computer Engineering (CESCE 2011). Available at: https://www.sciencedirect.com/science/article/pii/S1878029611008632.

[24] LIGHTGBM. *Parameters Tuning* [online]. 2024 [cit. 2024-3-19]. Available at: https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html.

[25] LIU, D. and LEE, J.-H. CNN Based Malicious Website Detection by Invalidating Multiple Web Spams. *IEEE Access*. 2020, vol. 8, p. 97258–97266. DOI: 10.1109/ACCESS.2020.2995157.

[26] MAHAPATRA, A. *[ML basics][Regression] How to tell if a dataset is linear or not?* [online]. 2019 [cit. 2024-3-23]. Available at: https://medium.com/@abhinav.mahapatra10/ml-basics-regression-how-to-tell-if-a-dataset-is-linear-or-not-594a4f1e8aaf.

[27] MALATO, G. *Are you still using 0.5 as a threshold?* [online]. 2021 [cit. 2024-3-28]. Available at: https://www.yourdatateacher.com/2021/06/14/are-you-still-using-0-5-as-a-threshold/.

[28] MANANDHAR, G. *How to run MongoDB with Docker and Docker Compose a Step-by-Step guide* [online]. 2023 [cit. 2024-5-3]. Available at: https://geshan.com.np/blog/2023/03/mongodb-docker-compose/.

[29] MANDOT, P. *What is LightGBM, How to implement it? How to fine tune the parameters?* [online]. 2017 [cit. 2023-12-23]. Available at: https://medium.com/@pushkarmandot/https-medium-com-pushkarmandot-what-is-lightgbm-how-to-implement-it-how-to-fine-tune-the-parameters-60347819b7fc.

[30] MBAABU, O. *Introduction to Random Forest in Machine Learning* [online]. 2020 [cit. 2023-12-22]. Available at: https://www.section.io/engineering-education/introduction-to-random-forest-in-machine-learning/.

[31] MCGAHAGAN, J., BHANSALI, D., PINTO COELHO, C. and CUKIER, M. A Comprehensive Evaluation of Webpage Content Features for Detecting Malicious Websites. In: *2019 9th Latin-American Symposium on Dependable Computing (LADC)*. 2019, p. 1–10. DOI: 10.1109/LADC48089.2019.8995713. ISBN 978-1-7281-6622-3.

[32] MEDCALC. *ROC curve analysis* [online]. 2024 [cit. 2024-4-30]. Available at: https://www.medcalc.org/manual/roc-curves.php.

[33] MORARU, A. and DONAHUE, P. R. *Top 50 most impersonated brands in phishing attacks and new tools you can use to protect your employees from them* [online]. 2023 [cit. 2023-12-26]. Available at: https://blog.cloudflare.com/50-most-impersonated-brands-protect-phishing.

[34] NAIM, O., COHEN, D. and BEN GAL, I. Malicious website identification using design attribute learning. *International Journal of Information Security*. 2023, vol. 22, p. 1207–1217. DOI: https://doi.org/10.1007/s10207-023-00686-y.

[35] Nvidia. *XGBoost* [online]. 2023 [cit. 2023-12-22]. Available at: https://www.nvidia.com/en-us/glossary/data-science/xgboost/.

[36] Paganini, P. *CROOKS ABUSE GITHUB PLATFORM TO HOST PHISHING KITS* [online]. 2019 [cit. 2023-12-26]. Available at: https://securityaffairs.com/84495/hacking/github-hosting-phishing-kits.html.

[37] Plugge, E., Hows, D., Membrey, P. and Hawkins, T. *The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB*. Apress, 2015. ISBN 1484211820, 9781484211823.

[38] Radar, S. *Top 10 TLDs Threat Actors Use for Phishing* [online]. 2022 [cit. 2023-12-26]. Available at: https://socradar.io/top-10-tlds-threat-actors-use-for-phishing/.

[39] Saini, A. *AdaBoost Algorithm: Understand, Implement and Master AdaBoost* [online]. 2023 [cit. 2023-12-22]. Available at: https://www.analyticsvidhya.com/blog/2021/09/adaboost-algorithm-a-complete-guide-for-beginners/.

[40] Saleem, S. *Neural Networks in 10mins. Simply Explained!* [online]. 2023 [cit. 2024-4-8]. Available at: https://medium.com/@sadafsaleem5815/neural-networks-in-10mins-simply-explained-9ec2ad9ea815.

[41] science, M. in data. *What Is a Decision Tree?* [online]. 2023 [cit. 2023-12-21]. Available at: https://www.mastersindatascience.org/learning/machine-learning-algorithms/decision-tree/.

[42] Sheng, S., Wardman, B., Warner, G., Cranor, L., Hong, J. et al. An empirical analysis of phishing blacklists. Carnegie Mellon University. 2009.

[43] Solomon, B. *Async IO in Python: A Complete Walkthrough* [online]. 2023 [cit. 2023-11-24]. Available at: https://realpython.com/async-io-python/.

[44] Tabsharani, F. *Support vector machine (SVM)* [online]. 2023 [cit. 2023-12-22]. Available at: https://www.techtarget.com/whatis/definition/support-vector-machine-SVM.

[45] Team, E. A. *How to explain the ROC curve and ROC AUC score?* [online]. 2024 [cit. 2024-4-30]. Available at: https://www.evidentlyai.com/classification-metrics/explain-roc-curve.

[46] Team, T. . *Introduction to Decision Trees: Why Should You Use Them?* [online]. 2023 [cit. 2023-12-21]. Available at: https://365datascience.com/tutorials/machine-learning-tutorials/decision-trees/.

[47] Techs, W. *Usage statistics of top level domains for websites* [online]. 2023 [cit. 2023-12-26]. Available at: https://w3techs.com/technologies/overview/top_level_domain.

[48] Vidhya, A. *Introduction to XGBoost Algorithm in Machine Learning* [online]. 2021 [cit. 2023-12-22]. Available at: https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/.

[49] Xu, Y., Zhou, Y., Sekula, P. and Ding, L. Machine learning in construction: From shallow to deep learning. *Developments in the Built Environment*. 2021, vol. 6, p. 100045. DOI: https://doi.org/10.1016/j.dibe.2021.100045. ISSN 2666-1659. Available at: https://www.sciencedirect.com/science/article/pii/S2666165921000041.

[50] Yildirim, S. *Understanding the LightGBM* [online]. 2020 [cit. 2023-12-23]. Available at: https://towardsdatascience.com/understanding-the-lightgbm-772ca08aabfa.

[51] Zhang, Y. *New advances in machine learning*. BoD–Books on Demand, 2010. ISBN 953307034X, 9789533070346.

Appendix A

Content of attached SD

The attached SD card contains the following items:

1. `/scrapingapp` - web scraping application with jupyter notebook files and training and control datasets
2. `/webextension` - web extension source code and utilized classifiers
3. `/tex` - latex source code
4. `manual.pdf` - manual for scraping application and web extension
5. `xpolon03.pdf` - the thesis in pdf format

Appendix B

Correlation matrix of all features

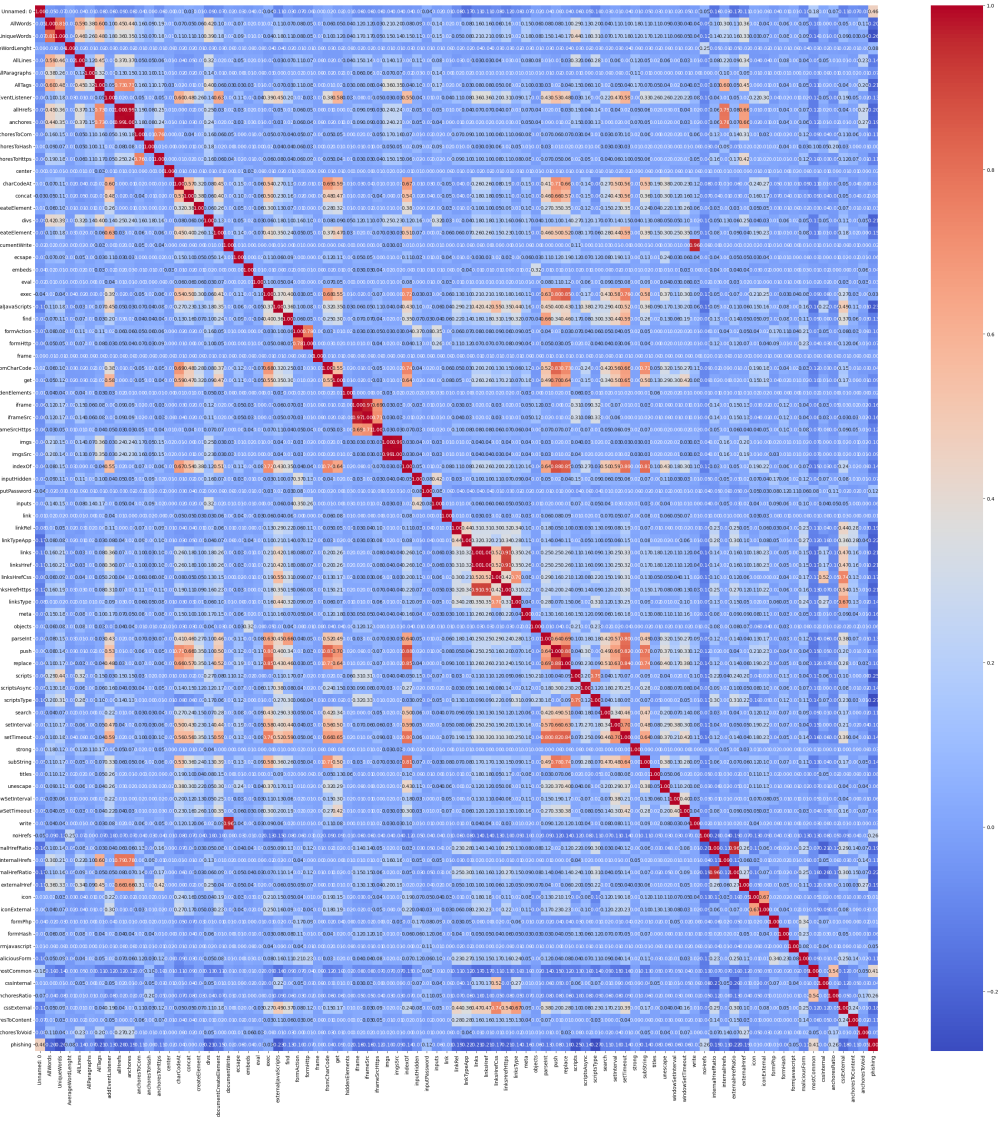

Figure B.1: Correlation matrix of all features.