Training a Malicious URL Classifier & Deploying it as a Chrome Extension

Abstract

In this project, we created a malicious URL ensemble classifier by training on 2 datasets (*Dataset of Malicious and Benign Webpages, Malicious & Benign URLs*). Our ensemble classification model is made up of 3 different models:

1. A neural network classification model that uses Natural Language Processing (NLP) and takes in as input an array of content strings.
2. A random forest classification model.
3. A gradient boosting classification model.

Subsequently, we deployed the malicious URL classifier via a Chrome extension.

Lastly, the effectiveness of our malicious URL classifier was evaluated on a third dataset (*Malicious URLs Dataset*) which was released around 2 - 3 months ago (24 July 2021). Our model achieved an overall accuracy of 71.45%.

Introduction

Technology has significantly changed our lives; one thing that cannot be denied is the fact that technology is so intertwined in our lives that we sometimes do not even realise how much it has helped us. However, one also needs to be cautious as technology can be a double-edged sword. This thus gave rise to the concept of cybersecurity.

Cybersecurity is the protection of all categories of data from theft and damage. This includes sensitive data, personal information, intellectual property, data, and governmental and industry information systems [1]. With the increased digital transformation and the rise of digital ecosystems, cybersecurity has become ever more important. This is especially accelerated with the COVID-19 pandemic, where it has completely redefined disruption and further expedited the movement towards digitalization.

As such, cyber-attacks present a growing threat to businesses, governments, and individuals. Be it from hacktivist groups or state-sponsored cyber warfare units, these attacks are of increasing concern, given their growth in scale and sophistication. Of these cyber-attacks, social engineering approaches, despite being one of the simplest, are one of the most effective ways of gaining access to sensitive and confidential information.

The United States Computer Emergency Readiness Team (US-CERT) defines phishing as a form of social engineering that uses e-mails or malicious websites to solicit personal information from an individual or company by posing as a trustworthy organization or entity [2]. Phishing often is done via malicious URLs.

While organizations should educate employees on how to identify and respond to phishing emails and links, there are readily available software, such as HTTrack, that facilitate the reproduction of websites. As a result, even trained users can still be tricked into revealing private or sensitive information when interacting with a malicious website that they believe to be legitimate.

Therefore, this shows that merely education is insufficient to guard against phishing attacks. To better protect against phishing and malicious URLs, computer-based and data-driven solutions should be implemented as it would enable a computer to detect malicious websites and protect users from interacting with them.

Background

Malicious websites have been widely used to mount various cyberattacks. Detection of these malicious websites and identification of threat types are critical to thwart these attacks. Knowing the type of a threat enables estimation of severity of the attack and helps with the adoption of effective countermeasures [3].

Being able to recognize malicious websites relies on their Uniform Resource Locators (URLs). A URL is a global address of a document in the World Wide Web, that helps to locate a document on the Internet. Even in cases where the content of websites is duplicated, the URLs can still be used to distinguish real sites from imposters [4]. Thus, a possible approach is to use a blacklist of malicious URLs developed by anti-virus groups.

Blacklists can be constructed from numerous sources, including those from human feedback. This method is highly accurate but time-consuming. Blacklisting also helps to ensure no false positives

will be experienced. However, it should be noted that the blacklist is only effective for known malicious URLs and cannot be exhaustive because new malicious URLs keep cropping up continuously. The requirement for the URLs to be an exact match results in malicious websites easily evading this security measure.

Hence, this calls for the need of approaches that can automatically classify a new, previously unseen

URL as either a phishing site or a legitimate one. To do this, machine-learning based approaches can be used. Machine-learning is a system that can categorize new phishing sites through a model developed using training sets of known attacks. This can address the weakness of blacklisting as it can detect new unknown malicious URLs. However, there are currently very few available training data sets containing phishing URLs to train appropriate machine learning models. Consequently, studies are required to evaluate the effectiveness of this approach based on the data sets that do exist.

Objective

Cybersecurity attacks present a growing threat to businesses, governments and individuals. With the accelerated pace in digital transformation due to the pandemic, the need for cybersecurity has become ever more important. In this report, we propose a method to use machine learning to detect malicious URLs as an added line of defence.

In this project, two existing datasets are identified to help train the classification model. The trained model will then be deployed to test if malicious URLs are detected via a Chrome Extension. The effectiveness of our model is assessed with a third, extremely recent dataset.

We hope that our work will help to make cyberspace a safer space for all.

Data Collection

To train our model, we will be using 2 datasets (*Dataset of Malicious and Benign Webpages, Malicious & Benign URLs*). As an additional testing set, we would use a third labelled dataset (*Malicious URLs Dataset*) solely for testing our trained model. The third dataset will only be used in the last section, Applying to Real-world URLs.

Dataset Name	**No. of Rows**	**No. of Columns**	**Source**	**Date Posted**
Dataset of Malicious and Benign Webpages	1,561,934	11	Malcrawler, Google Safe Browsing API, Mendeley Data [5, 6, 7]	4 April 2020
Malicious & Benign URLs	450,176	2	PhisTank [8]	31 May 2019
Malicious URLs Dataset	651,191	2	PhishTank, PhishStorm, ISCX-URL 2016 [9, 10]	24 July 2021

Table 1: Characteristics of the various datasets

Dataset of Malicious and Benign Webpages

This dataset has the following variables.

Variable	**Details**	**Variable**	**Details**
url	URL of webpage	tld	Top level domain of a webpage
ip_add	IP address of a webpage	who_is	Whether the WHO IS domain information is complete
geo_loc	Geographic location of the hosted webpage	https	Whether the site uses HTTP or HTTPS
url_len	Length of URL	content	Raw webpage content including JavaScript code
js_len	Length of JavaScript code on website	js_obf_len	Length of obfuscated JavaScript code
label	Benign or malicious		

Table 2: Variables of Dataset 1

Malicious & Benign URLs

This dataset has the following variables.

Variable	**Details**
URL	URL of webpage
label	Benign or malicious

Table 3: Variables of Dataset 2

Malicious URLs Dataset

Variable	Details
URL	URL of webpage
label	Benign or malicious

This dataset has the following variables.

Table 4: Variables of Dataset 3

Data Visualization

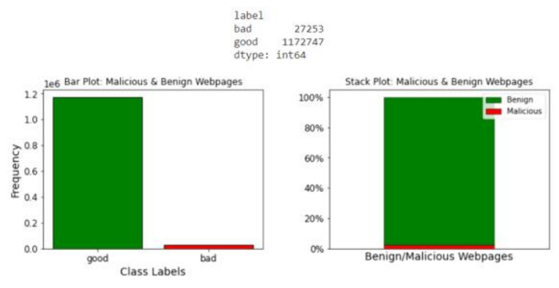
Before training our model on the datasets, we will be doing an initial exploratory analysis using data visualizations. By doing so, we will be able to better understand which features are more relevant for our models to train on. Furthermore, this will serve as a sanity check to ensure that our models are learning the correct features.

We did data visualizations on the first dataset: *Dataset of Malicious and Benign Webpages*. This was the only dataset that had features ready for us to use.

The dataset comprises 1.2 million records. There are ten attributes in the dataset, including url, ip_add, geo_loc, url_len, is_len, js_obf_len, tld, who_is, https and content, as well as the class attribute named *'label'*. We will be visualizing all of these attributes in the following sections.

Analysis of the Class Label

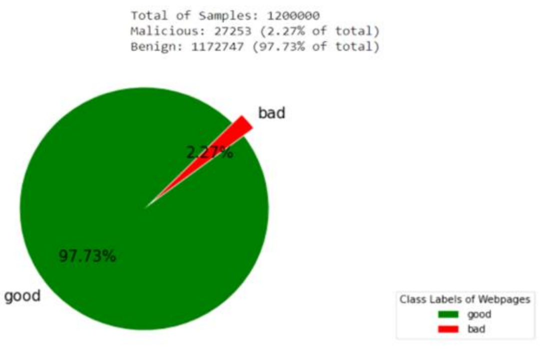
The class label is given in the last column of the table and has two values, *'good'* and *'bad'* representing benign and malicious web pages respectively. In reality, malicious web pages are few and far between as compared to benign web pages. [17] estimates that there are 100,000 malicious web pages generated daily as compared to 2 billion web pages worldwide [18]. This inequality shows in our dataset as well, since it has been scraped from the Internet. The class label and its inequality is visualised and analysed below in detail.



```
label
bad        27253
good     1172747
dtype: int64
```

Figure 1: Bar Chart of Class Label and its Inequality

Looking at the chart, we can see that our dataset consists of approximately 1.17 million web pages labelled 'good' and only approximately 27 thousand web pages labelled 'bad'.



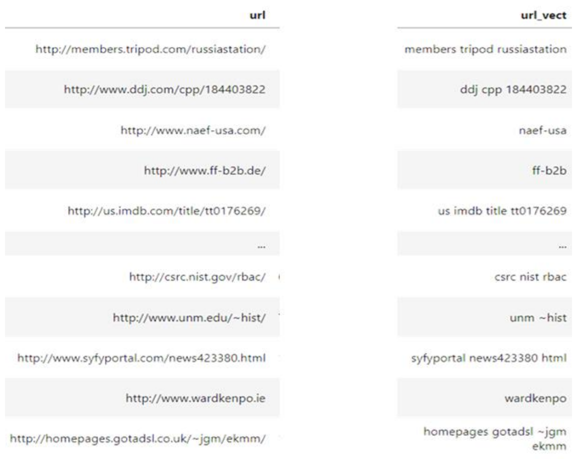
```
Total of Samples: 1200000
Malicious: 27253 (2.27% of total)
Benign: 1172747 (97.73% of total)
```

Figure 2: Pie Chart of Class Label

This corresponds to 97.73% of our dataset consisting of benign web pages while only 2.27% are malicious web pages. This is an important context to keep in mind when looking at the visualizations of attributes later.

Analysis of the 'url' attribute:

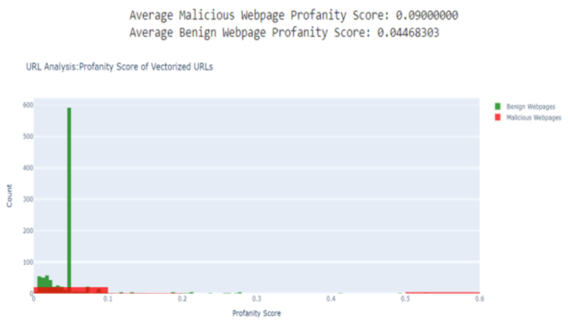
The first column of the dataset has the 'url' attribute. We first split the web page URLs into its constituent words based on the following punctuations commonly found in URLs: ? \\ . ; / \. These words are then used to generate vectorized value for each URL.



url	url_vect
http://members.tripod.com/russiastation/	members tripod russiastation
http://www.ddj.com/cpp/184403822	ddj cpp 184403822
http://www.naef-usa.com/	naef-usa
http://www.ff-b2b.de/	ff-b2b
http://us.imdb.com/title/tt0176269/	us imdb title tt0176269
...	...
http://csrc.nist.gov/rbac/	csrc nist rbac
http://www.unm.edu/~hist/	unm ~hist
http://www.syfyportal.com/news423380.html	syfyportal news423380 html
http://www.wardkenpo.ie	wardkenpo
http://homepages.gotadsl.co.uk/~jgm/ekmm/	homepages gotadsl ~jgm ekmm

Figure 3: URL and its corresponding Vector

Then, a profanity score based on good or bad words found in the URL is calculated. This score is then plotted in the graph below.



```
Average Malicious Webpage Profanity Score: 0.09000000
Average Benign Webpage Profanity Score: 0.04468303
```

Figure 4: Graph of Profanity Score

At the right of the graph where the profanity score is higher, there are mostly only malicious web pages. Meanwhile, most of the benign web pages are clustered towards the left of the graph where the profanity score is lower. The average profanity score for malicious web pages is also higher than that of benign web pages. This indicates that this could be an important feature in our dataset in determining the class label.

<u>Analysis of ‘ip_add’ and ‘ geo_loc’ attributes:</u>

The 'ip_add' and 'geo_loc' are the 2nd and 3rd columns in the dataset. The 'ip_add' is the IP address of the web server where the web page is hosted. The 'geo_loc' has been computed from the IP address using the GeoIP Database and gives the country where the IP address is located.

The country wise distribution of IP addresses of web pages in the dataset is plotted below on the world map.

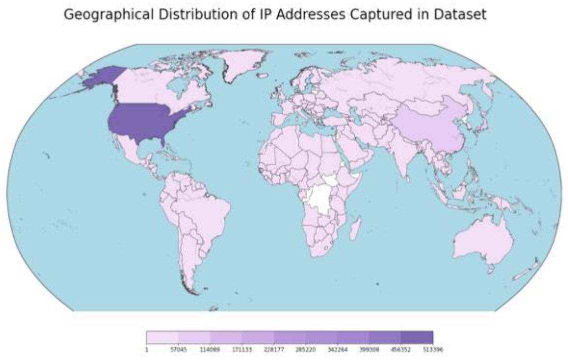

Geographical Distribution of IP Addresses Captured in Dataset

Figure 5: Distribution of IP addresses

The country wise distribution of IP addresses of malicious and benign web pages is also plotted on the two world maps below.

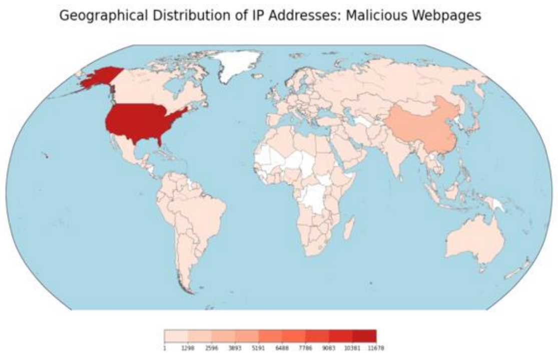

Geographical Distribution of IP Addresses: Malicious Webpages

Figure 6: Distribution of Malicious Webpages

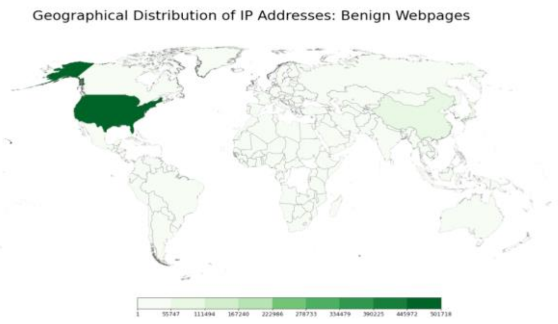


Figure 7: Distribution of Benign Webpages

As can be seen from the three maps above, the dataset covers the complete globe. Majority of the IP addresses are active in US and China, as most web servers exist in those countries. No distinct pattern differentiating malicious or benign web pages with respect to geographic location emerged.

<u>Analysis of numerical attributes: 'url_len', 'js_len' and 'js_obf_len'</u>

The 'url_len', 'js_len' and js_obf_len' are the 4th, 5th and 6th columns of the dataset respectively. Since they are numerical attributes, they have been discussed and visualised together in this section.

First, a univariate visualisation of these individual attributes will be carried out.

Analysis of 'url_len' Attribute using Univariate Plots:

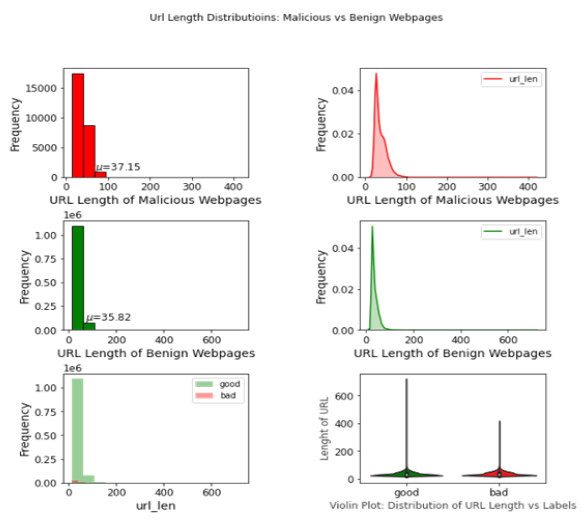


Figure 8: URL Length Distributions

As can be seen from above plots of 'url_len, the average URL length of malicious web pages (37.15) is similar to that of benign web pages (35.82). Hence, no distinct pattern emerges in distinguishing malicious and benign web pages

Analysis of ‘js_len' attribute using Univariate Plots:

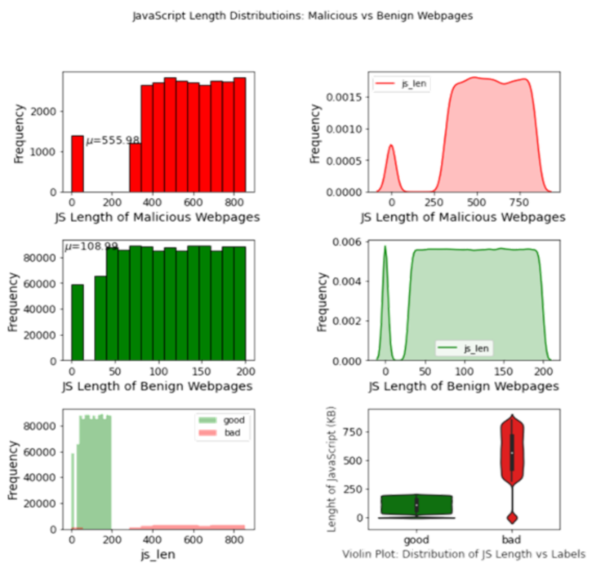


Figure 9: JavaScript Length Distribution

As seen from plots above, the average JavaScript length of malicious web pages is 555.98 KB, while that of benign web pages is smaller at 108.99 KB. Thus, a clear distinct pattern can be visualised between the 'js_len' of the two classes.

Analysis of ‘js_obf_len' attribute using Univariate Plots:

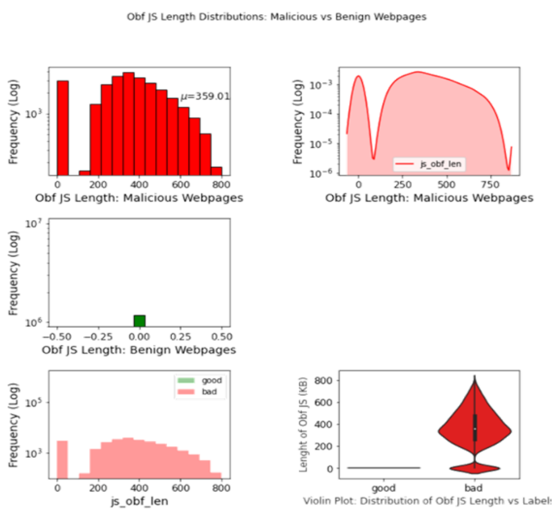
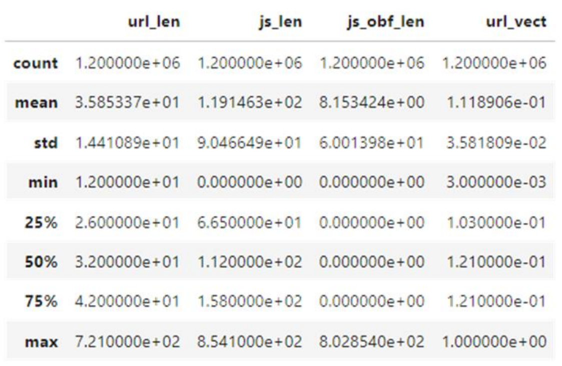


Figure 10: Obfuscated JavaScript Length Distribution

As seen from plots above, none of the benign web pages have obfuscated JavaScript code. On the other hand, malicious web pages have an average obfuscated JavaScript length of 359.01 KB. Thus, a clear pattern emerges here.

Trivariate Analysis of all three numerical attributes: 'url_len', 'js_len' & 'js_obf_len'

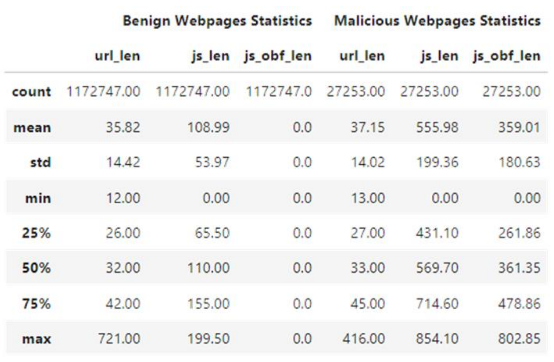
We now do a trivariate analysis of all three numerical attributes.



	url_len	js_len	js_obf_len	url_vect
count	1.200000e+06	1.200000e+06	1.200000e+06	1.200000e+06
mean	3.585337e+01	1.191463e+02	8.153424e+00	1.118906e-01
std	1.441089e+01	9.046649e+01	6.001398e+01	3.581809e-02
min	1.200000e+01	0.000000e+00	0.000000e+00	3.000000e-03
25%	2.600000e+01	6.650000e+01	0.000000e+00	1.030000e-01
50%	3.200000e+01	1.120000e+02	0.000000e+00	1.210000e-01
75%	4.200000e+01	1.580000e+02	0.000000e+00	1.210000e-01
max	7.210000e+02	8.541000e+02	8.028540e+02	1.000000e+00

Figure 11: Numerical Attributes of dataset

The statistical values of these three numerical columns are given above.



	Benign Webpages Statistics			Malicious Webpages Statistics		
	url_len	js_len	js_obf_len	url_len	js_len	js_obf_len
count	1172747.00	1172747.00	1172747.0	27253.00	27253.00	27253.00
mean	35.82	108.99	0.0	37.15	555.98	359.01
std	14.42	53.97	0.0	14.02	199.36	180.63
min	12.00	0.00	0.0	13.00	0.00	0.00
25%	26.00	65.50	0.0	27.00	431.10	261.86
50%	32.00	110.00	0.0	33.00	569.70	361.35
75%	42.00	155.00	0.0	45.00	714.60	478.86
max	721.00	199.50	0.0	416.00	854.10	802.85

Figure 12: Numerical Attributes of Benign and Malicious Webpages

When we split the values based on class as shown in the table above, a distinction emerges for the values of 'js_len' and 'js_obf_len' between the two class labels. On the other hand, the statistical values for ‘url_len’ remained similar between malicious and benign web pages.

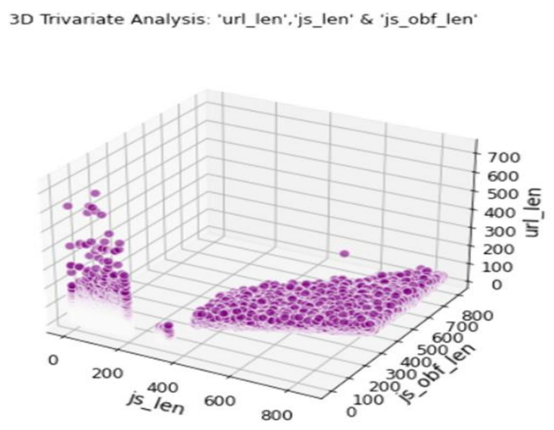


Figure 13: 3D Trivariate Analysis of Numerical Attributes

Looking at the 3D trivariate analysis, we can see a cluster that emerges between high ‘js_len’ and ‘js_obs_len’ as well as low ‘js_len’ and ‘js_obs_len’.

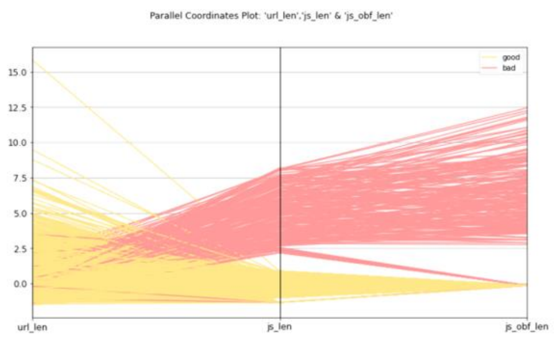


Figure 14: Parallel Coordinates of Numerical Attributes

When we visualized the data in parallel coordinates, we see that malicious and benign web pages are clearly divisible for the ‘js_len’ and ‘js_obs_len’ attribute. However, there is a high amount of overlap for the ‘url_len’ attribute.

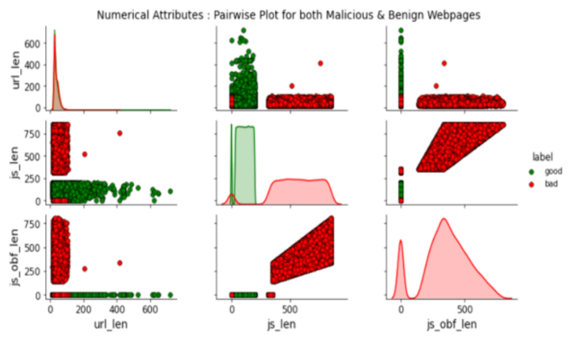


Figure 15: Pairwise Plot of Numerical Attributes

We plot the pairwise plot of malicious and benign web pages above. When looking at the plots with’url_len and ‘js_len’ as well as ‘url_len’ and ‘js_obf_len’, there is a relatively high amount of overlap between malicious and benign web pages. In contrast, for the plot with ‘js_len’ and ‘js_obf_len’, there is a low amount of overlap.

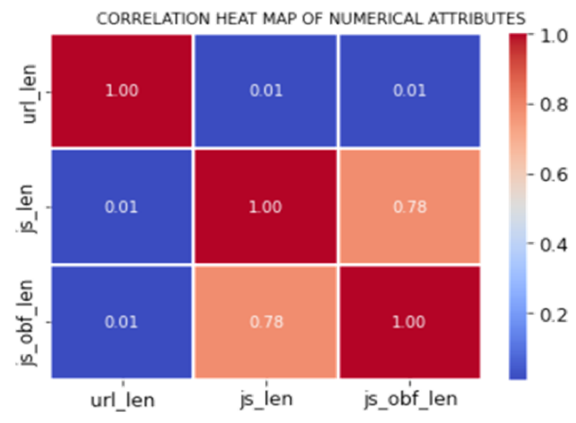


Figure 16: Correlation Heat Map of Numerical Attributes

The correlation heat map is shown above. As seen, 'js_len' and 'js_obf_len' are highly correlated and have a distinct pattern for the two class labels. Hence, these two variables will be analysed further through bivariate analysis to gain further insight.

<u>Bivariate analysis of 'js_len' & 'js_obf_len'</u>

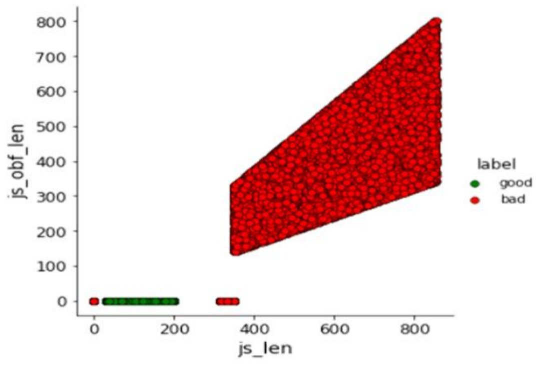


Figure 17: Bivariate Scatter Plot

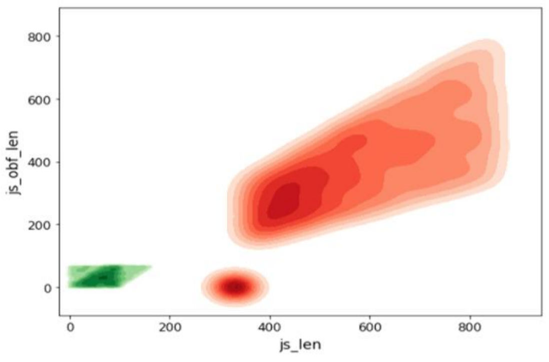


Figure 18: Bivariate Density Plot

Looking at the bivariate scatter and density plots above, we can clearly see that 'js_len' and 'js_obf_len' can segregate the two classes with low overlap.

<u>Analysis of ‘tld’ attribute:</u>

The 'tld' attribute is the 7th column in the dataset. It is a categorical attribute that gives the top level domain name of the web page.

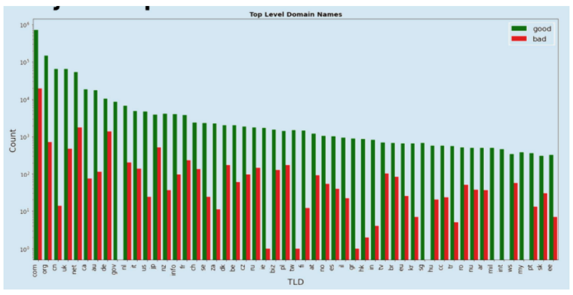


Figure 19: Top Level Domain Names Analysis

Upon completing the analysis and generating the results, the tld attribute did not show any definite patterns. The only clear result is that the tld of “gov”, “sg”, “mil”, “int” and “my” did not have the occurrence of malicious web pages.

<u>Analysis of ‘whois’ attribute:</u>

The 'who_is' attribute is the 8th column of the dataset. It is a categorical attribute with two values, 'complete' or 'incomplete', reflecting whether the registration details are completed or not.

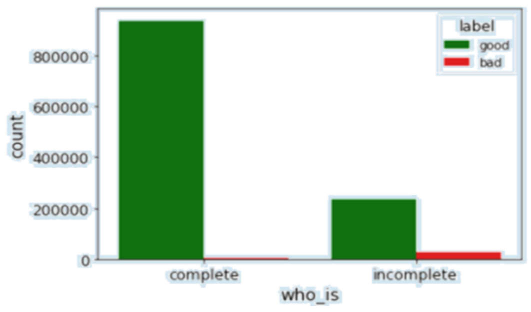


Figure 20: Who-is Analysis

Upon completing the analysis and generating the results, there is a definite pattern where the majority of malicious web pages have incomplete whois profile. Such patterns will aid in the training of models.

<u>Analysis of ‘https’ attribute:</u>

The 'https' attribute is the 9th attribute in the dataset. It is a categorical attribute with two values, 'yes' and 'no', indicating whether the web page is delivered using the secure HTTPS protocol or otherwise.

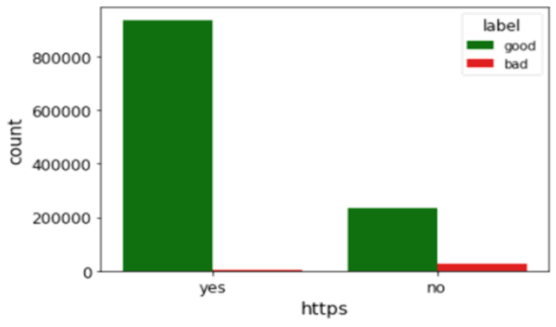


Figure 21: HTTPS Analysis

Upon completing the analysis and generating the results, there is a definite pattern where the majority of benign web pages have used the HTTPS protocol, while most of the malicious web pages do not.

<u>Analysis of ‘content’ attribute:</u>

The 10th column of the dataset is the 'content' attribute. This attribute is the raw web content of the web page, including JavaScript code. However, this raw web content was cleaned and processed to remove punctuations in order to reduce data size. The web content has been stored as a separate attribute in the dataset, so that more attributes could be extracted for future requirements. Also, this raw content may be used in machine learning techniques that can use unstructured data, for example, deep learning.

We carried out visualization on the 'content' attribute using various techniques.

Sentiment Polarity Analysis of Web Content

We did sentiment polarity analysis on the web content deduced from sentences on the web page. The sentiment polarity score ranges from -1 to 1, where -1 means a negative statement and 1 means a positive statement.

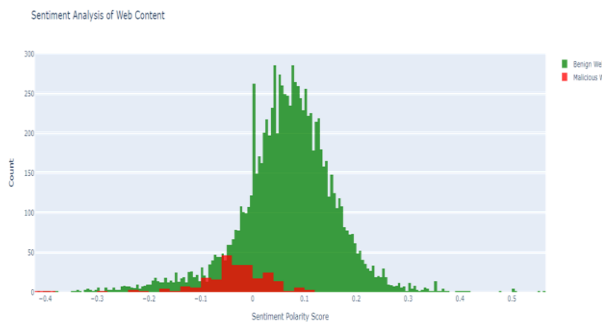


Figure 22: Sentiment Polarity Analysis

The sentiment analysis of the web content is displayed above. As seen, benign web pages have a higher positive sentiment score than malicious web pages.

Profanity Analysis of Web Content

We also did the same profanity analysis that we did on the 'url' attribute on the web content.

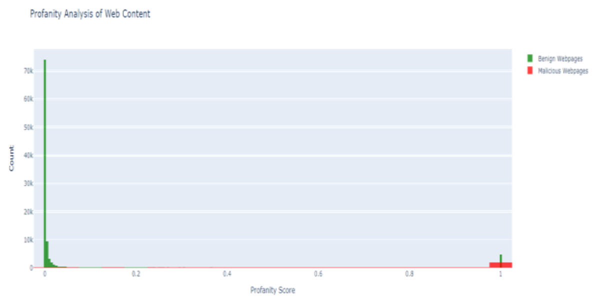


Figure 23: Profanity Analysis

The profanity analysis of the web content is displayed above. This analysis gives a score based on bad/obscene words found on the web page. As seen, malicious web pages have a higher profanity score than benign web pages.

Analysis on length of web content

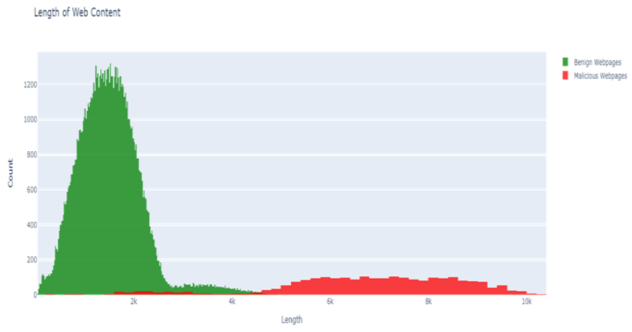


Figure 24: Length of Web Content Analysis

Based on the analysis on the length of web content as seen from above, benign web pages have a shorter length of web content than malicious web pages.

Analysis on word count of web content

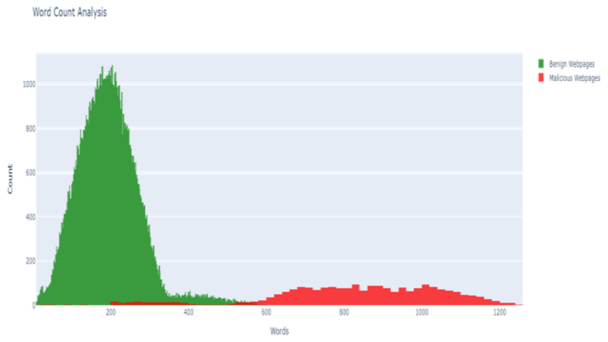


Figure 25: Word Count Analysis

The word count analysis of web content is displayed above. As seen, malicious web pages have higher word counts compared to benign web pages.

Analysis of Complete Dataset: Reducing all Attributes to 3 Dimensions Using PCA

Finally, for the purpose of 3D visualization of the complete dataset, multiple attributes of the dataset are reduced to three principal components using the Principal Component Analysis (PCA).

PCA is a technique used to reduce the number of dimensions (features) in a dataset.

Although feature selection techniques help us select the most important 'k' features corresponding to the target feature, this will not be an overall representation of all the features (instead, it would be just a few features selected from 'n' number of features).

To tackle this problem, we have the PCA Dimensionality reduction technique. The principal components we shall have would be an overall representation of all the features in the dataset. In this way, information loss is minimized.

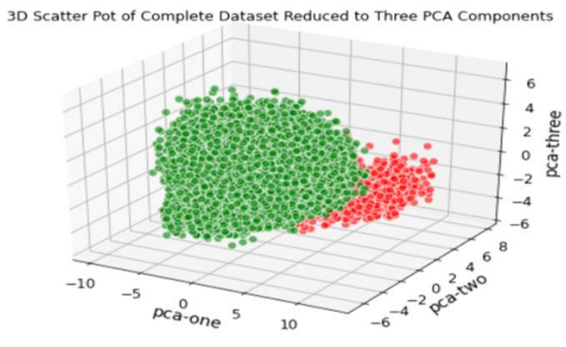


Figure 26: 3D Scatter Plot of 3 PCA Components

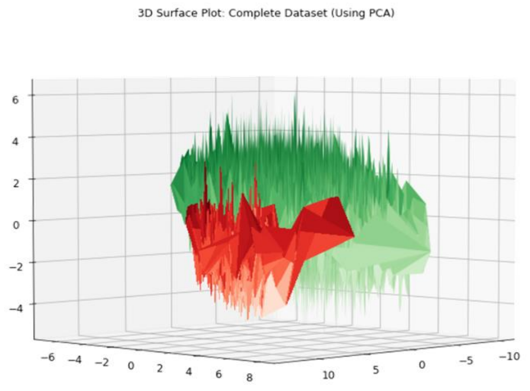


Figure 27: 3D Surface Plot using PCA Components

From the 3D scatter plot and 3D surface plot above, we can see that the dataset can indeed be segregated into the two classes, malicious (bad) web pages and benign (good) web pages.

Feature Engineering

Feature engineering is the process of transforming raw data into features that better represent the underlying problem to the predictive models. This will result in improved model accuracy and generalizations to new, unseen data [11].

In our project, we will use feature engineering to generate more features. This is especially the case for dataset 2, which only has the URL strings and labels.

<u>Feature engineering for dataset 1: *Dataset of Malicious and Benign Webpages*</u>

As this dataset already has other features, we will only be engineering 4 sets of new features.

1. Generating a URL profanity score using the Profanity Check library [12]. To do so, we will first clean the URL by extracting a set of words from the URL string itself.

```
from urllib.parse import urlparse
from tld import get_tld

# Function for cleaning the URL text before calling the
profanity check
def clean_url(url):
    url_text=""
    try:
        domain = get_tld(url, as_object=True)
        domain = get_tld(url, as_object=True)
        url_parsed = urlparse(url)
        url_text= url_parsed.netloc.replace(domain.tld,"
").replace('www',' ') +" "+ url_parsed.path+"
"+url_parsed.params+" "+url_parsed.query+"
"+url_parsed.fragment
        url_text = url_text.translate(str.maketrans({'?':' ','\\':' ','.':'
',';':' ','/':' ','\'':' '}))
        url_text.strip(' ')
        url_text.lower()
    except:
        url_text = url_text.translate(str.maketrans({'?':' ','\\':' ','.':'
',';':' ','/':' ','\'':' '}))
        url_text.strip(' ')
    return url_text
```

For example, the website *http://us.imdb.com/title/tt0176269* will be converted to "*us imdb title tt0176269*". With the array of string tokens, we can pass them to the Profanity Check library, which will generate a probability between 0 and 1. For this particular website example, the generated score is 0.0149, which is rather small.

2. Split the IP address from a single string into the 4 subcomponents.

For example, the IP address 192.168.1.254 will be split to [192, 168.1.254]. The justification for doing so was that previous research had indicated that using IP address splits was a better feature than using one-hot or binary encoding [13].

3. Binary encoding for the columns *who_is* and *https*. A '*yes'* is coded to a 1, while a '*no'* is coded to a 0.

4. Ordinal encoding for the columns *geo_loc* and *tld*. Ordinal encoding is where an ordered number is assigned for each categorical value.

For example, for the category *geo_loc*, the first category (America) would be assigned a value of 1 whereas the second category (Bolivia) would be assigned a value of 0 and so on.

<u>Feature engineering for dataset 2: *Malicious & Benign URLs*</u>

As this dataset only has the URL feature, we would need to generate all of the features manually instead. There are a total of 9 sets of features generated.

1. WHO IS data. Data extracted are binary variables indicating if the WHO IS record exists, the days since record creation, the days since the last update and days until expiration.

2. Lengths of various components like the URL string, path, hostname, first directory and subdomains.

3. Counts of characters and strings such as the characters: - @ ? % . + and strings: *https www.*

4. Counts of parameters and fragments. These are obtained by counting the characters of % and &.

5. Using regex to determine the existence of IP addresses in the URL string

```
# Check if IP address is used in the url
import re
def having_ip_address(url):
    match = re.search(
        '(([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-
5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.'
        '([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\/)|'  # IPv4
        '((0x[0-9a-fA-F]{1,2})\\.(0x[0-9a-fA-F]{1,2})\\.(0x[0-9a-fA-
F]{1,2})\\.(0x[0-9a-fA-F]{1,2})\\/)' # IPv4 in hexadecimal
        '(?:[a-fA-F0-9]{1,4}:){7}[a-fA-F0-9]{1,4}', url)  # Ipv6
    if match:
        return 1
    else:
        return 0
```

6. Using regex to determine if URL shortening services were used.

```
# Check if shortening service is used
def shortening_service(url):
    match =
re.search('bit\.ly|goo\.gl|shorte\.st|go2l\.ink|x\.co|ow\.ly|t\.co|ti
nyurl|tr\.im|is\.gd|cli\.gs|'

'yfrog\.com|migre\.me|ff\.im|tiny\.cc|url4\.eu|twit\.ac|su\.pr|tw
url\.nl|snipurl\.com|'

'short\.to|BudURL\.com|ping\.fm|post\.ly|Just\.as|bkite\.com|s
nipr\.com|fic\.kr|loopt\.us|'

'doiop\.com|short\.ie|kl\.am|wp\.me|rubyurl\.com|om\.ly|to\.ly|
bit\.do|t\.co|lnkd\.in|'

'db\.tt|qr\.ae|adf\.ly|goo\.gl|bitly\.com|cur\.lv|tinyurl\.com|ow\.l
y|bit\.ly|ity\.im|'

'q\.gs|is\.gd|po\.st|bc\.vc|twitthis\.com|u\.to|j\.mp|buzurl\.com|c
utt\.us|u\.bb|yourls\.org|'

'x\.co|prettylinkpro\.com|scrnch\.me|filoops\.info|vzturl\.com|q
r\.net|1url\.com|tweez\.me|v\.gd|'
                'tr\.im|link\.zip\.net',
                url)
    if match:
        return 1
    else:
        return 0
```

7. Calculating the entropy score.

```
def entropy(url):
    string = url.strip()
    prob = [float(string.count(c)) / len(string) for c in
dict.fromkeys(list(string))]
    entropy = sum([(p * math.log(p) / math.log(2.0)) for p in
prob])
    return entropy
```

8. URL profanity score using the Profanity Check library.
9. Top level domain ordinal encodings.

Machine Learning

<u>Methodology</u>

We will train 3 models by splitting the first dataset into 2 parts.

<u>*Splitting dataset 1 into 2 smaller datasets*</u>

Dataset 1a - Contains only the raw web page content strings. This is so that we can focus on only training a Natural Language Processing (NLP) model. This is our model 1.

Dataset 1b - Contains the remaining features. This is our model 2.

<u>*Training dataset 2*</u>

Dataset 2 – Trained dataset 2 with another model. This will be our model 3.

Our training methodology (for each model) is as follows:

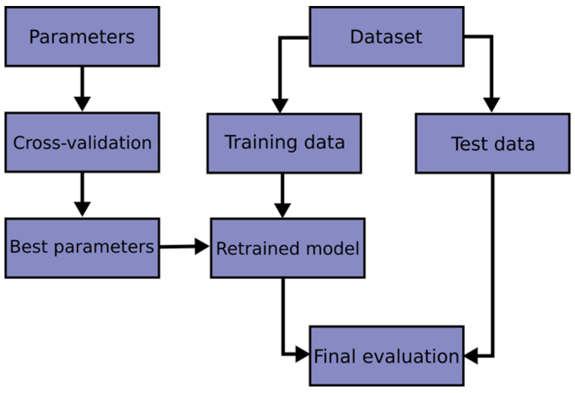


Figure 28: Training Methodology

For each model (and its assigned dataset), we will split the dataset into 2 sub-components; a training set and a test set. The test set is used in the end, after we have obtained an optimal set of model parameters via cross-validation.

For each classification model (e.g. logistic classifier, random forest classifier, decision tree classifier), there are its own set of parameters to tune and optimize. For example, the random forest classifier parameters are the criterion to be used for splitting (either entropy or gini coefficient) and the number of estimators (trees) used. To find out the optimal parameters, we will do 5-fold cross validation.

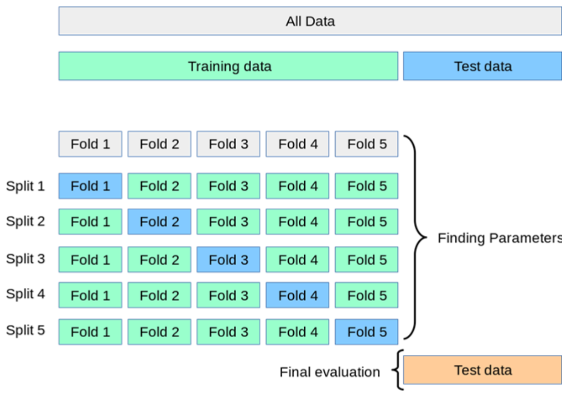


Figure 29:5-Fold Cross Validation

5-fold cross validation is used to compare different model parameters with each other. From 5-fold cross validation, we will get 5 sets of accuracy scores. With the accuracy scores, we can average them out to get the average cross validation accuracy score.

By comparing these different cross validation accuracy scores, we will be able to decide which are the best model parameters to use.

After obtaining the best model parameters, we will retrain with the whole training set and obtain an accuracy score on the test set. We will use the test set accuracy score to obtain the best classification model.

Training on Dataset 1a

Dataset 1a represents the website content strings as well as the labels. As the content is raw strings, we will be using natural language processing to help us.

Our model of choice is the neural network encoder. The encoder will be represented by an embedding.

We will be using transfer learning by using an already trained neural network that has been trained on 130GB of English Google News corpus data [14]. By using the already trained network, we need to only train the output layers of the network. This will help us reduce our training time and data samples needed.

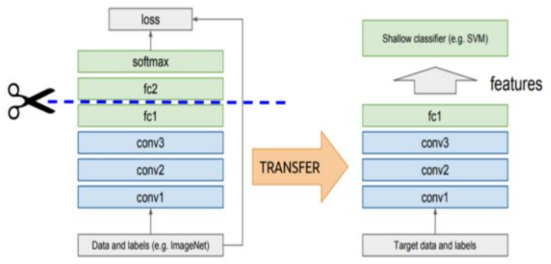


Figure 30: Transfer Learning

The figure above depicts transfer learning. On the left is a pre-trained model that has been trained on similar data to ours, in this case English Google News data.

By keeping the middle layers, we are able to 'transfer' whatever that model has learnt to our use case. To get the model useful for predicting classes, we will combine the pre-trained embedding network as follows.

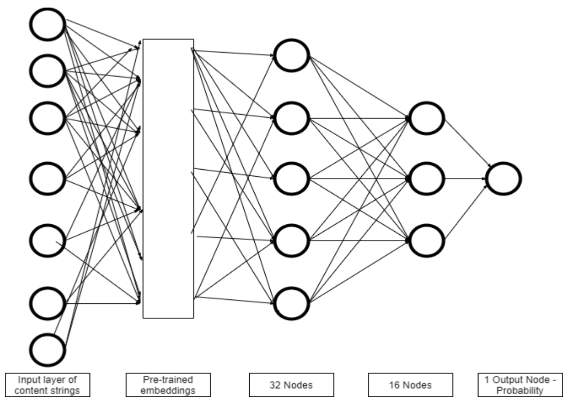


Figure 31: Combination of Pre-Trained Embedding Network

The input will be our website content strings. It will serve as input into the pre-trained model.

After the embedding, we will gradually decrease the number of nodes, from 32 to 16 to 1. As we are doing binary classification, this single node can be used to output probabilities for the binary class.

The code for instantiating our neural network is as follows:

```
encoder = hub.load("https://tfhub.dev/google/tf2-
preview/gnews-swivel-20dim/1")

def create_model(optimizer='adam'):
   model = keras.Sequential([
   hub.KerasLayer(encoder,
input_shape=[],dtype=tf.string,trainable=True),
   keras.layers.Dense(32, activation='relu'),
   keras.layers.Dense(16, activation='relu'),
   keras.layers.Dense(1, activation='sigmoid'),
   ])
   model.compile(loss='binary_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
   return model

# Print model summary
print(create_model().summary())

Model: "sequential"
_________________________________________________________________
_________________
Layer (type)              Output Shape             Param #
=================================================================
====================
keras_layer (KerasLayer)     (None, 20)             400020
_________________________________________________________________
_________________
dense (Dense)             (None, 32)             672
_________________________________________________________________
_________________
dense_1 (Dense)            (None, 16)             528
_________________________________________________________________
_________________
dense_2 (Dense)            (None, 1)              17
=================================================================
====================
Total params: 401,237
Trainable params: 401,237
Non-trainable params: 0
_________________________________________________________________
_________________
None
```

For our parameter tuning, we have decided to optimize on the type of optimizer that the network uses. Optimizers are used to decide how the neural network should adjust its gradients during back-propagation. The optimizers we have chosen are SGD (Stochastic Gradient Descent), RMSprop (Root Mean Squared Propagation), Adagrad (Adaptive Gradient) and Adam (Adaptive Moment Estimation). These optimizers each have their own unique algorithm to help decide how to converge to the global optimal point of minimizing training errors. For a brief comparison of the various optimizers, please see [15].

Our code and results from parameter tuning are as follows.

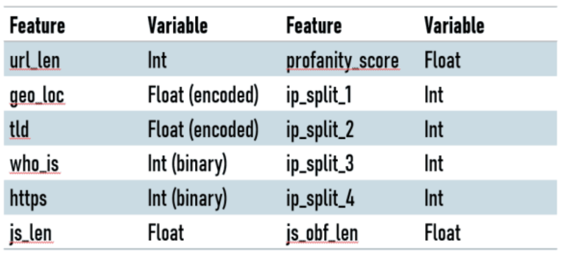
```
# Use grid search to find the best optimizer to use
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adam']
param_grid = dict(optimizer=optimizer)
grid = GridSearchCV(estimator=model,
param_grid=param_grid, n_jobs=1,cv=5)
grid_result = grid.fit(X_train,y_train)

# summarize results for which optimizer to use
print("Best: %f using %s" % (grid_result.best_score_,
grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
   print("%f (%f) with: %r" % (mean, stdev, param))

Best: 0.998339 using {'optimizer': 'RMSprop'}
0.994321 (0.000263) with: {'optimizer': 'SGD'}
0.998339 (0.000164) with: {'optimizer': 'RMSprop'}
0.987725 (0.007276) with: {'optimizer': 'Adagrad'}
0.997896 (0.000230) with: {'optimizer': 'Adam'}
```

From our cross-validation, the best parameters are the RMSProp optimizer, with the highest score of 0.998339.

To see the code used for training for this model, please refer to the file *Dataset_1a_NLP_Model.ipynb.*

Training on Dataset 1b

Dataset 1b represents the remaining features (such as *top level domain*, *js_len*) as well as the engineered *profanity_score* and classification labels.



Feature	Variable	Feature	Variable
url_len	Int	profanity_score	Float
geo_loc	Float (encoded)	ip_split_1	Int
tld	Float (encoded)	ip_split_2	Int
who_is	Int (binary)	ip_split_3	Int
https	Int (binary)	ip_split_4	Int
js_len	Float	js_obf_len	Float

Table 5: Variables of Dataset 1b

Table showing the variables inside dataset 1b.

For this dataset, we will be fitting the following classification models:

1. Random Forest Classifier
2. Decision Tree Classifier
3. Logistic Regression Classifier
4. Gradient Boosting Classifier

The pseudocode for fitting the models is as follows:

```
models = [random_forest_classifier, deecision_tree_classifier,
logistic_regression_classifer, gradient_bootsing_classifier]

for model in models:
    model_grid = GridSearchCV(estimator = model, param_grid
= model_params, cv = 5)
    model_grid.fit(X_train, y_train)
    model_accuracy = model.score(X_test, y_test)
    print(model_grid, model_accuracy)
```

Model Name	Test Accuracy
Random Forest Classifier	0.999
Decision Tree Classifier	0.9984
Logistic Regression Classifier	0.9978
Gradient Boosting Classifier	0.9907

Table 6: Accuracy Scores of the models fitted on dataset 1b

As the random forest classifier has the highest accuracy score, we will be using it for our prediction purposes.

Random Forest Model Details

The random forest parameters are as follows:

```
print(rf_model.get_params())

{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'entropy',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 110,
 'n_jobs': None,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}
```

In this model, 110 separate decision trees are used in the final random forest model.

Trying to visualize one of the trees is near-impossible due to the vast number of features and leaves.

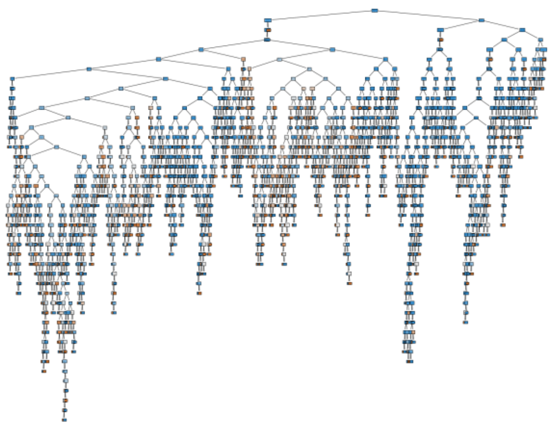


Figure 32: Decision Tree Visualization for one tree in random forest

However, we can still view the feature importance for the whole forest of trees.

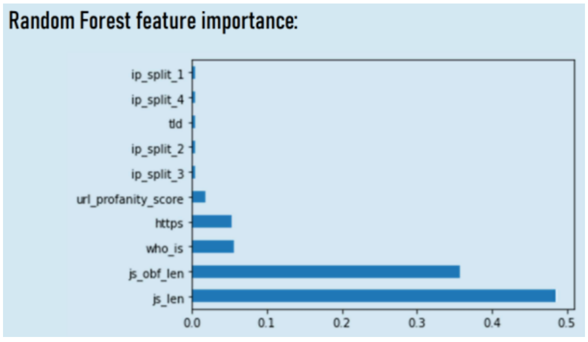


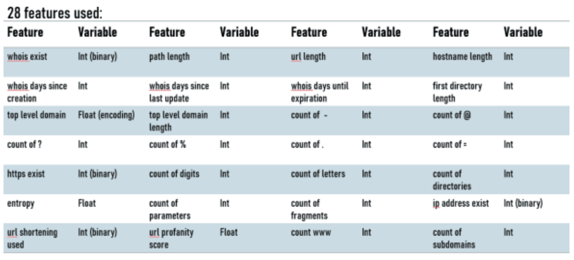
Figure 33: Feature importance for the random forest model

From the above, the most important indicator for a malicious website is the *length of the javascript code* and *obfuscated javascript code*. Intuitively, this makes sense as malicious websites would usually try to embed javascript code on the website itself. Only more legitimate websites would host their javascript code on another file on the web server or have shorter javascript code.

The next most important variable will be the *who is* binary variable followed by the *https binary variable*. Intuitively, this too makes sense as it would be more difficult for malicious websites to obtain *who is records* and obtain *https* status.

To see the code used for training for this model, please refer to the file *Dataset_1b_Random_Forest_Model.ipynb.*

Training on Dataset 2



28 features used:

Feature	Variable	Feature	Variable	Feature	Variable	Feature	Variable
whois exist	Int (binary)	path length	Int	url length	Int	hostname length	Int
whois days since creation	Int	whois days since last update	Int	whois days until expiration	Int	first directory length	Int
top level domain	Float (encoding)	top level domain length	Int	count of -	Int	count of @	Int
count of ?	Int	count of %	Int	count of .	Int	count of =	Int
https exist	Int (binary)	count of digits	Int	count of letters	Int	count of directories	Int
entropy	Float	count of parameters	Int	count of fragments	Int	ip address exist	Int (binary)
url shortening used	Int (binary)	url profanity score	Float	count www	Int	count of subdomains	Int

Table 7: Variables of dataset 2

For dataset 2, all of the features were engineering from the URL string variable itself.

Similar to the previous dataset 1b, we will be training the same set of classification models and select the best model based on the test accuracy

The classification models are:

1. Random Forest Classifier
2. Decision Tree Classifier
3. Logistic Regression Classifier
4. Gradient Boosting Classifier

The pseudocode for fitting the models is as follows:

```
models = [random_forest_classifier, deecision_tree_classifier,
logistic_regression_classifer, gradient_bootsing_classifier]

for model in models:
    model_grid = GridSearchCV(estimator = model, param_grid =
model_params, cv = 5)
    model_grid.fit(X_train, y_train)
    model_accuracy = model.score(X_test, y_test)
    print(model_grid, model_accuracy)
```

Model Name	Test Accuracy
Random Forest Classifier	0.9977
Decision Tree Classifier	0.9963
Logistic Regression Classifier	0.8663
Gradient Boosting Classifier	0.9978

Table 8: Accuracy Scores of the models fitted on dataset 2

As the gradient boosting classifier has the highest accuracy score, we will be using it for our prediction purposes.

Gradient Boosting Model Details

The gradient boosting model parameters are as follows:

```
xgb_model.get_params()


{'objective': 'binary:logistic',
 'use_label_encoder': True,
 'base_score': 0.5,
 'booster': 'gbtree',
 'colsample_bylevel': 1,
 'colsample_bynode': 1,
 'colsample_bytree': 1,
 'gamma': 0,
 'gpu_id': -1,
 'importance_type': 'gain',
 'interaction_constraints': '',
 'learning_rate': 0.300000012,
 'max_delta_step': 0,
 'max_depth': 6,
 'min_child_weight': 1,
 'missing': nan,
 'monotone_constraints': '()',
 'n_estimators': 100,
 'n_jobs': 8,
 'num_parallel_tree': 1,
 'random_state': 0,
 'reg_alpha': 0,
 'reg_lambda': 1,
 'scale_pos_weight': 1,
 'subsample': 1,
 'tree_method': 'exact',
 'validate_parameters': 1,
 'verbosity': None}
```

The gradient boosting classifier is similar to a random forest classifier as both use a number of decision trees.

In this case, 100 estimator decision trees are used.

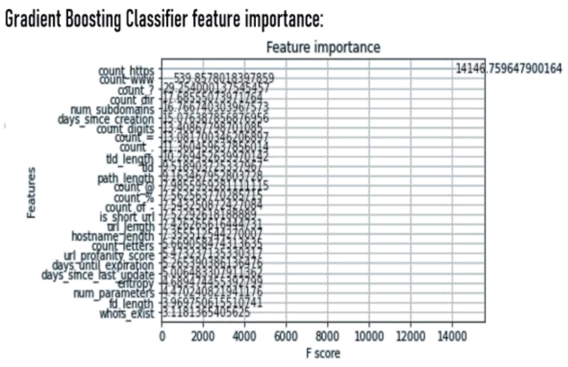


Figure 34: Feature importance for the gradient boosting model

From the above feature importance, the most important feature is the *count_https* feature. This coincides with the observation from model 2, where *https_binary* is also one of the top few features.

The next most important feature is the *count_www* feature. This means that for this dataset, the malicious URLs tend to be those that do not contain 'www' in their URLs.

To see the code used for training for this model, please refer to the file *Dataset_2_Gradient_Boost_Model.ipynb*.

Model Deployment

3 models were trained using the 2 datasets, namely the NLP Neural Network, Random Forest Classifier, and the Gradient Boosting Classifier. The table below summarises the important factors of each model.

Model	**Most Important Variable**	**Implication**
NLP Neural Network	Content string of website	Website must be active
Random Forest Classifier	JS code variables	Website must be active
Gradient Boosting Classifier	‘https’ string	Website need not be active

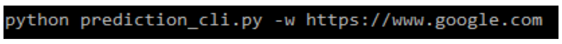
Table 9: Most important variable for each classification model

The models can now be used to test other websites. In order to consolidate the results and to simply determine if a website is benign or malicious, a ‘majority vote’ model is used:

1. If a website is active, we can get predictions using all 3 models and get a majority vote on the final outcome.
2. If a website is inactive, we will be unable to generate features for the NLP Neural Network and Random Forest classifier models. Hence, only the Gradient Boosting Classifier model is used to determine the final outcome as it has more offline features.

<u>Python Script</u>

The initial deployment of the models will be done through a Python script. The main functions of the script correspond to the 3 models above, namely the ‘*nlp_model_prediction()*’ function, ‘*rf_model_prediction()*’ function, and the ‘*xgb_model_prediction()*’ function. A URL (for example, https://www.google.com) is parsed into the script using the following command:

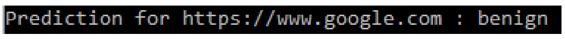


```
python prediction_cli.py -w https://www.google.com
```

The input URL is then cleaned, extracting the useful information required by the models to perform the prediction. The main functions are called and the ‘majority vote’ occurs according to the code below.

```
def overall_prediction(url):


    # If URL does not have a http, add the http and check if a request is possible
    # If so, keep the 'http' inside
    temp_url = ''
    if 'http' not in url:
        temp_url = r'http://' + url
        is_http_needed = False
        try:
            temp_1 = urlparse(temp_url)
            temp = socket.gethostbyname(temp_1.netloc)
            is_http_needed = True
        except Exception as e:
            is_http_needed = False

        if is_http_needed == True:
            url = temp_url

    # Check if the URl is active by checking if host name exists
    parsed_url = urlparse(url)
    is_active = False
    ip_addr = None
    try:
        ip_addr = socket.gethostbyname(parsed_url.netloc)
        is_active = True
    except Exception as e:
        is_active = False

    # Check if a GET request is possible
    get_address = parsed_url.scheme + r'://' + parsed_url.netloc
    try:
        r = requests.get(get_address)
        if 'http' in r.url:
            is_active = True
    except Exception as e:
        is_active = False


    # If not active, only use model 3
    if is_active == False:
        prediction = xgb_model_prediction(url = url,
                     xgb_model = xgb_model,
                     tld_encoder = xgb_encode_tld)
        return prediction
    else:
        # Use all 3 models and get a majority vote
        prediction_1 = nlp_model_prediction(url)
        prediction_2 = rf_model_prediction(url = url,
                    rf_model = random_forest_model,
                    tld_encoder = random_forest_encode_tld,
                    geo_encoder = random_forest_encode_geo_loc)
        prediction_3 = xgb_model_prediction(url = url,
                     xgb_model = xgb_model,
                     tld_encoder = xgb_encode_tld)

        # Get majority prediction
        preds = Counter([prediction_1, prediction_2, prediction_3])
        # print(preds)
        majority, count = preds.most_common()[0]
        return majority
```

The output is then displayed as follows



```
Prediction for https://www.google.com : benign
```

To use the CLI app, please see the *CLI App* folder and the associated readme file.

Flask API

We then adapted the Python script into an Application Programming Interface (API) that is hosted on a simple Flask server, so that web applications such as our Chrome Extension will be able to call it. Flask is a Python web framework, which is a set of resources and tools for developers to build and to manage web apps. It is easy to implement and has high scalability, allowing us to add new features to the API in the future if needed. It supports standard HTML methods such as GET and POST requests, which will be used to call the API [19].

Adding the code below allows us to adapt the Python script into an API using Flask.

```
app = flask.Flask(__name__)
app.config["DEBUG"] = True

@app.route('/api', methods=['GET'])
def overall():
    if 'url' in request.args:
        url = str(request.args['url'])
    else:
        return "Error: No url field provided"

    result = overall_prediction(url)

    return jsonify(result)
```

As a proof of concept, the Flask server will be run on a local machine. The functionality of the server is shown in the figure below.

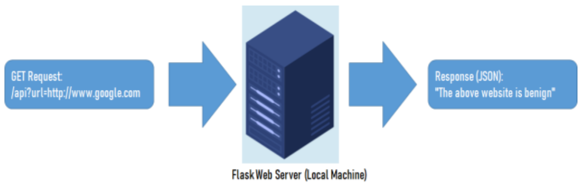

GET Request:
/api?url=http://www.google.com
Response (JSON):
"The above website is benign"
Flask Web Server (Local Machine)

Figure 35: Functionality of the server

A html GET request is submitted to the server, which then returns a JSON response, such as "*The above website is benign*" or "*The above website is malicious*".

Chrome Extension

We developed a browser extension that is capable of calling our Flask API and displaying the result of whether a page is deemed benign or malicious by our machine learning models. Google Chrome was chosen as the platform to write our extension on as it currently has the largest market share of close to 70% [20]. It has a robust framework for extension implementation and supports existing web technologies such as HTML, CSS and Javascript. It is also easy for developers to upload code for testing by using developer mode, enabling them to upload their extensions at the push of a button [21].

The main goals of our Chrome extension are as follows:

1. Able to successfully call our Flask API.
2. Easy to use, able to activate using simple user input such as clicking, without the need for any commands.
3. Simple UI that displays whether the current page is benign or malicious.

To use the Chrome Extension, please see the *Chrome Extension* folder and the associated readme file.

Creating the Extension

A Chrome Extension consists of mainly 3 parts:

1. The manifest
2. The front end code for the UI (in HTML)
3. The back end code that powers the extension (in Javascript).

The manifest (*manifest.json*) consists of the metadata of the extension itself, the name, description, version, permissions etc.

The back end code (*popup.js*) as follows:

```
// pressing the button calls the getCurrentTabUrl() function.
let GetURL = document.getElementById('GetURL');
GetURL.onclick = function(element) {
  getCurrentTabUrl();
};

// gets the URL of the current tab and sends it as a GET request
to the Flask API
function getCurrentTabUrl() {
  var queryInfo = {
    active: true,
    currentWindow: true
  };
  chrome.tabs.query(queryInfo, (tabs) => {
    var tab = tabs[0];
    var url = tab.url;
    var mal;
    document.getElementById('url').innerHTML = url;
    fetch('http://127.0.0.1:5000/api?url=' +
url).then(function(response){return response.json();}).then(data
=>document.getElementById('mal').innerHTML = data).then(()
=> console.log(mal))
  });
}
```

When the button in the UI is pressed, the *getCurrentTabUrl()* function is called, which gets the URL of the current tab and sends it as a GET request to the Flask API.

The API then provides a response, which will then be displayed in the UI. When paired with the front end web page code (*popup.html*), the extension itself works according to the schematic below.

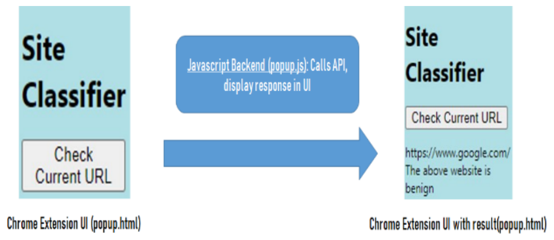

Site Classifier
Check Current URL
Chrome Extension UI (popup.html)
Javascript Backend (popup.js): Calls API, display response in UI
Site Classifier
Check Current URL
https://www.google.com/
The above website is benign
Chrome Extension UI with result(popup.html)

Figure 36: Chrome Extension

Applying to Real-world URLs

The *Malicious URLs Dataset* is the third dataset of URLs that we obtained. The main motivation was that since this dataset was released very recently around 2 months ago (24 July 2021), we could use this dataset to test our overall prediction model.

Comparing this dataset with our current training datasets, there are 650,112 new URLs that our model has not seen before.

Due to processing limitations, we could only carry out predictions for 4,497 rows.

The results from our test dataset is as follows:

	Predicted Benign	**Predicted Malicious**
Actual Benign	2675	707
Actual Malicious	577	538

Table 10: Confusion matrix from test dataset

In our calculations, we define maliciousness as positive.

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

The accuracy on the test is 71.45%, precision is 43.21% and recall is 48.25%.

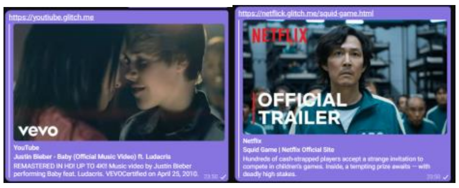
Our final model accuracy performance is moderate. It is better than a random guess (50%) but more tweaking can be done to ensure that it is sufficiently good enough (around 80% accuracy is desired).

An elaboration of the model's limitations, along with possible improvements, will be given in the next section.

On a side note, we tested our classification model with two other fellow Systems Security groups' malicious URLs and the results are as follows:

URL	**Description**	**Actual label**	**Predicted class**
https://autofill-one.herokuapp.com/	Website with normal form	benign	benign
https://autofill-one.herokuapp.com/invisible	Website with hidden form designed to extract auto login credentials	malicious	malicious
https://youtiube.glitch.me	Website redirecting to Rick Roll video. Uses telegram preview exploit	malicious	malicious
https://netflick.glitch.me/squid-game.html	Redirected website. Uses telegram preview exploit	malicious	malicious
https://uob.glitch.me	Redirected website. Uses telegram preview exploit	malicious	malicious

Table 11: Chrome Extension Results of other Malicious URLs

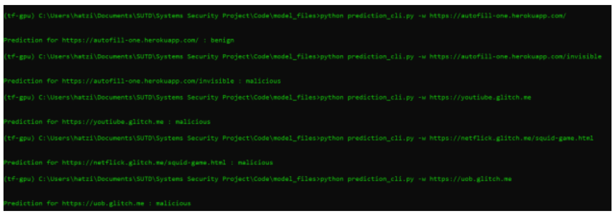


https://youtiube.glitch.me
YouTube
Justin Bieber - Baby (Official Music Video) ft. Ludacris
REMASTERED IN HD! UP TO 4K! Music video by Justin Bieber performing Baby feat. Ludacris. VEVOCertified on April 25, 2010.

https://netflick.glitch.me/squid-game.html
Netflix
Squid Game | Netflix Official Site
Hundreds of cash-strapped players accept a strange invitation to compete in children's games. Inside, a tempting prize awaits — with deadly high stakes.

Figure 37: Malicious Websites via Telegram's Preview Feature

The above figure shows the malicious websites created by another Systems Security group exploiting Telegram's preview feature.



```
(tf-gpu) C:\Users\hatzi\Documents\SUTD\Systems Security Project\Code\model_files>python prediction_cli.py -w https://autofill-one.herokuapp.com/

Prediction for https://autofill-one.herokuapp.com/ : benign

(tf-gpu) C:\Users\hatzi\Documents\SUTD\Systems Security Project\Code\model_files>python prediction_cli.py -w https://autofill-one.herokuapp.com/invisible

Prediction for https://autofill-one.herokuapp.com/invisible : malicious

(tf-gpu) C:\Users\hatzi\Documents\SUTD\Systems Security Project\Code\model_files>python prediction_cli.py -w https://youtiube.glitch.me

Prediction for https://youtiube.glitch.me : malicious

(tf-gpu) C:\Users\hatzi\Documents\SUTD\Systems Security Project\Code\model_files>python prediction_cli.py -w https://netflick.glitch.me/squid-game.html

Prediction for https://netflick.glitch.me/squid-game.html : malicious

(tf-gpu) C:\Users\hatzi\Documents\SUTD\Systems Security Project\Code\model_files>python prediction_cli.py -w https://uob.glitch.me

Prediction for https://uob.glitch.me : malicious
```

Figure 38: Predictions of the URLs using our CLI app

Hence, despite our classification model having only 71.45% accuracy, it does appear to be beneficial in some use cases.

Machine Learning Model Limitations

Classification Metrics Used

An essential part of machine learning is to evaluate the model to assess how true the returned results are (i.e. the number of correct predictions). More often than not, classification accuracy is used to measure the performance of the model. Classification accuracy is the ratio of number of correct predictions to the total number of input samples [22].

$$Accuracy = \frac{Number\ of\ Correct\ predictions}{Total\ number\ of\ predictions\ made}$$

Likewise, we also used classification accuracy to measure the performance of our various models. However, it should be noted that classification accuracy is not enough to truly judge our model. Other forms of evaluation metrics can also be used to aid in the evaluation of the model performance.

Classification accuracy alone may result in a false sense of achieving high accuracy [23]. Hence, an improvement would be to incorporate other performance metrics such as Logarithmic Loss, Confusion Matrix, Area under Curve, F1 Score, Mean Absolute Error and Mean Squared Error.

Using Majority Vote

For our final prediction model, we had used a majority vote based on the individual outcomes of each model. However, giving equal weight to all models may not have been optimal.

A better approach would be to train the whole ensemble (combination) of models together instead of training each model individually. By doing so, we can tweak the weights for each model.

Currently, the prediction model (that uses majority vote) can be interpreted as giving equal weights to all 3 models.

Mathematically,

$$\text{Weighted score} = \frac{\theta_1 Model_1 + \theta_2 Model_2 + \theta_3 Model_3}{\theta_1 + \theta_2 + \theta_3} \text{ where } Model_i \subseteq [0,1]$$

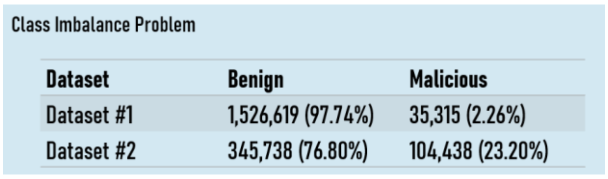
$$\text{Final prediction} = \begin{cases} 1 & \text{if weighted score} \geqslant 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Each $Model_i$ will return a binary prediction i.e. [0,1]. For our current majority model, $\theta_1 = \theta_2 = \theta_3$

However, by assigning different weights, perhaps Model 1 and Model 2 would have a larger weight than Model 3 as Models 1 & 2 were trained on a much larger dataset than Model 3.

By optimising these weights, this should help improve overall accuracy.

Class Imbalance Problem



Class Imbalance Problem

Dataset	Benign	Malicious
Dataset #1	1,526,619 (97.74%)	35,315 (2.26%)
Dataset #2	345,738 (76.80%)	104,438 (23.20%)

Table 12: Class Distribution of each dataset

We can observe that there is a large class imbalance problem for the first dataset. Only 2.26% is malicious compared to 23.2% for the second dataset.

One possible way to address this is to implement Synthetic Minority Oversampling Technique (SMOTE) [16]. In SMOTE, new minority samples are generated by looking at the neighbours of minority sample data points.

By generating more samples of the malicious data points, the classification models will be able to discern and learn better. This would help to increase the overall accuracy for Model 3 (which was trained on the smaller dataset).

Chrome Extension Limitations

The current chrome extension only displays whether a web page is deemed "*benign*" or "*malicious*" based on the response from the API, which derives the response from the machine learning models. Casual users might be satisfied with this level of functionality. However, some users might want to know the reasons behind the response, such as the various metrics that were used as inputs to the machine learning models (*profanity score, presence of whois data, javascript code length* etc). Therefore, future work on the chrome extension could involve adding this particular functionality for end users to make a more informed decision on whether they should continue browsing the particular website or not.

Additional functionality such as allowing the user to interact with the API directly through the use of a text box by entering the URL into it could be added as well, so that the user can get information about the website without accessing the website directly. A mock-up of the improved chrome extension is shown in the image below.

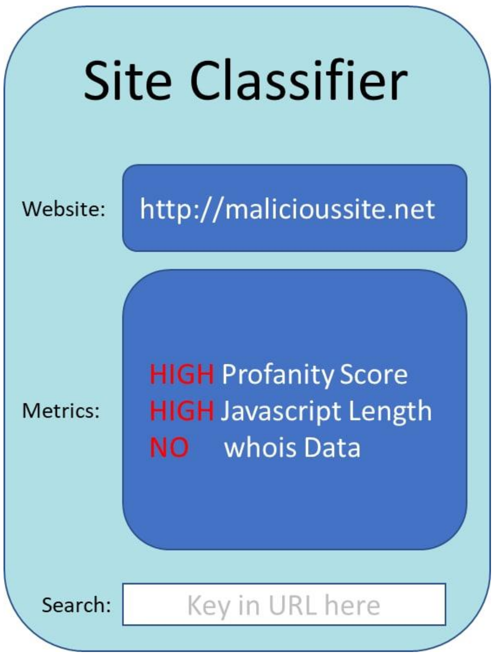


Figure 39: Mock-up of Improved Chrome Extension

Furthermore, the random forest model has a built-in 'decision_path' function which can be used to explain the decisions for the classification decision [24].

The gradient boosting classifier does not have a built-in function however [25] has implemented a way to interpret the explain-ability of features of a gradient boosting model when used for prediction.

We can use these functions to better explain the decision-making process to the user.

Conclusion

In this project, we analyzed 2 malicious URL datasets (*Dataset of Malicious and Benign Webpages*, *Malicious & Benign URLs*) to better understand them via data visualizations. After performing multiple visualisations, we were able to shortlist these attributes (*profanity score, js length, whois status, http status* and *web content strings*) which may have a greater factor in determining the website classification.

Through data visualization, we were able better understand which of the ten features are more relevant for our models to train on. The 'url', 'js_len', 'js_obf_len', 'who_is', 'https' and 'content' attributes seem to be able to clearly differentiate between malicious and benign web pages with low overlap.

After understanding the 2 datasets, we engineered more features that will help augment the machine learning training process. Some of these features were the usage of IP addresses in the URL, usage of shortening services and WHO IS domain information.

Subsequently, we proceeded to the machine learning process itself. We trained 3 different models: the NLP neural network, random forest classifier and gradient boosting classifier.

After we have our trained model, we deployed them into a back-end flask API with a front-end Chrome extension. This will help users easily utilize our model predictions when they are actively browsing.

Next, we used a very recently released third dataset (*Malicious URLs Dataset*) to test how accurate our model was. Our model achieved a 71.45% accuracy, 43.21% recall and 48.25% precision.

Lastly, we addressed our model limitations and gave some possible strategies that would help increase our model performance.

References

1. A. T. Tunggal, “Why is Cybersecurity Important?”. Retrieved from https://www.upguard.com/blog/cybersecurity-important#:~:text=Cybersecurity%20is%20important%20because%20it,governmental%20and%20industry%20information%20systems., 2021

2. N. Lord (2018), “What is a Phishing Attack? Defining and Identifying Different Types of Phishing Attacks”. Retrieved from https://digitalguardian.com/blog/whatphishing-attack-defining-and-identifying-different-types-phishingattacks, 2018

3. Hyunsang Choi et al, “Detecting Malicious Web Links and Identifying Their Attack Types”. Retrieved from https://www.usenix.org/legacy/events/webapps11/tech/final_files/Choi.pdf, n.d.

4. Mozilla Developer Network Web Docs (2021), “What is a URL?”. Retrieved from https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_URL, 2021

5. A.K. Singh. (2020 September 12). “Malicious and Benign Webpages Dataset”. doi: 10.1016/j.dib.2020.106304. PMCID: PMC7648114. PMID: 33204771. Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7648114/

6. A.K. Singh. (2020 April 4). “Dataset of Malicious and Benign Webpages”. Retrieved from https://www.kaggle.com/aksingh2411/dataset-of-malicious-and-benign-webpages

7. A.K. Singh. (2020 May 2). “Dataset of Malicious and Benign Webpages”. DOI:10.17632/gdx3pkwp47.2. Mendeley Data. Retrieved from https://data.mendeley.com/datasets/gdx3pkwp47/2

8. Siddharth Kumar. (2019 May 31). “Malicious And Benign URLs. Kaggle. Retrieved from https://www.kaggle.com/siddharthkumar25/malicious-and-benign-urls

9. URL dataset (ISCX-URL 2016). University of New Brunswick. Canadian Institute for Cybersecurity. Retrieved from https://www.unb.ca/cic/datasets/url-2016.html

10. Manu Siddhartha. (2021 July 24). “Malicious URLs dataset”. Kaggle. Retrieved from https://www.kaggle.com/sid321axn/malicious-urls-dataset/metadata

11. Jason Brownlee. (2014 September 26). “Discover Feature Engineering, How To Engineer Features and How to Get Good at It”. Retrieved from https://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/

12. Victor Zhou. (2019 August 23). “Profanity check library”. Retrieved from https://pypi.org/project/profanity-check/

13. Enchun Shao. (2019 December 3). “Encoding IP Address as a Feature for Network Intrusion Detection”. Purdue University. Retrieved from https://hammer.purdue.edu/articles/thesis/Encoding_IP_Address_as_a_Feature_for_Network_Intrusion_Detection/11307287

14. Google. (2021 July 10). “tf2-preview/gnews-swivel-20dim Text embedding”. TensorFlow Hub, Retrieved from https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1

15. Sanket Doshi. (2019 Jan 13). “Various Optimization Algorithms For Training Neural Network”. Retrieved from https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6

16. Nitesh V. Chawla et. al. (06/02). “SMOTE: Synthetic Minority Over-sampling Technique”. Journal of Artificial Intelligence Research 16 (2002) 321-357. Retrieved from https://scholar.google.com.sg/scholar_url?url=https://www.jair.org/index.php/jair/article/download/10302/24590&hl=en&sa=X&ei=M59eYYqQI6XGywS7_I6ABQ&scisig=AAGBfm0zNdcfXdPynWxoQ3FsFum2KdF9ow&oi=scholarr

17. Patricia Stainer. (2021 May 4). “Alarming Cybersecurity Statistics for 2021 and the Future”. Retrieved from https://www.retarus.com/blog/en/alarming-cybersecurity-statistics-for-2021-and-the-future/

18. Radoslav Ch.(2021). “How many websites are there? How many are active in 2021?”. Retrieved from https://hostingtribunal.com/blog/how-many-websites/

19. Patrick Smyth (2018 04 02). “Creating Web APIs with Python and Flask”. Retrieved from https://programminghistorian.org/en/lessons/creating-apis-with-python-and-flask

20. StatCounter (2021 September). “Browser Market Share Worldwide - September 2021”. Retrieved from https://gs.statcounter.com/browser-market-share

21. Chrome Developers. (2021 July 22). “Chrome Extension Documentation”. Retrieved from https://developer.chrome.com/docs/extensions/mv3/getstarted/

22. Jason Brownlee. (2014 March 21). “Classification Accuracy Is Not Enough: More Performance Measures You Can Use”. Retrieved from https://machinelearningmastery.com/classification-accuracy-is-not-enough-more-performance-measures-you-can-use/

23. Aditya Mishra. (2018 Feb 24). “Metrics to Evaluate your Machine Learning Algorithm”. Retrieved from https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234

24. Scikit Learn. (2021). “Random Forest Classifier, decision_path”. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.decision_path

25. Delgado Panadero. (2021 March 10). “Implementing Explainability for Gradient Boosting Trees”. Retrieved from https://towardsdatascience.com/implementing-explainability-for-gradient-boosting-trees-9dde33ecdabd