2020 5th International Conference on Information Science, Computer Technology and Transportation (ISCTT) | 978-1-7281-8575-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/ISCTT51595.2020.00060

Automatic Detection of Malicious URLs using Fine-Tuned Classification Model

Chiyu Ding
Jiangsu Tianyi High School
Jiangsu, China

***Abstract*—URLs are frequently used to surf the Internet in modern society. Especially after the outbreak of the COVID-19, quarantine makes needs for the Internet and usages of URLs reach an unprecedented level. Unfortunately, not every URL is believable, because some of them can attack your computer, steal your personal information, and even spread computer virus like Trojan. This project designs machine learning algorithms to detect malicious URLs efficiently and protect Internet users from malicious URLs.**

In this project, the goal is to get a fine-tuned machine learning model, I will first introduce the dataset of URLs which is crucial to train the model. Then I will show the procedure and some findings of my data exploration. After that, I will present the methods including the algorithms and some improvements I make to optimize my model. Finally, I will show the results and the conclusion. To make my models better, I not only apply hyperparameter tuning, but also data resampling and cross validation to the model. The procedure is repeated several times to ensure the stability. In order to evaluate the performance of my models accurately, I adopt multiple methods. After improving the algorithms, by using the F1 score to evaluate performance, the result boosts significantly from original 0.14 to around 0.90. With the ultimate well-trained model, we can predict the safety of all the URLs on the Internet accurately, which can secure our personal information and data.

Keywords-URL; Internet; machine learning

I. INTRODUCTION

Nowadays, we all have access to the Internet and all the resources on it. URLs, the uniform resource locators, are applied to get files downloaded or into another website while surfing on the Internet. As the increasing development of information technology and the emergence of contents in the World Wide Web, billions of URLs are created. However, not every URL is completely safe. We must keep alert while clicking on those URLs, because you might enter some malicious URLs by accident. Especially in recent times, with the breakout of COVID-19, more and more people are forced to stay home. Most of them get access to the Internet to know what happens outside by their electronic devices. According to forbes.com, after the outbreak of COVID-19, the total Internet hits surged by between 50% and 70%. [1] Those malicious URLs may steal our personal information, make us download unwanted software without permit, or even invade our computer. They are so harmful that we must take action to prevent these URLs from harming our cyber security.

A feasible and easy-to-use method to solve this problem is to make a blacklist of malicious URLs. When users meet a malicious URL, they can mark it or put it into a database of malicious URLs to remind themselves that these are malicious URLs. When somebody goes to surf a URL, he or she just needs to check the database to check whether this URL is marked malicious or not to easily judge good and bad URLs. This database is called blacklist. The database of malicious URLs will be updated over time. Of course, people can also communicate with each other about whether a URL is malicious or not according to their experience to enlarge their blacklist. It is quite an easy method. However, its disadvantage is obvious too. That is, the capacitance is quite limited while the production of malicious URLs can be very fast. To eliminate people's loss is not accessible through this simple method. Moreover, hackers can even manage to change the blacklist of users, which makes the list useless. Moreover, hackers can manage to hide malicious URL after a short URL to lure people to enter. Also, only someone suffers from the malicious URLs, even loses some important things, can the users know that the URL is not safe. It cannot predict whether a URL is malicious or benign. [2]

One improved method based on the blacklist is a heuristic approach. [2,3] The main idea is to change the blacklist of URLs to the blacklist of signatures. When a common attack appears, the system marks a signature as the symbol of such an attack. After marking enough types of attack, when users enter a URL, the system will scan the signature of the URL and check whether it is in the blacklist or not check the URL is safe or not. Its advantage is that since it will scan the signature of the URLs, even a new URL was made to attack users, its signature can be identified as malicious and users will know that the URL is not safe. However, it is only useful to block the attack of common threats. Other complex attacks or newly-built threats, may not be identified as malicious by the blacklist.

Since what the blacklist method can do to protect the safety of Internet users is quite limited, there is a need to develop other powerful methods to predict whether a URL is malicious or benign actively. In this case, I find machine learning a highly effective method. Machine learning uses a database of URLs as train data to train a model to predict the safety of the URLs. [4] While training, machine learning uses a specific algorithm to find the relationship between the features of the URLs and the status of the URLs and get a model. Then using this model, we can predict whether a URL is malicious or benign. To test the performance of the model, we can use another database of URLs as test data, get the results of the model predicting the test data, and evaluating the results.

978-1-7281-8575-0/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCTT51595.2020.00060

To train the data, the selection of features is quite crucial. If just simply using the original URLs, the list of strings of the URLs is just like the blacklist instead of the prediction. As a result, we must find some important information as typical features in the URLs. These features also need to follow certain principles. After gathering the features, we must use certain data structures to store them to make training easier. Using this method, we can also pick some other URLs and select the same features as test data.

After getting the train data and test data, we need to choose an appropriate algorithm to train the predicting model. Up to now, people have designed many machine learning algorithms, including classic methods like Linear Regression, Logistic Regression, Naïve Bayes, and Decision Tree, and advanced methods such as deep learning.

In this report, I will focus on two algorithms: Naïve Bayes and Logistic Regression.

Naïve Bayes is a classifying algorithm using Bayes' theorem. [5] It calculates the conditional possibilities of each feature in every label to predict the final result. For every single kind of feature, Naïve Bayes assumes that they are all independent. For example, if Naïve Bayes is used to predict whether one kind of stationery is a pencil or not, it will not consider the relationship between any of the two features of that kind of stationery, like its shape and its usage.

Logistic Regression is also a classifying method. [6] It uses a logistic function to get a binary prediction. Since the result of the logistic function limits to 0 to 1, we can just assume that it is a binary answer: smaller than 0.5 for 0 and larger than 0.5 for 1. In this model, it uses the features to train a suitable set of parameters of the logistic function to fit the data for prediction.

In my project, I try to get useful and appropriate features through data exploration and statistical analysis. The meanings and the use of the features are quite feasible to explain. In the actual training procedure, I combine the original token counts in the URLs with manually crafted features like the numbers of levels, which can make the trained model have better performance. To evaluate the performance of the model, I manage to avoid the limits of the traditional evaluation by repeating the procedure of training and test using different separations of datasets. Of course, this needs data resampling and cross validation to select different training data and test data to get balanced data with originally imbalanced data. Also, while evaluating the models, I not only use the primitive method----simply calculating accuracy, but also utilize some other evaluating methods, F1 score and AUC score to give a comprehensive performance evaluation of my models.

After many improvements, using Naïve Bayes and Logistic Regression, I get two comparatively accurate prediction models to help users predict the status of URLs. Generally, with more features considered in the model, the better the performance of the model. Comparing the performance of the Naïve Bayes model and the Logistic Regression model, the latter is better. While using the Logistic Regression model with the optimal combination of features and hyperparameter tuning, the F1-score increases from the lowest 0.14 to the highest 0.91, and the AUC score also inclines from 0.53 to nearly 0.95, which is quite a lot performance progress.

II. Problem Definition

The problem we want to solve can be described shortly as following: given a large number of URLs, predict whether they are malicious or benign. This is a binary problem. For every URL, the answer choices are only "malicious" and "benign".

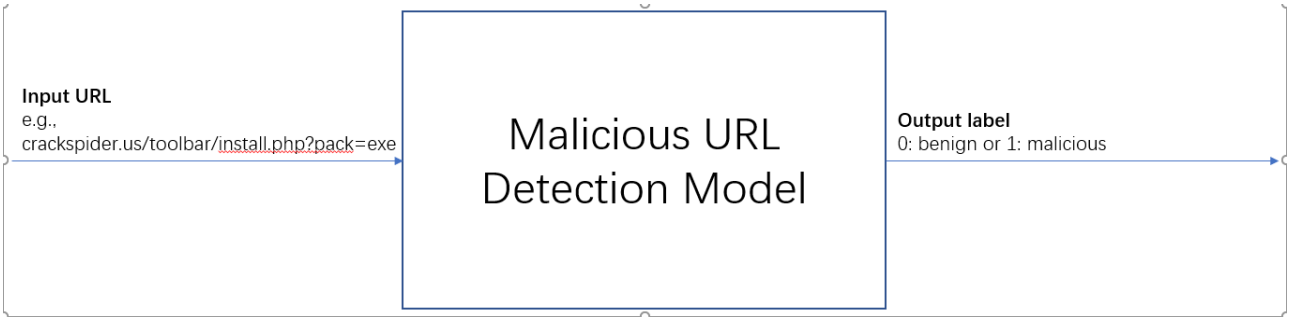

Figure 1. Machine learning model interface

In the original data, each data sample is composed of two parts: the text of the URL and its label. In the dataset, we can just use 1 for malicious URLs and 0 for benign URLs.

As illustrated in figure 1, we want to design a machine learning algorithm that can take the URL body as input and output a label indicating if the URL is malicious. The machine learning algorithm can use all the available features derived from the URL body based on the standard URL structure. The performance of the machine learning algorithm will be evaluation on two main evaluation metrics. The first is F1-score, which is the harmonic mean between precision and recall. The second is area under the curve(AUC), which represents the area under the Receiver Operating Characteristic curve. These two evaluation metrics can evaluate the performance of binary prediction models in an objective way even if the data classes are imbalanced.

In this project, we assume that our training data represent the latest presents of the malicious URLs. However, the malicious URLs are typically generated by hackers using certain algorithms. They might dynamically update their algorithms to combat with the detection system and come up with newly generated malicious URLs that are hard to be detected. Thus, we need to also dynamically update our detection model in practice. This could be achieved by

retraining our model with the latest malicious URLs. We do not perform the retraining in this project, but we believe with the new data we collect, this could be done straightforwardly.

III. DATASET

A. Dataset Description

Firstly, I will introduce the URL, the main data in the data set. The basic structure and the components of the URL is like the URL example in the following figure 2. In the beginning there will be the protocol of the URL, like http and ftp. We will see http more commonly in our daily life. Also, in some URLs in the dataset, the protocol part with the "http" protocol may be omitted since we use the "http" protocol so much in daily life. You will not see the protocol part of the URLs in some browsers like Google Chrome too. If we need to remember some URLs, we may also just ignore the protocol part. Then it comes to the main body part of the URL----the host name. Speaking more commonly, this part is the content we need to remember when putting a URL into our memories. In more detail, this part points to the terminal source of a website. That means, even if you only type this part into the browser, you can still manage to get to the website you want to browse. The host name is also separated into two parts: the prefix and the domain name. The prefix, usually "www" (World Wide Web), is also omitted in many cases. Of course, there are some URLs which do not use "www" as prefix, like en.wikipedia.org using "en" as prefix. The domain name, which is the main part of the host name, stands for the symbol of the website. In the domain name, there is a top-domain, which is the end of the domain name. It may contain the basic information of the website, like "com" stands for company, "org" stands for non-profit organization, and "gov" stands for government. The part after the domain name, if exists, is the path to the content. When this part exists, the URL stands for a file in the website instead of just a website. This file path in the website is like the path in our own computers. The file name and the extension is also like those in our local systems.

Figure 2. The Components of the URL

Then I will introduce the dataset that I will use. I got the original dataset from the URL https://www.kaggle.com/antonyj453/urldataset. The dataset has a great many of URLs, actually 420464 URLs in total, including malicious ones and benign ones. The number of the URLs is so large that I cannot deal with these URLs only by myself in person easily. It mainly contains two columns, one for the URLs and one for the label of the URLs, which is whether this URL is bad(malicious) or good(benign). What is worth to mention is that as I say above, some URLs in the dataset omit the protocol and the prefix part, which is "http://www." in common. Also, since URLs may be a specific webpage in some websites and some files in the website, whether a URL has the path to the content or not depends on what the URL will point to. For example, a very long URL in the dataset http://www.ime.edu.co/firmasime/Santander/pessoafisica/index1.php?%20id=20,37,43, PM,283,10,10,000000,10,8,2016,Monday.asp has every component I mention above. However, a quite short URL, ipl.hk, also in the dataset with omitting all the unnecessary parts in the URL, has only 6 characters.

B. Data Exploration

In the data exploration part, I try to find some useful features from the datasets. Some of the features may be used in the actual training, but others may not be used.

1) Number of Malicious and Benign URLs

So I first look at the number of URLs. There are 420464 URLs, including 75643 malicious ones and 344821 benign ones. No URL has missing label. Among all the URLs, 17.99% are malicious and 82.01% are benign. I find that the dataset is highly imbalanced. This presents challenges in training the machine learning model for detecting malicious URLs and the evaluation of the model performance.

In figure 3, I draw the number of malicious and benign URLs in a bar plot.

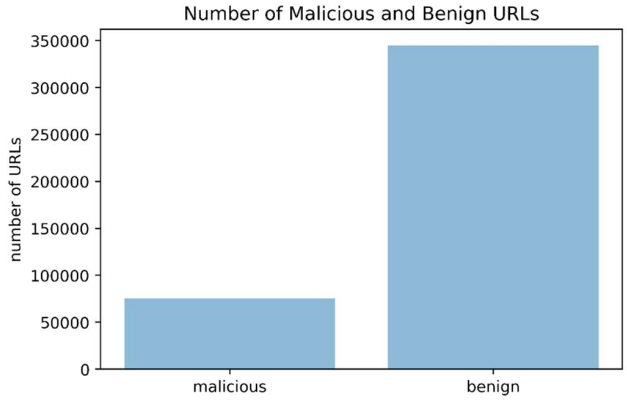

Figure 3. Number of Malicious and Benign URLs

2) Number of Malicious and Benign URLs

Then I come to see how many levels there are in each URL. The levels are separated by the symbol '/'. Among all the URLs, the maximum number of levels is 36, and the minimum number is 1. Overall, no matter whether the URL is malicious or benign, the URLs have the most concentration on level 2. Going beyond level 2, the number of URLs decreases as the number of the levels increases. Compared with the number of URLs in level 2, the number of URLs in level 1 is far less. From the broader outlook, I find that many URLs, no matter whether they are malicious or not, have 2, 3, or 4 levels.

In figure 4, I draw the number of URLs with different numbers of levels in a bar plot.

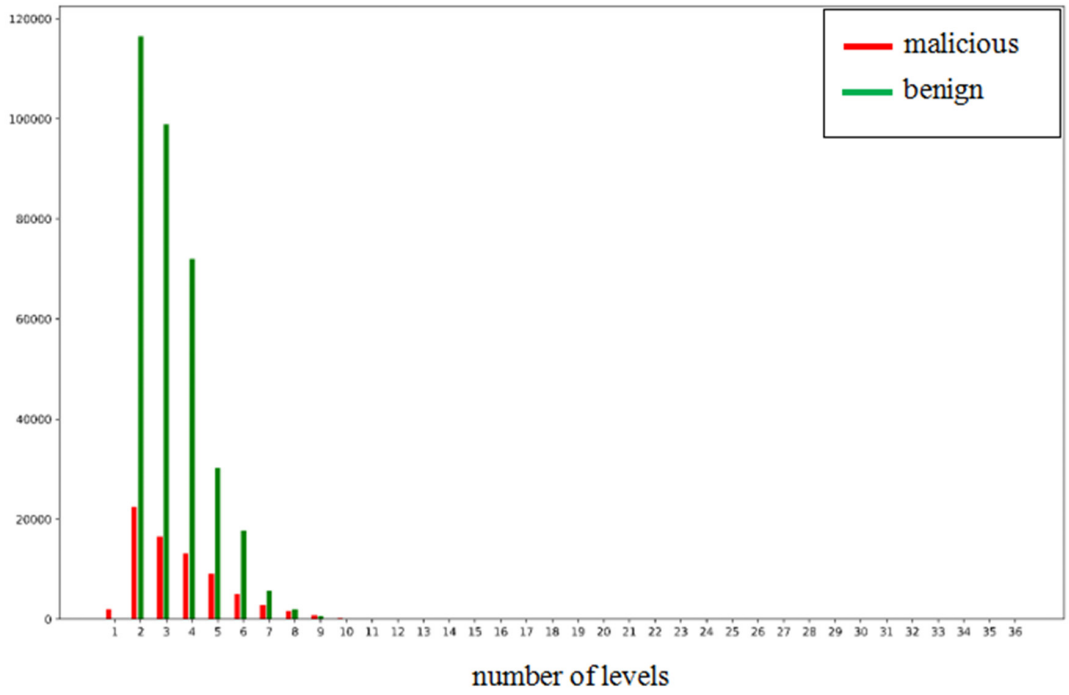

Figure 4. Number of URLs in each number of levels

However, URLs with more than 9 levels are quite few. Just because of this, I cannot see the data of number of URLs with more than 9 levels because the heights of the bars are too small, which leads to the bars of them too short for me to see. As a result, I draw another bar plot, figure 5, for the data of number of URLs with more than 9 levels.

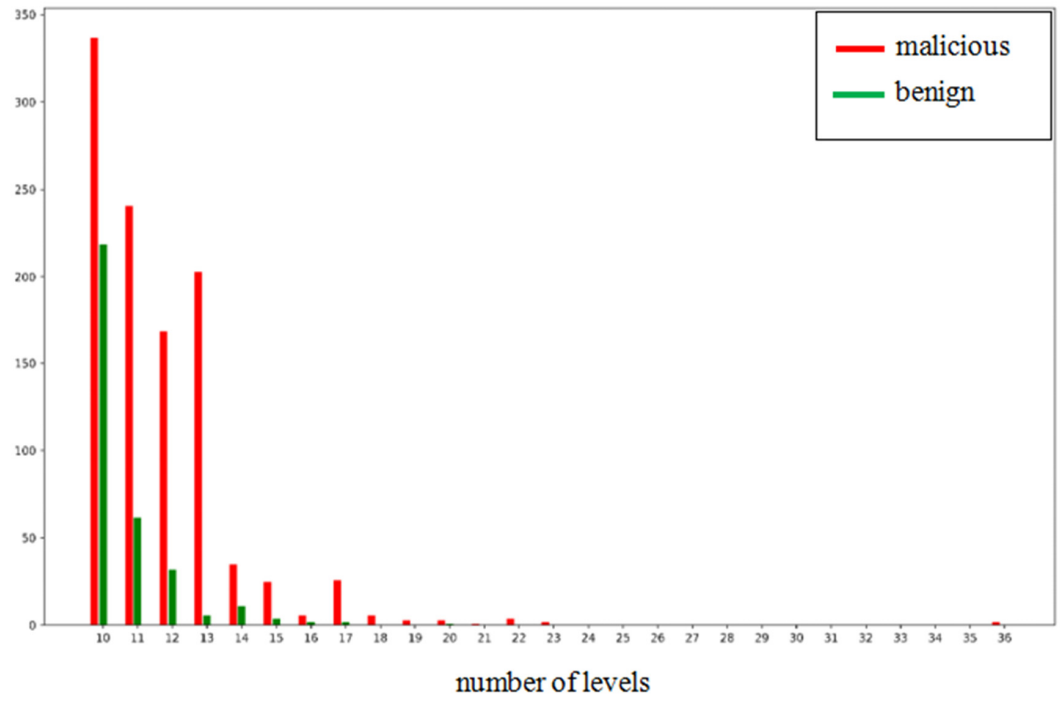

Figure 5. Number of URLs in each number of levels-More than 9 levels

With this graph, I am able to know about the data of number of URLs with more than 9 levels. From a broader outlook, the numbers of URLs will decrease as the number of levels increase, like the relationship mentioned above. However, among this part of URLs, this relationship is only roughly correct. Among this part of URLs, most URLs, no matter whether they are malicious or not, have 10 to 13 levels. Also, I find that even with more total benign URLs than total malicious URLs in the dataset, among URLs with more than 9 levels, the number of malicious URLs goes beyond the number of benign URLs. This may indicate a possible symbol of malicious URLs, which is the existence of too many levels. For example, the number of malicious URLs with 13 levels is 203. However, the number of benign URLs is less than 10, only 6. Given that the number of total benign URLs is about 4 times of that of malicious URLs, the fact that a URL has 13 levels is a strong indicator of malicious URL.

3) Country of each URL

Apart from that, I see some of the URLs have country code top-level domain, and I come to count the number of URLs in each country. However, there are too many countries all over the world and when I put all the countries on the barplot, I even cannot see a single clear bar. As a result, I only list the first ten countries with the most URLs.

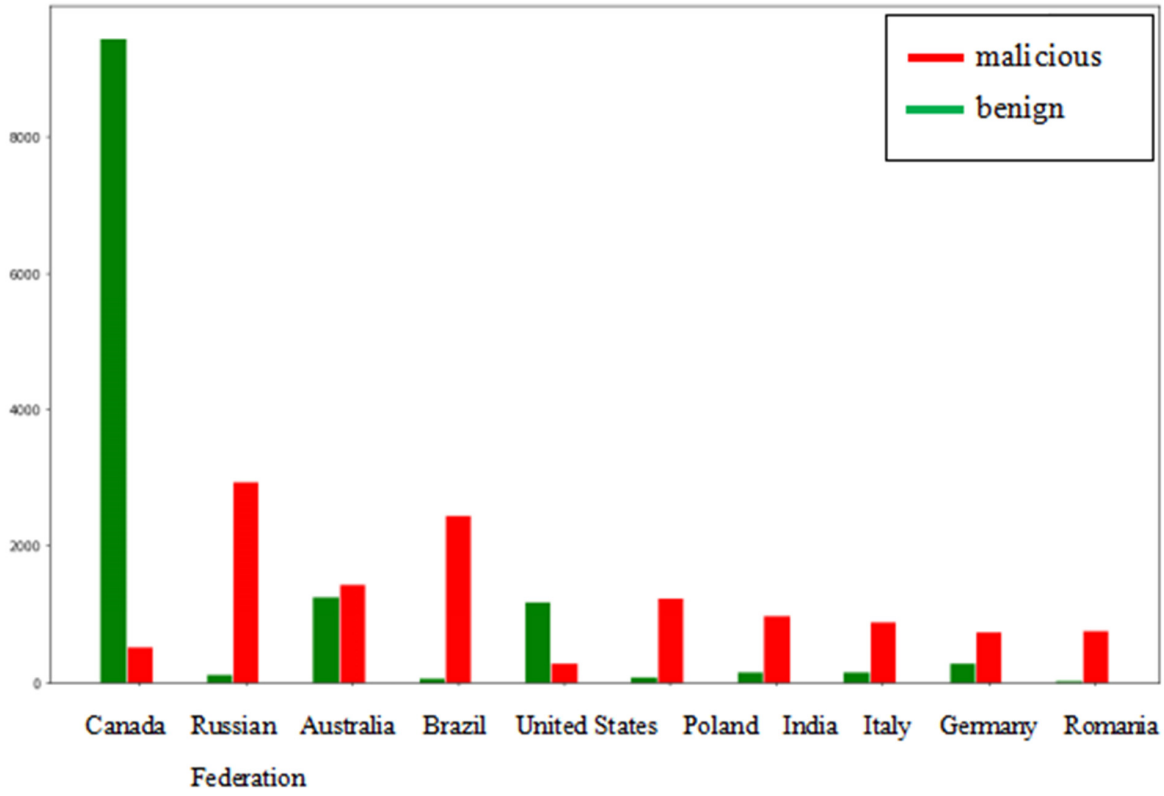

Figure 6. First ten countries with the most URLs in the dataset

In figure 6, I draw the first ten countries with the most URLs in the dataset.

In this graph, I found that most of the Canadian URLs, which take the most places in this dataset, are safe. Similarly, when compared to the number of malicious URLs in the US, the number of benign URLs in the US is much bigger. When it comes to Australian URLs, which take the third most places in the dataset, the number of malicious URLs outnumbers the number of benign URLs by a little bit. However, the original dataset is quite imbalanced. There are much more benign URLs than malicious URLs. And even in this case, in the rest of the ten countries, there are much more malicious URLs than benign URLs. For example, the numbers of benign URLs in Russian Federation, Brazil, and Poland are all quite small while the numbers of malicious URLs in these countries are quite huge. I even cannot see the bar of the number of benign URLs in Romania, which means the number is too tiny. According to which country the URL is from, or speaking more specifically, the country code top-domain of the URL, maybe I can give roughly correct prediction of the label of the URLs.

Of course, not all URLs have country code top-level domain. Many URLs just end with .com, .net, .org, and so on. Since in the actual training part I use the country code top-level domain as a feature to train the models, for those URLs without a country code top-level domain, I give them the code of the number of the last country in the country list and plus one. In this case, I get 185 different countries in my dataset, so I give those URLs without country information the number 186.

4) Special Characters in the URLs

Moreover, I choose to look at the special characters in the URLs. We will often see letters and numbers in the URLs. As the special existence in the URLs, I think they may be related

to the safety of the URLs. Consequently, I count the number of special characters in each URL and see the distribution of data in malicious and benign URLs separately.

To know which characters are special characters in the URLs, I come to look up for some sources and find a comparatively entire list of special characters in the URL https://secure.n-able.com/webhelp/NC_9-1-0_SO_en/Content/SA_docs/API_Level_Integration/API_Integration_URLEncoding.html.

To see the distribution more clearly, in figure 7, I make the boxplot of the distribution of the number of special characters in malicious and benign URLs.

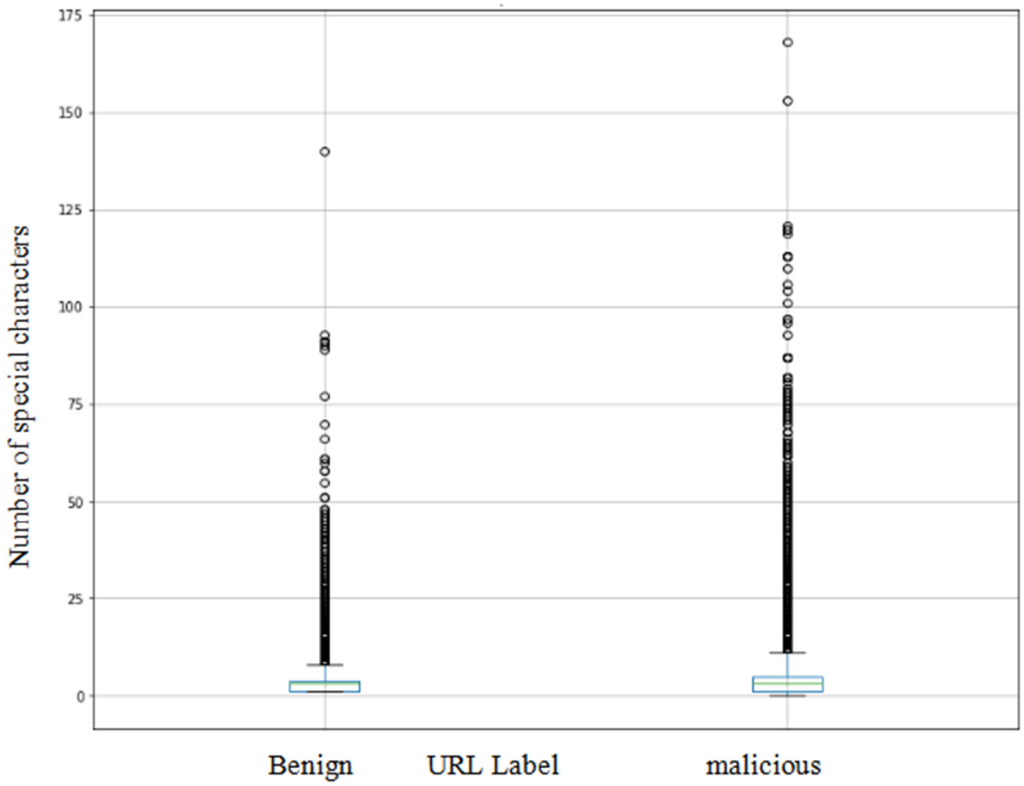

Figure 7. Number of special characters vs Label

From this graph, I find out that most of the URLs have less than twenty special characters. In the graph, the two boxes are both in a quite low position. However, there are still a lot of outliers which have more special characters than most URLs do. Comparing the two distributions of number of special characters, I find that malicious URLs do have more special characters than benign URLs do generally, and this may imply that a URL with many special characters is highly possible to be a malicious URL.

To compare the two distributions in details, I use the Student's T-Test. The T-Test is used to test the difference between the means of two normally distributed population. In the procedure, we will calculate a p-value according to the means, variances, and the total numbers of the data of the two population. The less the p-value, the more the two datasets are different from each other. By using the scipy.stats package in Python, I can easily calculate the p-value of the two datasets. However, I get a p-value of 0 in the end. At first I thought there must be some problem with it, but even if I used the function of the entire procedure of calculating the p-value, I still got a 0.

As a result, I come to use another method of comparing the distributions----Mann-Whitney U-Test. Like Student's T-Test, Mann-Whitney U-Test also tests the difference of the two means of the two distribution. What it is different from the T-Test is that it is a nonparametric test and it does not assume that the distribution is Gaussian. In U-Test, we will also calculate a p-value from the original data. To calculate the p-value in Mann-Whitney U-Test, I still import scipy.stats to help me easily get the value. To my surprise, I still get a p-value of 0 in U-Test. According to the similar results in Student's T-Test, I think that this is not an occasional phenomenon.

After thinking about this, I consider that it may be because the two datasets are not normally distributed. After all, the number of the data on each side of the medians in both datasets are not roughly equal. Finally, I found the data in these two datasets spread out much.

To find out the reasons in more detail, I draw the graph of distribution of URLs by special characters in histograms, with primitive data and normalized data. In figure 8 and figure 9 , I draw original and normalized histograms of number of special characters for malicious and benign URLs.

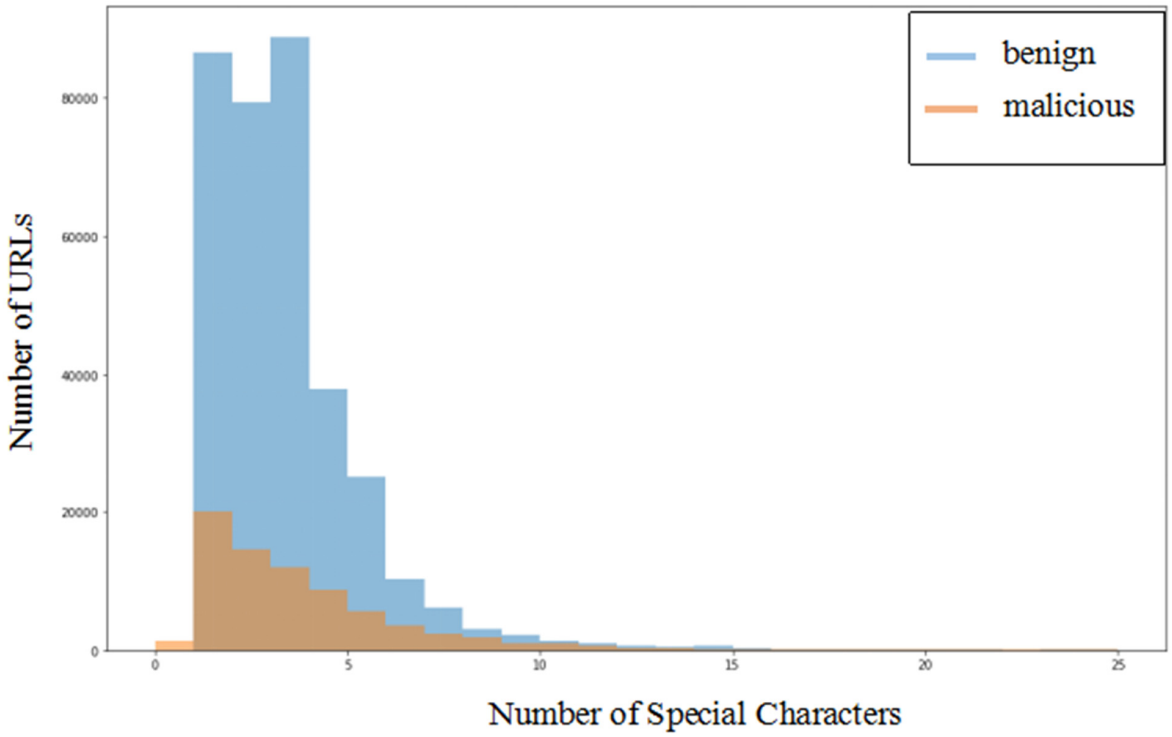

Figure 8. Histograms of number of two groups of URLs for each number of special characters

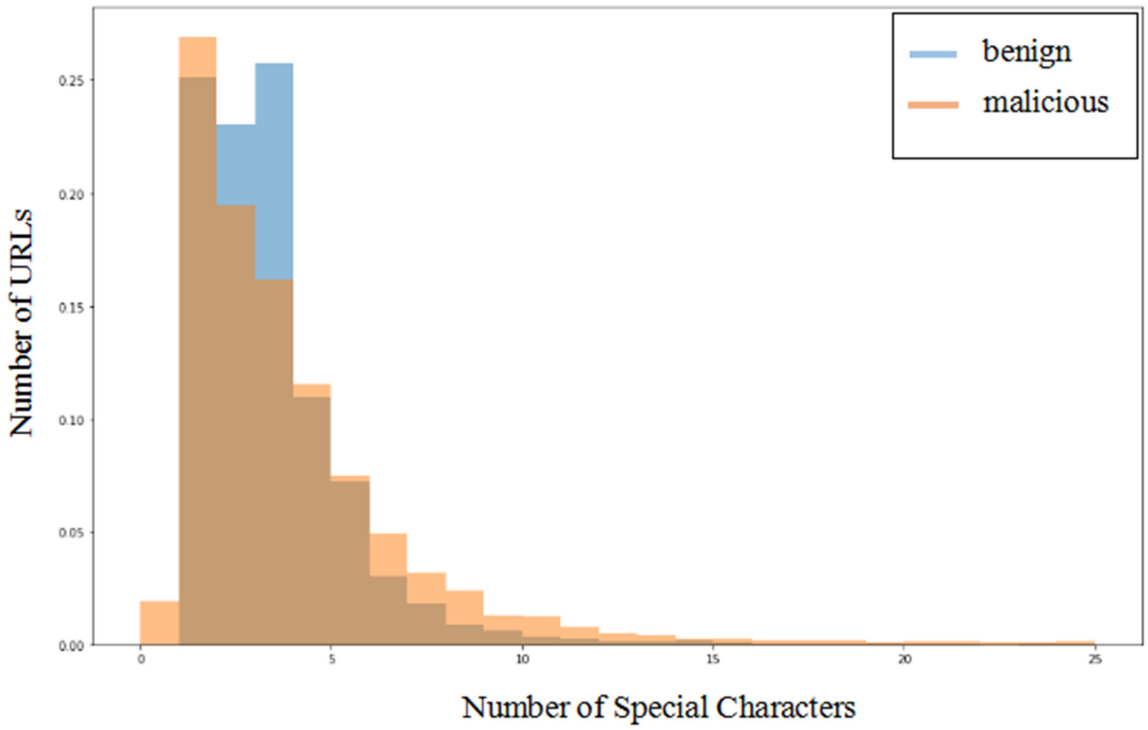

Figure 9. Normalized histograms of numbers of two groups of URLs for each number of special characters

From the two histograms, I find out that neither of the two distributions of the grouped URLs is Gaussian Distribution. They all show a rough pattern that as the number of special characters' increases, the number of URLs decreases, except when the number of special characters is zero. The numbers of both URLs with no special characters are quite low. Moreover,

the two distributions do show a great difference at number of special characters from 0 to 2. In benign URLs, the numbers of URLs with 1, 2, and 3 special characters are similar. However, in malicious URLs, the numbers of URLs with 1, 2, and 3 special characters' decrease accordingly. Also, when it comes to the numbers of URLs with more than 5 special characters, though in number the benign URLs are more than malicious URLs, it is because of their huge total number of URLs. In percentage, the malicious URLs with more than 5 characters are more than benign URLs, which may indicate than URLs with more than 5 characters are more likely to be malicious.

However, in these two figures, I also found that the numbers of URLs with more than 20 special characters are too low that I cannot see them clearly.

As a result, in figure 10 and figure 11, I draw the two histograms for URLs with more than 20 special characters. Since the URL with the most special characters has 175 special characters, I put 5 continuous numbers of special characters in a group to show the bars more clearly.

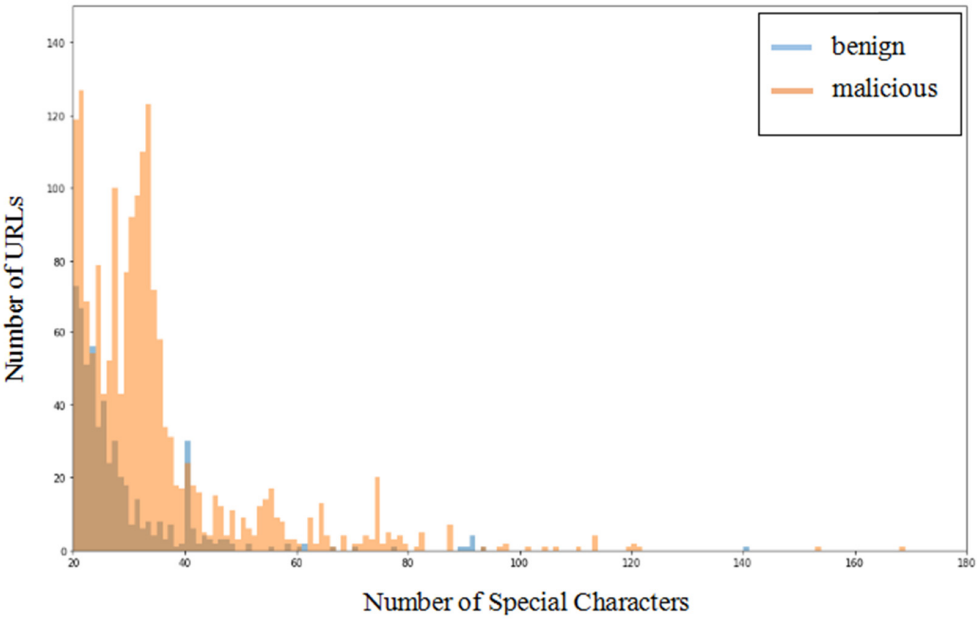

Figure 10. Histograms of number of two groups of URLs for more than 20 special characters

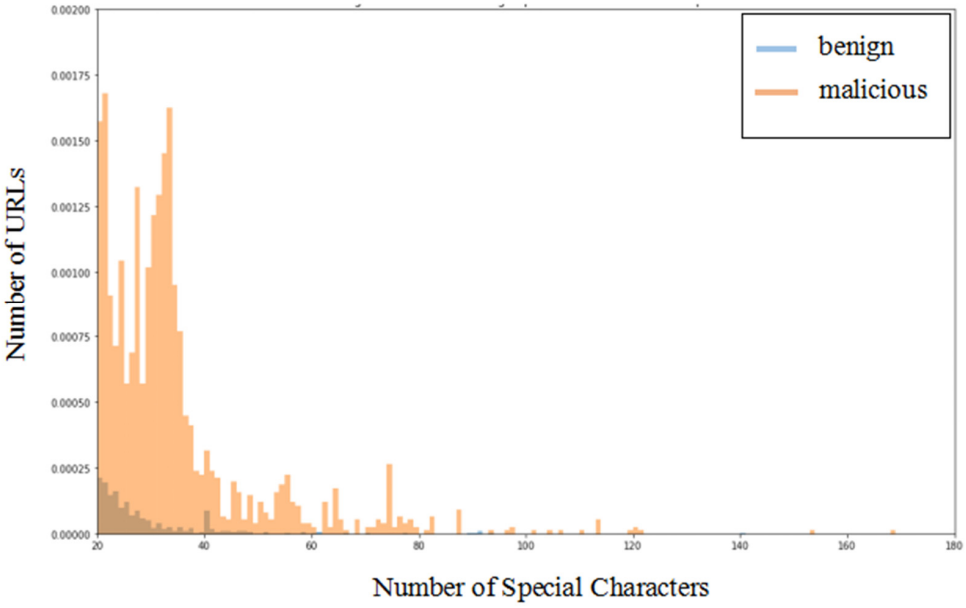

Figure 11. Normalized histograms of number of two groups of URLs for more than 20 special characters

In these two figures, I find that among this part of URLs, no matter whether they are malicious or benign, URLs with 20 to 40 special characters take much more place than the URLs with more than 40 special characters do. Generally, the fact still corresponds with a relationship that the more the special characters, the less the number of URLs. However, In the middle of the data of 20 to 40, more specifically, 22 to 31, the numbers of malicious URLs are much less than that of URLs with 20, 21, and 32, 33 special characters.

Comparing the two groups of URLs, I find that much more percent of the malicious URLs are among this part of URLs than benign URLs do. This does show that URLs with many special characters are much more likely to be malicious URLs.

5) Length of the URLs

Last but not the least, I will focus on the most easy-to-see feature of the URLs, the length of the URLs. Sometimes when browsing the Internet using the browser, we may find that the URL in the address bar quite long, even the whole address bar cannot show the entire URL. Also, we may find that sometimes the URL in the address bar will be quite short, and this may happen when entering the home page of websites with short web addresses.

To find the distribution of the lengths of the URLs, especially in groups of benign URLs and malicious URLs, I make another boxplot figure 12 to help me to observe the distribution.

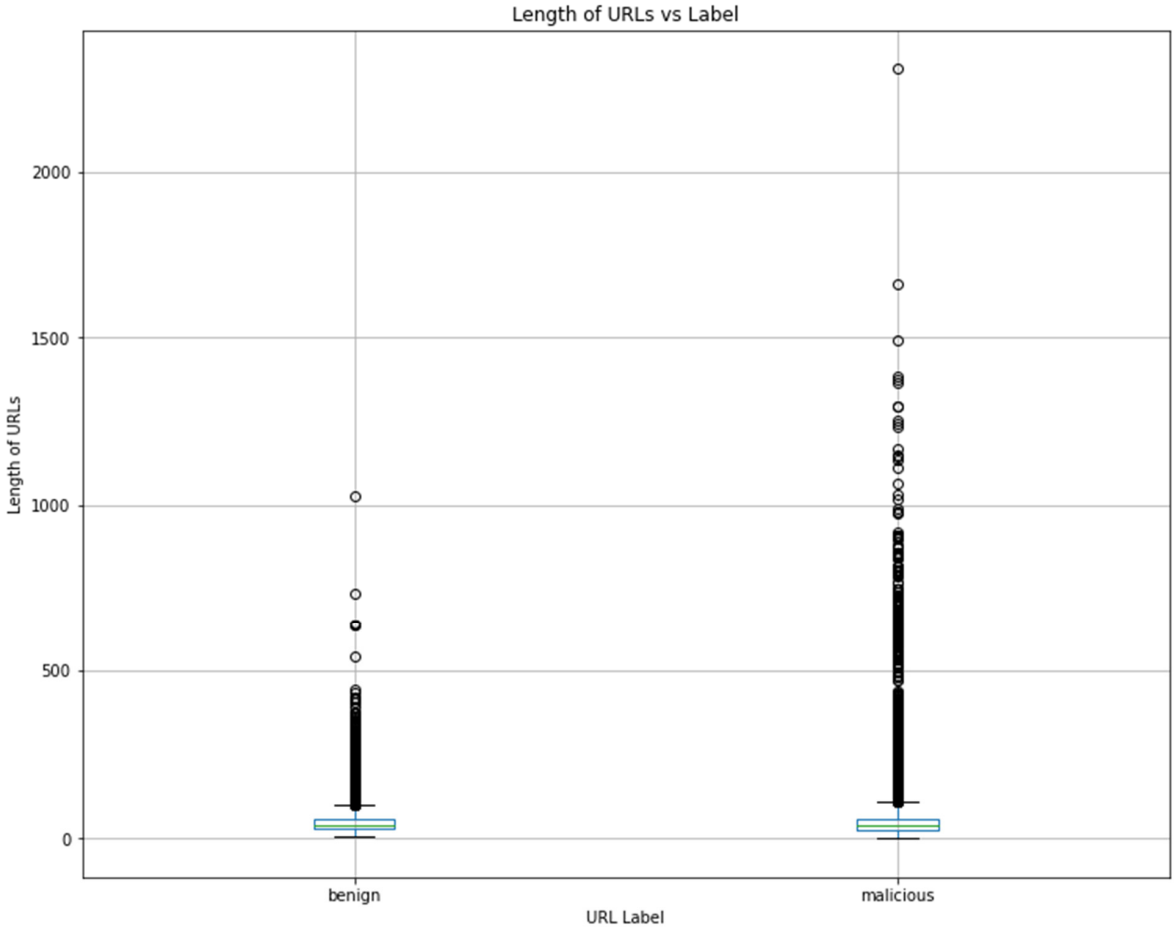

Figure 12. Length of URLs vs Label

Using this figure, I find that most of the URLs, no matter whether they are malicious or benign, have a length of less than 200 approximately. The positions of the two boxes are very low. However, there are still many outliers which are much longer in length.

Comparing the two groups of URLs, I find that the maximum of the length of malicious URLs is much larger than that of the benign URLs, which may indicate that in general the length of malicious URLs is larger than that of benign URLs.

To compare the two distributions of the length of URLs, I come to calculate the maximum, median, minimum, and mean of the two groups of URLs. By using scipy.stats, I can calculate these numbers quickly. And I get the result that the maximums of benign and malicious URLs are 1025 and 2307 respectively, the medians are 42 and 38 respectively, the minimums are 6 and 1 respectively, and the means are 47.00 and 54.45 respectively.

With the four pairs of numbers, I find that though through the graph it may seem that usually the lengths of malicious URLs are larger than those of benign URLs, with the specific

statistical data, it is not like what it seems like at all. As a result, the length of the URLs cannot be used as a possible dependable evidence to predict whether the URL is malicious or benign.

IV. METHODS

A. Procedure

1. What I will do is roughly separated into the three following steps:
2. Find appropriate and typical features in the URLs for algorithms' calculation.
3. Try different algorithms to train the data.

Use suitable evaluation methods to evaluate the model and improve it if possible.

The ultimate goal of using machine learning to solve this problem is to train a model with high accuracy and dependency to predict the status of new URLs users meet. All the three steps are important to achieve the ultimate goal. The first step aims to find good features to represent the original URLs. The goal of the second step is to find a good way to train the model. The third step focuses on using appropriate method to remark on my model to see the performance of my model more clearly.

In the first step, I need to find the lists of appropriate features for each URL and put them into a list. To get some features of URLs, I observe the data of URLs. One point which can be easily come up with is the domain name part of the URL. In this way, another part of the URL, the path, can also be a feature to utilize. To store the features, I want to use a two-dimensional array: one dimension for the URLs, and one dimension for the specific information of features of one URL.

For the second step, I will try Naïve Bayes and Logistic Regression, two algorithms to train the model. This will be discussed in more details in the subsequent parts of the essay.

In the final step, using suitable evaluating methods to review my model, one easy and common-used evaluating method is the accuracy of the model. That is, simply looking at the percentage of the right predictions the model makes. However, just concentrating on the accuracy of the model will not always make the model better because the ultimate prediction has only two results: right and wrong. Maybe the model calculates wrongly in the middle steps but with many mistakes the final result gets correct. In this case, the accuracy is not the best method to tell the performance of the model. So how can I test the performance of the model? The performance is based on accuracy in fact, but there is no need for me to see the accuracy in the outlook of the entire data. Perhaps I can get accuracy of some part of the data. And the specific methods will also be discussed in details in the subsequent part of the essay.

B. Feature Engineering

After doing the data exploration, I find some useful features to help me to train the model.

Here is the table of the information of the features, table 1

TABLE I. EXPLANATIONS OF MANUALLY-CRAFTED FEATURES

FEATURE NAME	EXPLANATION
NUMBER OF LEVELS	Using '/' as the symbol of dividing different levels, count the number of levels in each URL.
COUNTRIES	Where each URL comes from, getting from the country top-domain name (if exists).
NUMBER OF SPECIAL CHARACTERS	The number of special characters in each URL. The list of special characters comes from https://secure.n-able.com/webhelp/NC_9-1-0_SO_en/Content/SA_docs/API_Level_Integration/API_Integration_URLEncoding.html.
TOKEN COUNTS	The number of each token in each URL. A token is defined as a meaningful part of the URL and the tokens are counted in the whole URL dataset.

C. Token Counts

Counting the numbers of tokens in a URL can be quite difficult since there are a lot of symbols which can separate two tokens. If we just manually select the tokens and count the numbers, it can be such a waste of time. To get all the tokens more efficiently, I apply the TfidfVectorizer in python.

Tf-idf, which refers to term frequency-inverse document frequency, is a numerical value in statistics to show the importance of a word to a document. [7] It includes two values: term frequency(tf) and inverse document frequency(idf), and the tf-idf=tf*idf. Tf means the number of appearance of one word in a document, and idf is like the weight of the tf. Since each document has different lengths, to compare tfs of different documents, we need to normalize tfs, which is to divide the original tf by the total tokens in the document. Idf is calculated by $\ln\left(\frac{total\ documents}{(the\ number\ of\ documents\ that\ include\ a\ specific\ token)+1}\right)$. The +1 here is to prevent the phenomenon that the token does not appear in any of the documents. The more common a token is, the less the value of idf. Times tf and idf to get tf-idf, the bigger the value of tf-idf, the more important the token to the entire documents.

The TfidfVectorizer in python has many parameters and attributes. [8,9] To get the tokens easily, I just use fit_transform to get all the tokens and their numbers of appearance in the URL dataset using the default tokenizer which is used to separate tokens.

However, the default tokenizer may not be suitable for dividing the tokens in URLs, so I also make a special tokenizer made for recognizing tokens in the URLs. The code is shown as following:

```
def makeTokens(f):
    tkns_BySlash = str(f.encode('utf-8')).split('/')
# make tokens after splitting by slash
        total_Tokens = []
        for i in tkns_BySlash:
                tokens = str(i).split('-')
# make tokens after splitting by dash
                tkns_ByDot = []
                for j in range(0,len(tokens)):
                        temp_Tokens = str(tokens[j]).split('.')
# make tokens after splitting by dot
                        tkns_ByDot = tkns_ByDot +
temp_Tokens
                total_Tokens = total_Tokens + tokens +
tkns_ByDot
    total_Tokens = list(set(total_Tokens))
#remove redundant tokens
        if 'com' in total_Tokens:
                total_Tokens.remove('com')
#removing .com since it occurs a lot of times and it
should not be included in our features
        return total_Tokens
```

As we can see in the code, it splits the tokens by slash (/), dash(-), and dot(.). It also removes "com" from the token list since it is quite common to see and cannot be used to judge whether a URL is malicious or not.

In the actual experiment, I will use both the default and specially-made tokenizers and compare these two results to see which tokenizer better helps the model to judge malicious URLs.

D. Naïve Bayes

Firstly, I use Naïve Bayes algorithm to train the model. As introduced in the Introduction, this algorithm uses the Bayes theorem. The Bayes theorem is known as

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \quad (1)$$

where $P(A|B)\ and\ P(B|A)$ are conditional probabilities and $P(A)\ and\ P(B)$ are prior probabilities. Prior probability is the probability distribution of an uncertain quantity before observe the actual data in detail. It can be got by calculating the percentage of that uncertain quantity in the data. Conditional probability is the probability of one event given that another event happens. Using the Bayes theorem, we can calculate the probabilities and thus get the final results.

The Naïve Bayes algorithm acts like this. [5,10] First, calculate the prior probabilities of malicious and benign URLs, which in this case are the percentages of malicious and benign URLs in the entire dataset. Second, take the number of levels in each URL as an example feature, calculate the conditional probabilities of each URL with different levels in groups of malicious and benign URLs. Third, use the Bayes theorem to calculate the conditional probabilities of each URL in different statuses in each number of levels, and use these probabilities to predict the status of URLs with any other number of levels.

More specifically, we need to consider all the features, that means, to calculate the conditional probability $P(s|f1, f2, f3\ \dots)$, where s is the status of the URL and fn is the features of the URLs. According to the Bayes theorem, we can know that

$$P(s|f) = \frac{P(f|s)P(s)}{P(f)} \quad (2)$$

where f is the list of all the features. Since we know the features and their information in the dataset, the prior probabilities of the features are certain and we can calculate them from the dataset. Thus, we only need to know the value of $P(f1, f2, f3, \dots|s)P(s)$, which actually equals to $P(s, f1, f2, f3, \dots)$, the joint probability. However, if we plug chain rule in the formula,

$$P(f1, f2, f3, \dots|s)P(s) = \quad P(f1|s)P(f2, f3, \dots|s, f1)P(s) \quad (3)$$

Continue plugging in like this, we can get the original formula equals to

$$P(f1|s)P(f2|s, f1)P(f3, \dots|s, f1, f2) = P(f1|s)$$
$$P(f2|s, f1)P(f3|s, f1, f2)P(f4\ \dots|s, f1, f2, f3)P(s) \quad (4)$$

and finally becomes

$$P(f1|s)P(f2|s, f1)P(f3|s, f1, f2)$$
$$\dots P\big(fn\big|s, f1, f2, f3, \dots, f(n-1)\big)P(s) \quad (5)$$

And because of the assumption that each event is independent from each other, each feature will not influence each other and thus the conditional probabilities in this case are equal to the probabilities of all the features themselves, which means $P(A|B) = P(A)\ and\ P(B|A) = P(B)$, and in this case

$$P(f1|s)P(f2|s, f1)P(f3|s, f1, f2)$$
$$\dots P\big(fn\big|s, f1, f2, f3, \dots, f(n-1)\big)P(s)$$
$$= P(f1|s)P(f2|s)P(f3|s)\dots P(fn|s)P(s) \quad (6)$$

With both the joint probability and the prior probability calculated, we can get the conditional probability we want.

To use the Naïve Bayes model in Python, I import the sklearn.naive_bayes.GaussianNB(). However, during the training procedure, I find that Gaussian model does not support sparse matrix, but the feature token counts needs to store in the sparse matrix. As a result, I also apply sklearn.naive_bayes.Multinomial() to test all the possibilities in the Naïve Bayes part.

To introduce the Multinomial Naïve Bayes, I will first introduce multinomial distribution. Multinomial distribution is a generalization of binomial distribution. It is used to describe experiments like the possibilities of rolling each number a specific number of times while rolling a die. Given n independent trials each of which leads to a success for exactly one of multiple categories, with each category having a fixed success possibility, the multinomial distribution tells the

possibility of any combination of numbers of successes for different classes.

Different from Gaussian Naïve Bayes which assumes that the distribution of the data is Gaussian distribution, Multinomial Naïve Bayes assumes that the distribution of the data is multinomial distribution, which may fit more types of data especially counts since multinomial distribution describes the probability of observing counts among a number of categories. [11]

I train the model with different combinations of the features. The first combination includes one feature of levels of URLs. The second combination has two features including levels of URLs and countries. The third combination contains three features including levels of URLs, countries, and the number of special characters. I also combine the three features with tokens. All these features, like level of URLs, countries, and number of special characters, are explained in table 1 in the feature engineering part.

E. Logistic Regression

Another algorithm I use to train the model is the logistic regression.[6,12] It uses a function called logistic function or sigmoid function: $y = \frac{1}{1+e^{-t}}$. What is special about this function is that the range of the function is (0,1), and for the domain $(-\infty,0)$, the range is (0,0.5), and for the domain $(0,\infty)$, the range is (0.5,1). This characteristic allows us to see the logistic function as a binary function. We also need to use another function, which is $t = \omega^T x + b$, and thus the final formula becomes $y = \frac{1}{1+e^{-(\omega^T x+b)}}$. More specifically, we first need to apply linear transformation to the features, turning all the features $f1, f2$, and so on into a vector, which is called x here. Then we apply nonlinear transformation to put these features into the logistic function. When we train the model, the weight of the features, which is ω in the formula, will change to fit the data. While training, logistic regression also uses another algorithm called gradient descent and a likelihood function. The likelihood function is used to check the correctness of prediction of the model. To maximize the result of the likelihood function, gradient descent is applied for optimization. It utilizes a finding that if a real-valued function $F(x)$ is defined and differentiable at a point a, then the value of $F(x)$ decrease most quickly while going from point a in the direction of negative gradient of F at a. With this algorithm, we can get a local minimum fast and thus help us to fit the weight of variables in the model to the training dataset by just repeat using the formula that $b = a - step * \frac{d}{dx}F(a)$ where b is the next position and $step$ is the length of the step.

To use the Logistic Regression model in Python, I import the sklearn.linear_model.LogisticRegression().

There are a lot of parameters in the function LogisticRegression(), like penalty, fit_intercept, and n_jobs. [13] In this project, I will try to tune two of these parameters, max_iter and class_weight, to help improve the performance of the model. Max_iter controls the maximum number of iteration while training the model. Class_weight means to get weighted numbers of each class of data from the dataset. For example, if we set max_iter to 200, and class_weight to [1,4] (1 for benign and 4 for malicious) while training the model to detect malicious URLs, then while the model is fitting the data, it will iterate 200 times at most, and it will get a scale of 1:4 of benign and malicious URLs from the training dataset (if the scale of malicious and benign URLs in the dataset is 1:1) to train itself. What is worthwhile to mention is that while getting the data in a certain scale by controlling the class_weight, the actual weights of data will be the product of the given class_weight and the original scale of the data. In this case, using the dataset mentioned in this project, if I set class_weight to [1,4] (1 for benign and 4 for malicious), the model will get nearly the same number of malicious and benign URLs since the original scale of benign and malicious URLs in the dataset is 4:1.

In this project, I will try different combinations of features including token counts, and tune the hyper parameters to try to find a model with the best performance.

F. Performance Evaluation

To evaluate the performance of the models, I use several different evaluating methods.

The first one is a quite common method: accuracy. It just calculates the percentage of correct predictions in the total predictions.

Besides normal accuracy, I also use F1 score. [14,15] To introduce F1 score, I will tell about the precision and recall first. After the model finishes predicting, the entire prediction of the test dataset can have four results: predict true and actually it is true (true positive); predict true while actually it is false(false positive); predict false and actually it is false(true negative); predict false while it is true(false negative). Precision is the number of true positives divided by how many predictions the model makes is true, and recall is the number of true positives divided by how many data are actually true. F1 score combines the precision and recall. That is, calculating the harmonic mean of precision and recall. Showing the F1 score as a formula, it is $F_1 = \frac{2}{precision^{-1}+recall^{-1}}$. Since it takes both precision and recall into consideration, it can show the accuracy in balance.

The last evaluating method I use is the area under the curve. [16,17,18] The curve here means the ROC curve, which stands for receiver operating characteristic curve. To draw the ROC curve, we need to have ROC space first.

Figure 13 shows what a ROC space looks like.

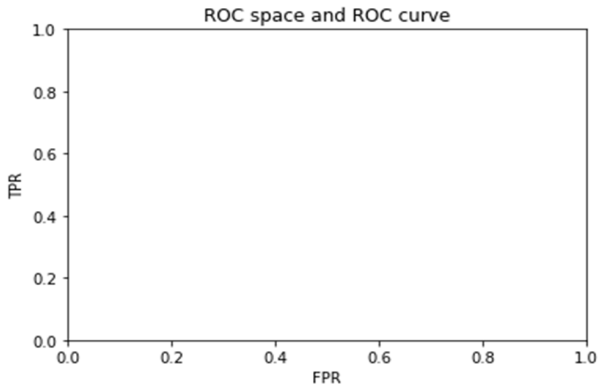

Figure 13. ROC Space and ROC curve

The X axis of the ROC space is for the false positive rate (FPR) and the Y axis of that is for the true positive rate (TPR). Both ranges for X axis and Y axis are [0,1]. Here, FPR is equal to the number of false positives divided by the total number of actual negative data, and TPR is equal to the number of true positives divided by the total number of actual positive data. When we train to get a model, we can get the TPR and FPR of the model and mark the point in the ROC space.

However, the performance can still be improved with fixed algorithm and features used. Sometimes we can change the sensitivity of the model to change the results of the model. Thus, we need a threshold here to help modify the sensitivity and the specificity of the model. [19] If we change the threshold, the standard of predicting true and false may also change and thus influence the TPR and FPR. Usually, when we decrease the threshold, the result of predicting will contain more positives than that without decreasing the threshold. However, the distributions of actual positives and negatives can overlap. In this case, TPR and FPR will change in the same direction. With the changing threshold, the TPR and FPR of the same model change, and that forms a curve, which is just the ROC curve. And area under the curve (AUC) just calculates the area under the ROC curve in the ROC space. In other words, assume that p1 equals to the probability of the model predicting actual positives correctly and p2 equals to the probability of the model predicting actual negatives wrongly, then AUC equals to the probability of p1 is larger than p2.

Apart from those evaluating methods, I attempt to train the models repeatedly with selecting different part of the original dataset as training data, calculate the corresponding evaluating scores, and get the mean and standard deviation of the evaluating scores of one model to find its comprehensive score. More specifically, while separating the training dataset and the test dataset, I will randomly divide the dataset for many times, and for each time, I will train the model with the training dataset and get a result evaluating score. After getting many scores of this model, I will calculate the mean and the standard deviation of the scores to let me know the general performance of the model. This method helps me to see that whether a model performs well occasionally or not, and see its true performance, effectively decreasing the deviation of the scores.

G. Hyperparameter Tuning

Tuning the hyper parameters is to modify the parameter of the model to improve its performance. While tuning, we may not know whether a step of tuning enhance the performance of the model or not, so we may want to use the test dataset to test the performance. Nevertheless, the test dataset is only used to test the performance of the model, so we cannot use it to train the model. Comparing the model to a student, we can see the training procedure as students learning knowledge and doing homework. In this case, using the test data to test the performance of the model becomes using test papers to test whether the student masters the knowledge that he or she learns or not, so it is not proper to use test dataset in training. However, though students cannot use the test to train himself or herself in advance, he or she can use some mock exams to simulate the tests to see whether he or she is good or not. In the model training, this can be achieved by cross validation.

Cross validation means to separate the training dataset into two parts: one part called train-sub is used for training; the other part called validation set is used for “mock tests”. [20] When a model finishes training using the train-sub, we can use the validation to test its performance and see whether the result is good or not.

With cross validation, we can both train the model and make an estimation of the performance of the model. And then, we can tune the hyper parameters to make the performance better.

However, the hyper parameter are mostly numbers without ranges. We cannot use all the numbers and test the performance each time. In this case, we need to set the ranges manually and find the optimum number to use. While looking for the optimum number for the hyper parameters, we need to evaluate the model each time to see its performance to judge the efficiency of the hyper parameters.

After tuning to the optimum condition, we can use test dataset to test the model, which can get the best results of the model.

For Logistic Regression, I will tune max_iter, which is the maximum number of iterations, and class_weight, which is the weights to get different classes of the data.

All the features I test are introduced in more detail in the former part of the report.

In figure14, I draw the AUC scores of each test of 50 tests for before and after hyper parameter tuning respectively.

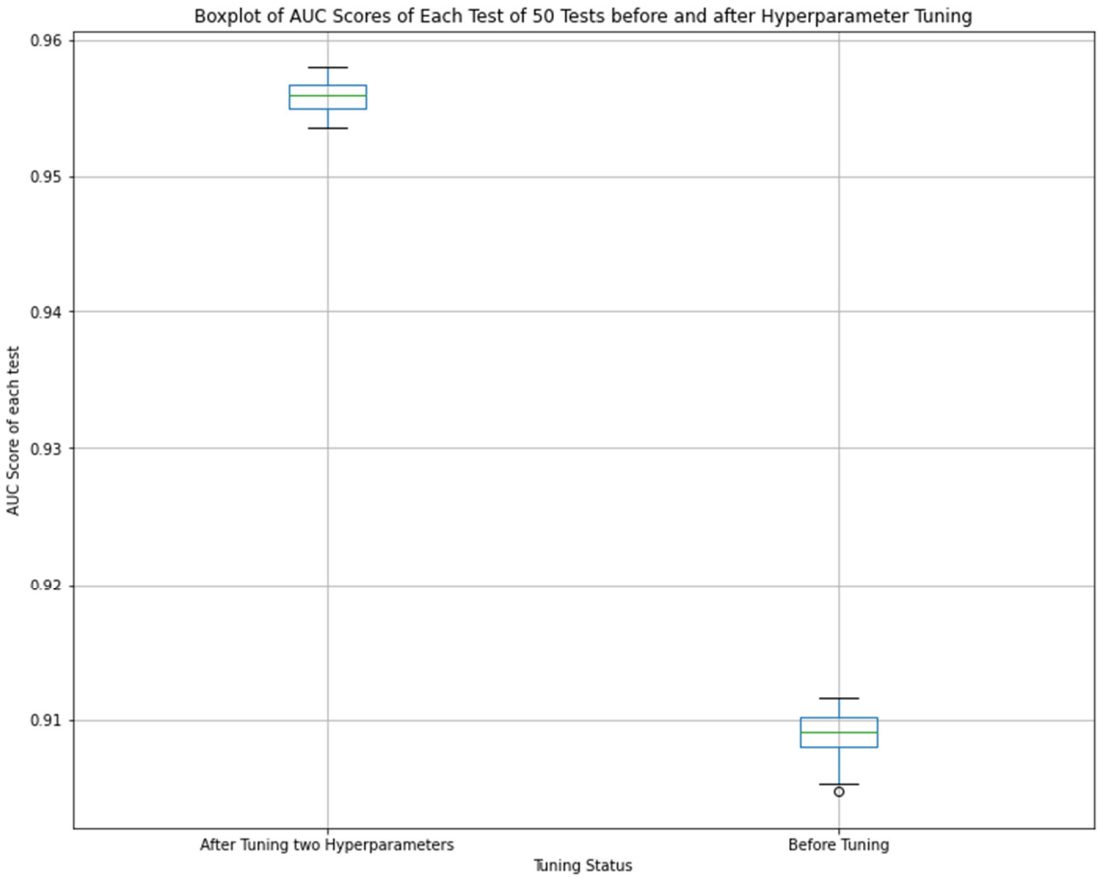

Figure 14. Boxplot of AUC Scores of Each Test of 50 Tests before and after Hyperparameter Tuning

For this graph, we can see that after tuning max_iter and class_weight, the AUC scores can reach more than 0.95, even nearly 0.96, while without hyperparameter tuning, the AUC scores of the model are only about 0.91. We can know that how much improvement hyperparameter tuning offers in this boxplot.

V. Results & Discussions

In this part, I will show all the results I get from my models, and try to find the reasons of some unexpected results. What is worth to mention is that all the decimal numbers are rounded to 5 decimal places.

With two algorithms, several combinations of features, some hyper parameters to tune, and a few evaluation methods, I get the results of different models I train.

For the Gaussian Naïve Bayes models, I only put one feature—number of levels—first to test the performance. What I need to mention is that I do the training of this part all with training by using random combinations of URLs in the dataset for 100 times and no hyper parameter tuning. After training and testing, I get the mean accuracy, F1 score, and AUC score of 0.81461, 0.14538, and 0.53089, and the standard deviations of them are 0.00129, 0.00362, and 0.00124 respectively. Then I add the feature countries into the training data and test the performance. Then I get the means 0.83374 for accuracy, 0.41828 for F1 score, and 0.63799 for AUC score, and the standard deviations 0.00115 for accuracy, 0.00364 for F1 score, and 0.00182 for AUC score. The standard deviations do not change a lot. However, looking at the means, though the accuracy only increases about 0.02, the other two evaluating scores both increase a lot with the addition of the country feature, especially the F1 score that increases to nearly 3 times. Seeing such a huge improvement, I add the last manually-crafted feature—number of special characters, but what surprises me is that the means of the scores are 0.82775 for accuracy, 0.43367 for F1 score, and 0.64773 for AUC score, and the standard deviations are 0.00120, 0.00341, and 0.00185 respectively. The standard deviations still do not change a lot. However, the means of F1 score and AUC score both only increase 0.01, and the accuracy even drops a little bit. Nevertheless, I still see this addition as an improvement since the F1 score and AUC score which reflect more specific information of the performance increase, and the decrease of the accuracy may be because of the specific URL dataset. If using these two models to predict all the URLs in the world, I think the model with more features will predict more accurately. To see the performance of the model in more detail, I draw the ROC curve of this model with randomly separated training data and test data to train and test the model in figure 15.

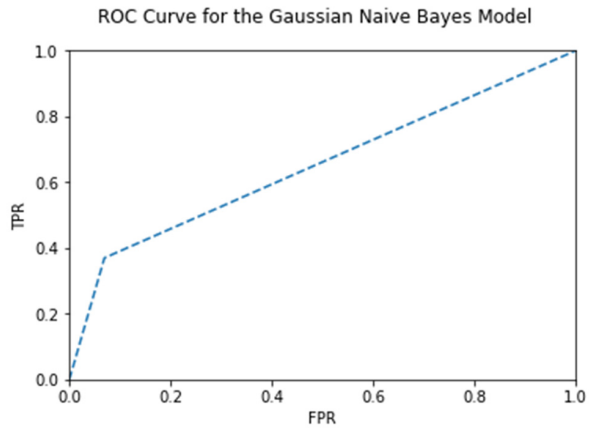

Figure 15. ROC Curve for the Gaussian Naïve Bayes Model

In this graph, we can see a polyline, which is the ROC curve. It is still distant from the up and left side of the plot, which means the performance is not so good. In this case, the AUC score is 0.65019.

There is still another feature—tokens—I do not apply in the Naïve Bayes models, so I add this to the Naïve Bayes model with three manually-crafted features. Since there are two tokenizers, I add them to two models respectively to compare the performance.

However, when I try to put the feature of token counts in the Gaussian model to train, I find that it will be an error because the Gaussian model does not support sparse matrix and I can only store the tokens in a sparse matrix.[21] After looking up some sources, I find that one member of the scikit-learn explains that "Another point: one reason sparse inputs are not implemented in GaussianNB is that very sparse data almost certainly does not meet the assumptions of the algorithm – when the bulk of the values are zero, a simple Gaussian is not a good fit to the data, and will almost never lead to a useful classification."[22]

As a result, using MultinomialNB, I add the feature token counts to the training data. Using the default tokenizer, combining the token counts with other three features, I get the means of 0.86907 for accuracy, 0.52712 for F1 score, and 0.68816 for AUC score. The standard deviations are 0.00118, 0.00382, and 0.00202 respectively. Comparing to the Gaussian model with three features, though this model performs less steadily, it does show a good small performance boost from the Gaussian model with 3 features, especially in F1 score which increases from only about 0.37 to nearly 0.53. This is the best Naïve Bayes model I can train, so I draw the ROC curve of it to see its performance in more detail using the same data as drawing the last ROC curve to train and test the model in figure 15.

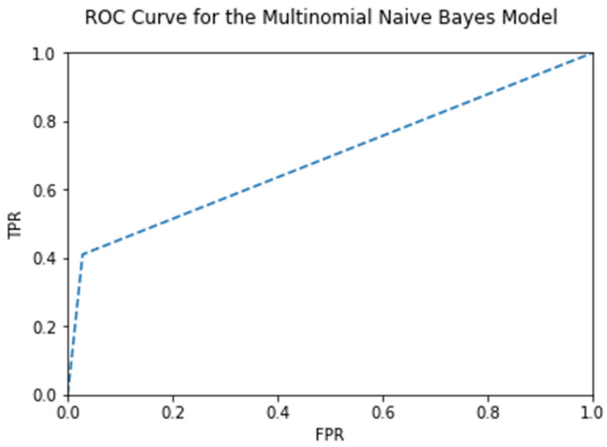

Figure 16. ROC Curve for the Multinomial Naïve Bayes Model

In this graph, the lower segment of the ROC curve is closer to the left side of the plot than that of the last ROC curve, and this shows that the AUC score of this model is larger than that of the model in the last ROC curve graph. In this graph, the AUC score is 0.69100.

Next, I use the Logistic Regression as the algorithms of the models. Firstly, I use token counts only to train the model. While training the Logistic Regression model, I find that training procedure is quite slow, so I only repeat random training for 50 times. Using the default tokenizer first, I test and get the mean accuracy of 0.96370, mean F1 score of 0.89091, and mean AUC score of 0.90831. The standard deviations are 0.00058, 0.00210, and 0.00172 respectively. We can see that the performance is quite decent already. Not only the accuracy, but also the F1 score and AUC score, are quite high. Then I use the specially-made tokenizer to make the token list and train the model. However, surprisingly, the mean scores are 0.96922 for accuracy, 0.76052 for F1 score, and 0.81056 for the AUC score. The standard deviations are 0.00054, 0.00366, and 0.00236 respectively. Except the accuracy, the other evaluating scores of specially-made tokenizers are lower than those of default tokenizers, and the little increase of the accuracy may be because of the specific URLs in the dataset too. The standard deviations of F1 score and AUC score also increase a bit.

To find out that why the specially-made tokenizers performs worse than the default tokenizers, I check the token lists of the two tokenizers. The tokens made by the default tokenizer are mostly normal. Most tokens are the possible contents between the dots. However, when I check the token list of the specially-made tokenizer, I find that many tokens are started by "b'", which I do not expect to appear and will probably not appear in most URLs. After searching for sources, I find out that the "b" is used to lead bytes' literals in Python 3. [23] However, it will influence the tokens in the token list. This makes me not use the specially-made tokenizer in the following tests.

Since in the two tokenizers, the default one performs better, so I use the tokens of default tokenizer and add three manually-crafted features to it to train the model. Combining all the features, I get the result of mean accuracy 0.92356, mean F1 score 0.75031, and mean AUC score 0.81320. The standard

deviations are 0.01334, 0.05050, and 0.03024 respectively. This also surprises because I think with more useful features, the model could advance its performance, but now it does not. Thinking of the reason, I consider that the numbers in the features may be quite large, like countries whose maximum value can reach 186, and this can influence the weight in the logistic function quite a lot. As a result, I decide to use normalized data of the three features to train the model. After training, I get the mean accuracy of 0.96341, mean F1 score of 0.89024, and mean AUC score of 0.90842, and the standard deviations are 0.00064, 0.00193, and 0.00155 respectively. It does perform better than the model without normalizing the three features, but what makes me confused is that it still scores a little bit less than the model with the feature of tokens only. The standard deviations of the model with normalizing the three features decrease quite a lot from those of the model without normalizing. I also try to train the model with token counts, normalized number of levels, and normalized number of special characters, because I think that many URLs in the dataset do not have country code top-level domain, and if just giving them the label of "others", it will make the performance of the model quite bad since both malicious and benign URLs can have no country code top-level domain. That means the feature country will instead have negative impact on its prediction. After training and test, I get the mean accuracy of 0.96394, mean F1 score of 0.89151, and mean AUC score of 0.90899, and standard deviations of 0.00064, 0.00192, and 0.00157 respectively. Though the unsteadiness increases a bit, the mean evaluating scores also increase a little, so the performance really improves. With these useful features, this model is good enough, so I draw a ROC curve of it with the same data as before to train and test the model in figure 17.

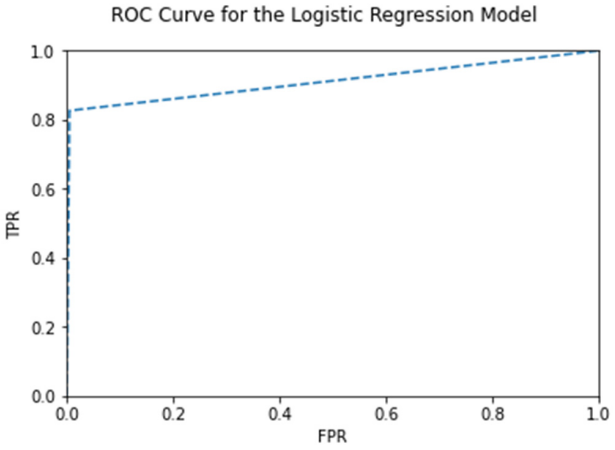

Figure 17. ROC Curve for the Logistic Regression Model

With this graph, we can see that the trapezoid-like shape formed by the ROC curve is quite large, and the performance of it is not bad as well. In this case, the AUC score is 0.91073.

Finding a good combination of features, which is token counts using default tokenizers, normalized number of levels, and normalized number of special characters, I will add hyper parameter tuning to advance the performance. First, I will tune the max_iter. By default, the max_iter is 100 according to the official website of sklearn. What is worthwhile to mention is that before tuning max_iter, when I train the Logistic Regression model with any combination of features, I always see a warning that total number of iterations reached limit, so I attempt to find a number of max_iter which will lead to the disappearance of the warning, and that is 200. With these two numbers, I try to tune max_iter from 100 to 200 with step length of 5, and use cross validation to select the best number of max_iter. After training, I find that the optimum number for max_iter is 110 using AUC score to sort the performance. Using the test dataset to test its performance, it gets 0.96400 for mean accuracy, 0.89172 for mean F1 score, and 0.90924 for mean AUC score. Standard deviations are 0.00064 for accuracy, 0.00187 for F1 score, and 0.00148 for AUC score. Though the improvement is quite small, with hyper parameter tuning, there is performance boost and enhancement in performance steadiness.

Next, I attempt to tune class_weight. The model will get all the data into actual training by default. However, the original URL dataset is quite imbalanced, with the scale of the number of malicious and benign URLs nearly 1:4, so I will change the class_weight and see if the modification can help advance the performance. I modify the scale of class_weight of malicious URLs to benign URLs from 8:1 to 1:8 with each step divided by 2. I was expecting that 4:1 where 4 for malicious and 1 for benign will perform the best since it will make the model get similar number of malicious and benign URLs from the dataset. After training, however, I find that 8:1 where 8 for malicious and 1 for benign scores the most in AUC score, which is 0.99114. Moreover, from 1:8 to 8:1, the scores show a increasing pattern, so I add a new scale which is 16:1 where 16 for malicious and 1 for benign to see whether the score will still increase, but I find that the AUC score of 16:1 is 0.98721, so it does show that 8:1 is the best scale for class_weight. Finding the best value for class_weight, I still repeat 50 times to test its actual performance. Finally, I get mean of 0.97013 for accuracy, 0.91093 for F1 score, and 0.92270 for AUC score. The standard deviations are 0.00104, 0.00313, and 0.00224 respectively. Though the unsteadiness seems larger, the scores do have a decent increase. As a result, with hyper parameter tuning, the performance is really enhanced. With this well-performed model, I also draw a ROC curve using the same data as before to train and test the model of it in figure 18.

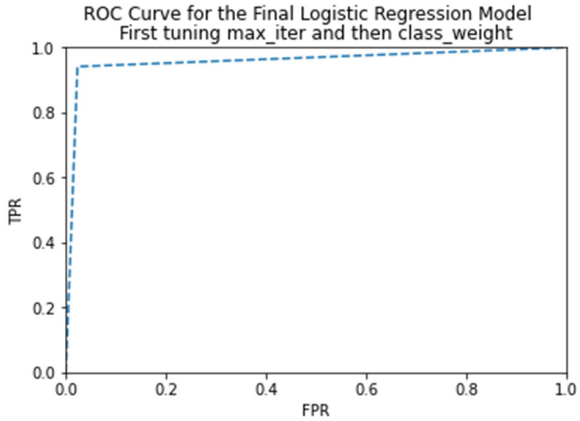

Figure 18. ROC Curve for the Final Logistic Regression Model

In this figure, we can see that the whole curve is closer to the up and left side of the plot than the last curve is, which means the AUC of it is higher than that of the last model. While drawing this curve, the AUC score is 0.95905.

While tuning, I am also confused about another thing: whether the order of hyper parameter tuning will influence the optimum value of the two parameters and the performance of the final model, so I give this a try. This time, I first tune the class_weight, still from 8:1 to 1:8 with each step divided by 2. Nevertheless, the optimum value is the same as that without the order switched. The optimum class_weight is still 8:1 with AUC score 0.99111. Even if I check the 16:1 for another time, it still shows the same pattern, and the AUC score for 16:1 is 0.98532. With class_weight tuned only, I test the performance, and get the mean evaluating scores of 0.97013 for accuracy, 0.91093 for F1 score, and 0.92270 for AUC score. The standard deviations are 0.00104, 0.00313, and 0.00224 respectively. Surprisingly, the scores of tuning class_weight first is the same as those of first tuning max_iter and then tuning class_weight. Even the standard deviations are the same. Then I tune the max_iter. However, unlike I tune the max_iter first, this time with class_weight tuned first, I find that the value of max_iter which will lead to the maximum AUC score is 125, and the corresponding AUC score is 0.99141. Testing the performance of this model, I get mean scores of 0.96909 for accuracy, 0.91591 for F1 score, and 0.95593 for AUC score, and standard deviations of 0.00071, 0.00194, and 0.00108 respectively. Compared to the model without order switched, though the model with switched order performs a bit worse, it outperforms the model without switched order in F1 score and AUC score. The AUC score even increases 0.04. The standard deviations also decrease a lot, with all decreasing nearly half. The drop of accuracy can be seen as the problem of specific data. Having the best model which I have trained, I draw another ROC curve with training and testing using the same data as before to show its performance in figure 19.

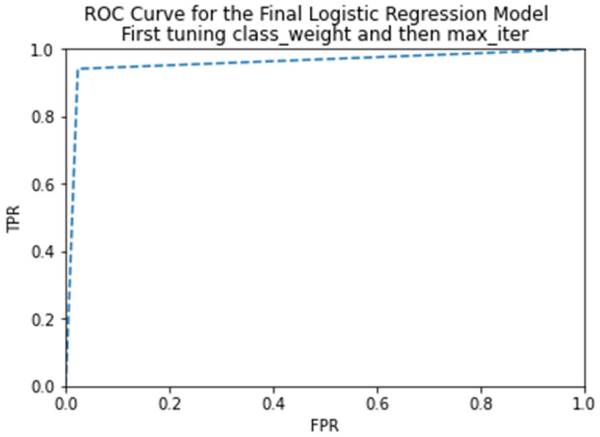

Figure 19. ROC Curve for the Final Logistic Regression Model 2

Only seeing the graph, it is too hard for me to tell the difference from the last graph. However, the AUC score of this is 0.95914, which is a bit larger than that of the last one. As a result, this model is the ultimate model which I have trained.

One point I notice is that no matter which order I use to tune the hyper parameter; the performance is not better when the max_iter is large enough to make the warning disappear. Considering this phenomenon, I find that it is because of model overfitting. [24] Though the larger the max_iter, the more it will fit the training data, it may not fit the test data so well. And since I use test data to tell their performance, it is not odd that with large enough max_iter, the model performs worse instead.

Ultimately, I make a table of my results to show that how well each model performs and how the performance of each model compares to those of other models in table 2.

TABLE II. RESULTS OF PERFORMANCE OF EACH MODEL

Models			Accuracy Information	F1 Score Information	AUC Score Information
Naïve Bayes	GaussianNB	Levels of URL only	Mean: 0.81461 Standard Deviation: 0.00129	Mean: 0.14538 Standard Deviation: 0.00362	Mean: 0.53089 Standard Deviation: 0.00124
		Levels of URL and Countries	Mean: 0.83374 Standard Deviation: 0.00115	Mean: 0.41828 Standard Deviation: 0.00364	Mean: 0.63799 Standard Deviation: 0.00182
		Levels of URL, Countries, and Number of Special Characters	Mean: 0.82775 Standard Deviation: 0.00120	Mean: 0.43367 Standard Deviation: 0.00341	Mean: 0.64773 Standard Deviation: 0.00185
	MultinomialNB	Levels of URL, Countries, Number of Special Characters, and Token Counts	Mean: 0.86907 Standard Deviation: 0.00118	Mean: 0.52712 Standard Deviation: 0.00382	Mean: 0.68816 Standard Deviation: 0.00202
Logistic Regression	Without Hyperparameter Tuning	Token Counts only (Default Tokenizer)	Mean: 0.96370 Standard Deviation: 0.00058	Mean: 0.89091 Standard Deviation: 0.00210	Mean: 0.90831 Standard Deviation: 0.00172
		Token Counts only (Specially-made Tokenizer)	Mean: 0.96922 Standard Deviation: 0.00054	Mean: 0.76052 Standard Deviation: 0.00366	Mean: 0.81056 Standard Deviation: 0.00236
		Levels of URL, Countries, Number of Special Characters, and Tokens	Mean: 0.92356 Standard Deviation: 0.01344	Mean: 0.75031 Standard Deviation: 0.05050	Mean: 0.81320 Standard Deviation: 0.03024

		Levels of URL, Countries, Number of Special Characters, and Tokens (First 3 features normalized)	Mean: 0.96341 Standard Deviation: 0.00064	Mean: 0.89024 Standard Deviation: 0.00193	Mean: 0.90842 Standard Deviation: 0.00155
		Levels of URL, Number of Special Characters, and Tokens (First 2 features normalized)	Mean: 0.96394 Standard Deviation: 0.00064	Mean: 0.89151 Standard Deviation: 0.00192	Mean: 0.90899 Standard Deviation: 0.00157
	Hyperparameter Tuning	Levels of URL, Number of Special Characters, and Tokens (First 2 features normalized, Tuning max_iter)	Mean: 0.96400 Standard Deviation: 0.00064	Mean: 0.89172 Standard Deviation: 0.00187	Mean: 0.90924 Standard Deviation: 0.00148
		Levels of URL, Number of Special Characters, and Tokens (First 2 features normalized, First tuning max_iter, and then class_weight)	Mean: 0.97013 Standard Deviation: 0.00104	Mean: 0.91093 Standard Deviation: 0.00313	Mean: 0.92270 Standard Deviation: 0.00224
		Levels of URL, Number of Special Characters, and Tokens (First 2 features normalized, Tuning class_weight)	Mean: 0.97013 Standard Deviation: 0.00104	Mean: 0.91093 Standard Deviation: 0.00313	Mean: 0.92270 Standard Deviation: 0.00224
		Levels of URL, Number of Special Characters, and Tokens (First 2 features normalized, First tuning class_weight, and then max_iter)	Mean: 0.96909 Standard Deviation: 0.00071	Mean: 0.91591 Standard Deviation: 0.00194	Mean: 0.95593 Standard Deviation: 0.00108

VI. Application

With this accurate model to help us predict malicious URLs, I would like to design an Application Program Interface (API) to help Internet users easily use this model to help them prevent entering malicious URLs.

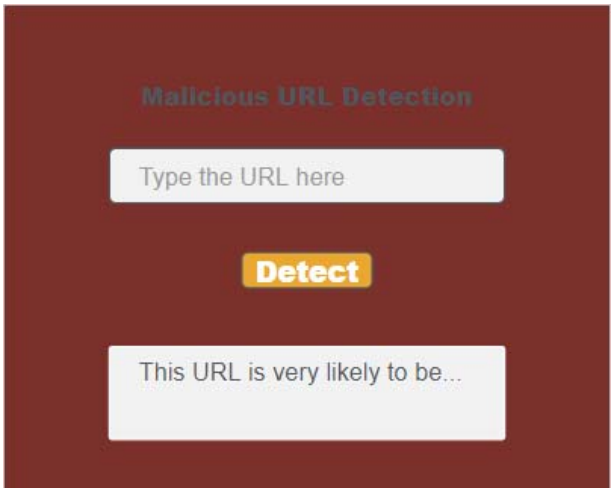

Figure 20. URL Detection Program Interface

Figure 18 shows a possible interface of the program. The interface is quite simple, but with all the things needed to be a malicious URL detection tool.

We can also design it as an extension set in the browsers like Chrome. Since people usually use browsers to surf the Internet and enter these URLs, with the help of a built-in extension in the browser, when people click to a new URL, the extension will just predict this URL automatically instead of users manually input the URL text into a program.

However, the API and browser extension are only imaginations. If I put them into practice, it will also take me a lot of time. At present, I only have those models.

VII. Conclusion

During the outbreak of COVID-19, hackers take advantage of malicious URLs to attack Internet users and steal their information with increasing Internet traffic. To detect them, I use machine learning to build a prediction model with good performance. After finding the proper features, I use different combinations of features to train and test the model and tune hyperparameter. The accuracy of the model is 97% and F1 score is close to 0.92. The model can be applied as programs or browser extensions to help people detect those malicious URLs. I hope that using this model, people can get away from all the malicious URLs and keep their personal information in safety.

Acknowledgment

Thanks to Mr. Hong Haibo for recommending the topic of the project and instructing me to write the Introduction and Results parts of the report for free. He also inspires me to come up with the possible uses of this project, which becomes the Application part of the report.

References

[1] COVID-19 Pushes Up Internet Use 70% And Streaming More Than 12%, First Figures Reveal, Mark Beech
https://www.forbes.com/sites/markbeech/2020/03/25/covid-19-pushes-up-internet-use-70-streaming-more-than-12-first-figures-reveal/#43ad6e113104

[2] Malicious URL Detection using Machine Learning: A Survey, Dotyen Sahoo, Chenghao Liu, Steven C.H. Hoi
https://arxiv.org/pdf/1701.07179.pdf

[3] Okunoye, O & Azeez, Nureni & Ilurimi, Funmilayo. (2017). A WEB ENABLED ANTI-PHISHING SOLUTION USING ENHANCED HEURISTIC BASED TECHNIQUE FUTA Journal of Research in Sciences. 304-321.

[4] What is machine learning? Karen Hao
https://www.technologyreview.com/2018/11/17/103781/what-is-machine-learning-we-drew-you-another-flowchart/

[5] Naive Bayes Classifier, Rohith Gandhi
https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c

[6] Logistic Regression — Detailed Overview, Saishruthi Swaminathan

https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc

[7] What is TF-IDF?
https://monkeylearn.com/blog/what-is-tf-idf/

[8] sklearn.feature_extraction.text.TfidfVectorizer
https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

[9] How to Encode Text Data for Machine Learning with scikit-learn, Jason Brownlee
https://machinelearningmastery.com/prepare-text-data-machine-learning-scikit-learn/

[10] Estimating Continuous Distributions in Bayesian Classifiers, George H. John, Pat Langley
https://dl.acm.org/doi/epdf/10.5555/2074158.2074196

[11] Multinomial Distribution: Definition, Examples
https://www.statisticshowto.com/multinomial-distribution/

[12] Perme, Maja & Blas, Mateja & Turk, Sandra. (2004). Comparison of Logistic Regression and Linear Discriminant Analysis: A Simulation Study. Metodološki Zvezki. 1. 143-161.

[13] sklearn.linear_model.LogisticRegression
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[14] The truth of the F-measure, Yutaka Sasaki
https://www.toyota-ti.ac.jp/Lab/Denshi/COIN/people/yutaka.sasaki/F-measure-YS-26Oct07.pdf

[15] Tharwat, A. (2020), "Classification assessment methods", *Applied Computing and Informatics*, Vol. ahead-of-print No. ahead-of print.
https://doi.org/10.1016/j.aci.2018.08.003

[16] Understanding AUC - ROC Curve, Sarang Narkhede
https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

[17] Receiver Operating Characteristic (ROC) Curve Analysis for Medical Diagnostic Test Evaluation, Karimollah Hajian-Tilaki
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3755824/

[18] How to Use ROC Curves and Precision-Recall Curves for Classification in Python, Jason Brownlee
https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/

[19] ROC curve analysis
https://www.medcalc.org/manual/roc-curves.php#:~:text=specificity%20will%20decrease.-,The%20ROC%20curve,to%20a%20particular%20decision%20threshold.

[20] Why and how to Cross Validate a Model? Sanjay. M
https://towardsdatascience.com/why-and-how-to-cross-validate-a-model-d6424b45261f

[21] In Depth: Naive Bayes Classification
https://jakevdp.github.io/PythonDataScienceHandbook/05.05-naive-bayes.html

[22] GaussianNB cannot accept sparse matrix input. Answered by Jake Vanderplas
https://github.com/scikit-learn/scikit-learn/issues/6440

[23] What does the 'b' character do in front of a string literal in Python? Rajendra Dharmkar
https://www.tutorialspoint.com/What-does-the-b-character-do-in-front-of-a-string-literal-in-Python#:~:text=In%20Python%203%2C%20Bytes%20literals,must%20be%20expressed%20with%20escapes.

[24] Overfitting in Machine Learning: What It Is and How to Prevent It
https://elitedatascience.com/overfitting-in-machine-learning