Identifying Malicious Websites Using Python Machine Learning

Introduction

The Python code used to identify dangerous phishing websites is explained in this article. The algorithm uses a variety of characteristics from the URL and website structure to classify websites as "malware," "phishing," "benign," or "defacement."

Statement

The case study you mention addresses the topic of multi-class classification in the context of malicious URL detection. The URLs are divided into four groups: defacement, malware, phishing, and benign (or safe). In cybersecurity, this categorization is essential since it protects users from hazardous information and helps identify possible dangers.

Work Flow

Three distinct machine learning models—Random Forest, XGBoost, and LightGBM—are used by the Python code in the case study to carry out this categorization. A dataset of URLs is used to train and test each model, and the accuracy of each model is computed and reported. A function to guess the class of a given URL is also included in the code. This function pulls various information from the URL, including the length of the URL, the existence of suspicious terms, the presence of particular characters, and the number of directories. The LightGBM model then uses these attributes as input to forecast the URL's class.

Dataset

We are going to use a dataset of 6,51,191 malicious URLs, of which 32,520 are malware URLs, 94,111 are phishing URLs, 4,28,103 are benign or safe URLs, and 96,457 are defacement URLs. Let's now talk about the various kinds of URLs that are included in our dataset: benign, malicious, phishing, and defacement URLs.

Benign URLs: You can safely access these URLs.

Malware URLs: When a victim visits one of these URLs, malware is injected into their system.

Defacement URLs: Using methods like code injection and cross-site scripting, hackers typically construct defacement URLs intending to infiltrate a web server and replace the hosted website with their own. Websites belonging to corporations, banks, governments, and religious organizations are frequently vandalized.

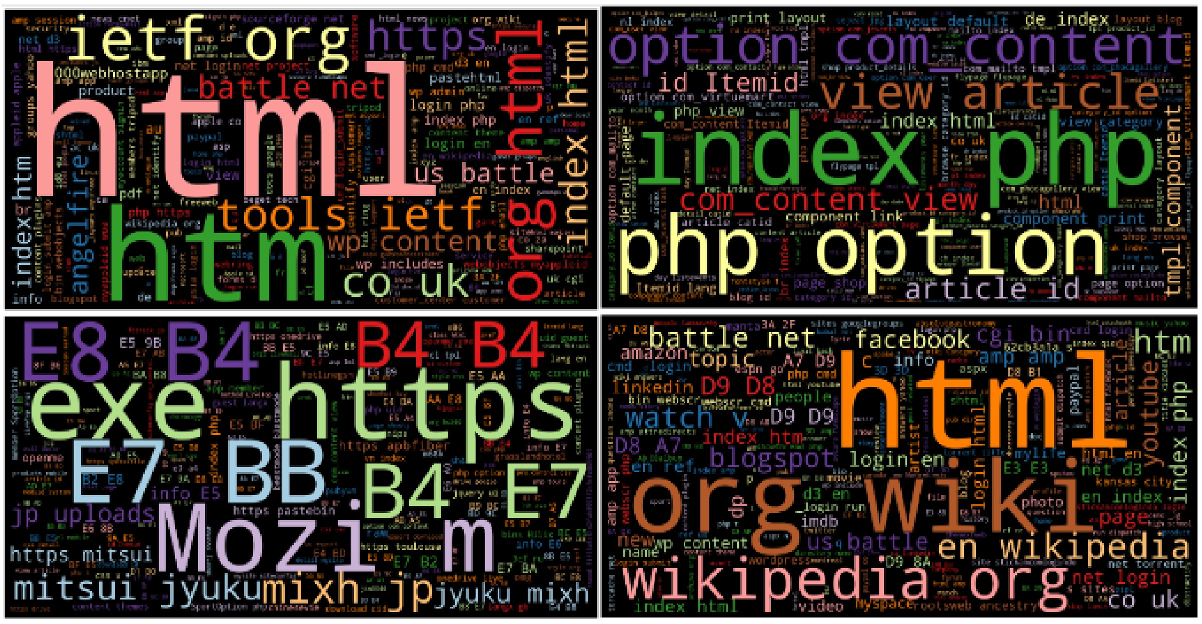
Phishing URLs: Hackers attempt to get sensitive financial or personal data, including credit card numbers, internet banking credentials, and login credentials, by fabricating phishing URLs.

For that We use Follwing URLs:

- mp3raid.com/music/krizz_kaliko.html
- infinitysw.com
- google.co.in
- myspace.com
- proplast.co.nz
- http://103.112.226.142:36308/Mozi.m
- microencapsulation.readmyweather.com
- xo3fhvm5lcvzy92q.download
- http://www.vnic.co/khach-hang.html
- http://www.raci.it/component/user/reset.html
- http://www.approvi.com.br/ck.htm
- http://www.juventudelirica.com.br/index.html
- roverslands.net
- corporacionrossenditotours.com
- http://drive-google-com.fanalav.com/6a7ec96d6a
- citiprepaid-salarysea-at.tk

URLs Word Cloud plotting

Word clouds are widely used in natural language processing to analyze patterns of words or tokens. They provide valuable insights into the distribution of words within a dataset. The word cloud of benign URLs contains common tokens like html, com, org, and wiki, while phishing URLs have tokens like tools, ietf, index, and html. Malware URLs contain tokens like exe, E7, BB, and MOZI. Defacement URLs have development terms like index, php, itemid, https, and option. Overall, the word cloud is an indispensable tool for analyzing patterns in textual data.



Importing Libraries

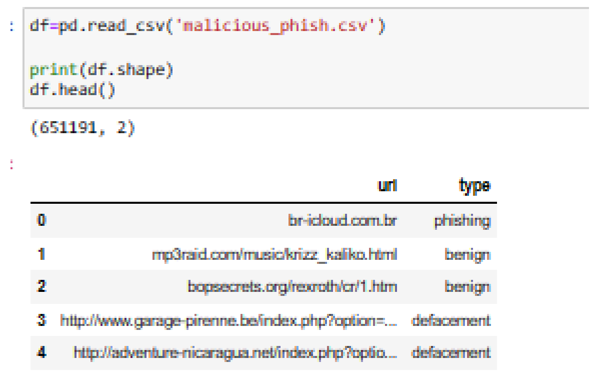
Here are the necessary Python libraries imported for this project.

```python
import pandas as pd
import itertools
from sklearn.metrics import classification_report,confusion_matrix, accuracy_score
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import xgboost as xgb
from lightgbm import LGBMClassifier
import os
import seaborn as sns
from wordcloud import WordCloud
```

the next step would be loading the dataset and checking some sample records to get a better understanding of the data.

Dataset

We'll import the dataset using pandas and examine some sample entries. The dataset has 651,191 entries with raw URLs and target variables. We'll then move on to feature engineering to create lexical features from the URLs.



```
df=pd.read_csv('malicious_phish.csv')

print(df.shape)
df.head()
```

```
(651191, 2)
```

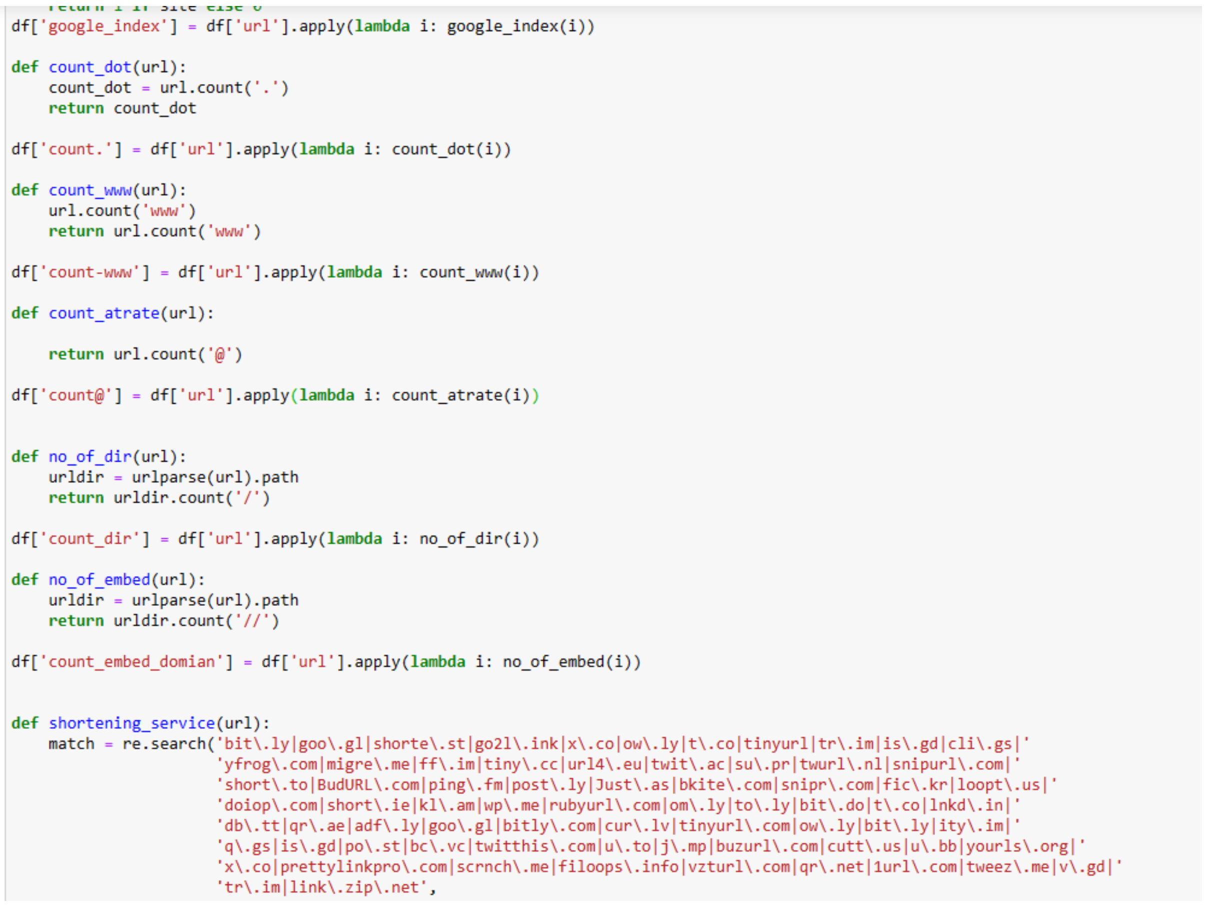
	url	type
0	br-icloud.com.br	phishing
1	mp3raid.com/music/krizz_kaliko.html	benign
2	bopsecrets.org/rexroth/cr/1.htm	benign
3	http://www.garage-pirenne.be/index.php?option=...	defacement
4	http://adventure-nicaragua.net/index.php?optio...	defacement

Feature engineering

Below are the features used for detecting malicious URLs:

- **having_ip_address:** checks whether the URL has an IP address or not.
- **abnormal_url:** checks whether the identity is part of the URL for a legitimate website.
- **google_index:** checks whether the URL is indexed in the Google search console.
- **Count . :** checks whether the URL contains more than three dots(.), indicating a malicious site.
- **Count-www:** checks whether the URL has no or more than one www in its URL.
- **count@:** ignores everything prior to the "@" symbol in the URL.
- **Count_dir:** checks whether the URL contains multiple directories, which generally indicates a suspicious website.
- **Count_embed_domain:** checks the number of embedded domains in the URL.
- **Suspicious words in URL:** check the presence of frequently occurring suspicious words in the URL as a binary variable.
- **Short_url:** checks whether the URL uses URL shortening services.
- **Count_https:** checks the presence or absence of HTTPS protocol in the URL.
- **Count_http**: checks whether the URL has more than one HTTP.
- **Count%:** checks the number of % symbols in the URL.
- **Count?:** checks the presence of the symbol "?" in the URL.
- **Count-:** checks the presence of dashes in the prefix or suffix of the brand name.
- **Count=:** checks the presence of equal to (=) in the URL.
- **url_length:** checks the length of the URL.
- **hostname_length:** checks the length of the hostname in the URL.
- **First directory length:** determines the length of the first directory in the URL.

- **Length of top-level domains:** checks the length of the top-level domain (TLD) in the URL.
- **Count_digits:** checks the presence of digits in the URL.
- **Count_letters:** checks the number of letters in the URL.



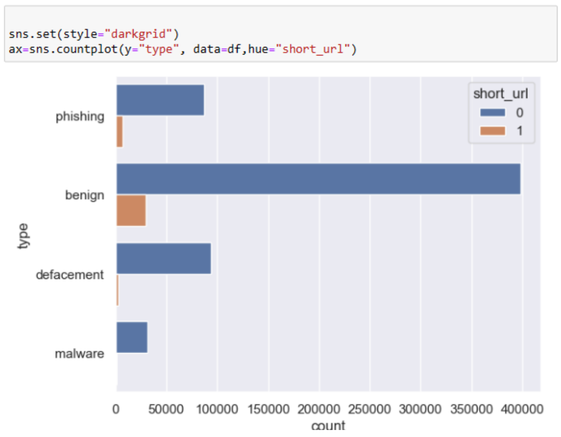
```
    return 1 if site else 0
df['google_index'] = df['url'].apply(lambda i: google_index(i))

def count_dot(url):
    count_dot = url.count('.')
    return count_dot

df['count.'] = df['url'].apply(lambda i: count_dot(i))

def count_www(url):
    url.count('www')
    return url.count('www')

df['count-www'] = df['url'].apply(lambda i: count_www(i))

def count_atrate(url):

    return url.count('@')

df['count@'] = df['url'].apply(lambda i: count_atrate(i))


def no_of_dir(url):
    urldir = urlparse(url).path
    return urldir.count('/')

df['count_dir'] = df['url'].apply(lambda i: no_of_dir(i))

def no_of_embed(url):
    urldir = urlparse(url).path
    return urldir.count('//')

df['count_embed_domian'] = df['url'].apply(lambda i: no_of_embed(i))


def shortening_service(url):
    match = re.search('bit\.ly|goo\.gl|shorte\.st|go2l\.ink|x\.co|ow\.ly|t\.co|tinyurl|tr\.im|is\.gd|cli\.gs|'
                      'yfrog\.com|migre\.me|ff\.im|tiny\.cc|url4\.eu|twit\.ac|su\.pr|twurl\.nl|snipurl\.com|'
                      'short\.to|BudURL\.com|ping\.fm|post\.ly|Just\.as|bkite\.com|snipr\.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\.ly|bit\.ly|ity\.im|'
                      'q\.gs|is\.gd|po\.st|bc\.vc|twitthis\.com|u\.to|j\.mp|buzurl\.com|cutt\.us|u\.bb|yourls\.org|'
                      'x\.co|prettylinkpro\.com|scrnch\.me|filoops\.info|vzturl\.com|qr\.net|1url\.com|tweez\.me|v\.gd|'
                      'tr\.im|link\.zip\.net',
```

```python
df['short_url'] = df['url'].apply(lambda i: shortening_service(i))

def count_https(url):
    return url.count('https')

df['count-https'] = df['url'].apply(lambda i : count_https(i))

def count_http(url):
    return url.count('http')

df['count-http'] = df['url'].apply(lambda i : count_http(i))

def count_per(url):
    return url.count('%')

df['count%'] = df['url'].apply(lambda i : count_per(i))

def count_ques(url):
    return url.count('?')

df['count?'] = df['url'].apply(lambda i: count_ques(i))

def count_hyphen(url):
    return url.count('-')

df['count-'] = df['url'].apply(lambda i: count_hyphen(i))

def count_equal(url):
    return url.count('=')

df['count='] = df['url'].apply(lambda i: count_equal(i))

def url_length(url):
    return len(str(url))
```

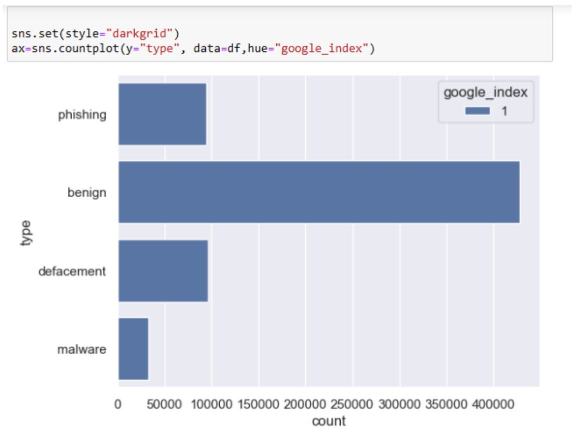
```python
def hostname_length(url):
    return len(urlparse(url).netloc)

df['hostname_length'] = df['url'].apply(lambda i: hostname_length(i))

df.head()

def suspicious_words(url):
    match = re.search('PayPal|login|signin|bank|account|update|free|lucky|service|bonus|ebayisapi|webscr',
                      url)
    if match:
        return 1
    else:
        return 0
df['sus_url'] = df['url'].apply(lambda i: suspicious_words(i))


def digit_count(url):
    digits = 0
    for i in url:
        if i.isnumeric():
            digits = digits + 1
    return digits


df['count-digits']= df['url'].apply(lambda i: digit_count(i))


def letter_count(url):
    letters = 0
    for i in url:
        if i.isalpha():
            letters = letters + 1
    return letters
```

above 22 features, the dataset looks like the below.

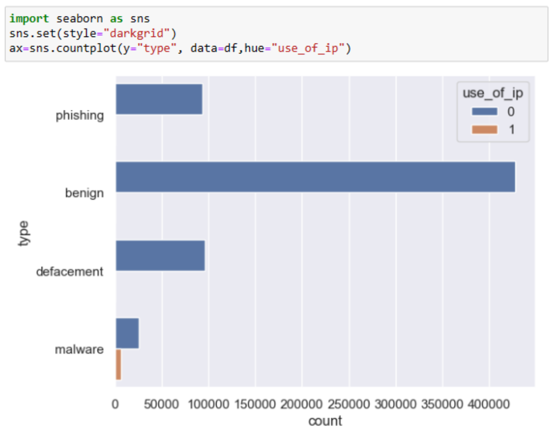
	use_of_ip	abnormal_url	count.	count-www	count@	count_dir	count_embed_domian	short_url	count-https	count-http	...	count?	count-	count=	url_length
0	0	0	2	0	0	0	0	0	0	0	...	0	1	0	16
1	0	0	2	0	0	2	0	0	0	0	...	0	0	0	35
2	0	0	2	0	0	3	0	0	0	0	...	0	0	0	31
3	0	1	3	1	0	1	0	0	0	1	...	1	1	4	88
4	0	1	2	0	0	1	0	0	0	1	...	1	1	3	235



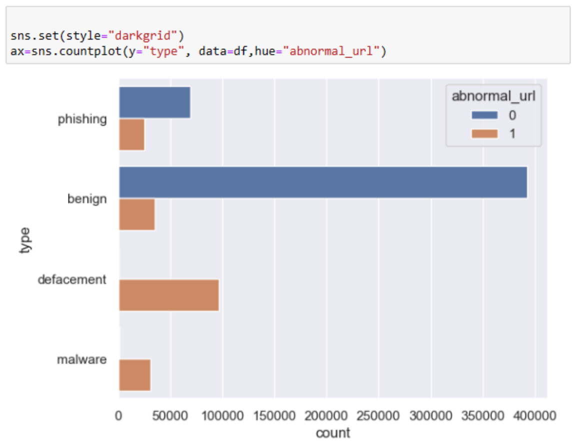
```
sns.set(style="darkgrid")
ax=sns.countplot(y="type", data=df,hue="short_url")
```



```
sns.set(style="darkgrid")
ax=sns.countplot(y="type", data=df,hue="google_index")
```



```
import seaborn as sns
sns.set(style="darkgrid")
ax=sns.countplot(y="type", data=df,hue="use_of_ip")
```



```
sns.set(style="darkgrid")
ax=sns.countplot(y="type", data=df,hue="abnormal_url")
```

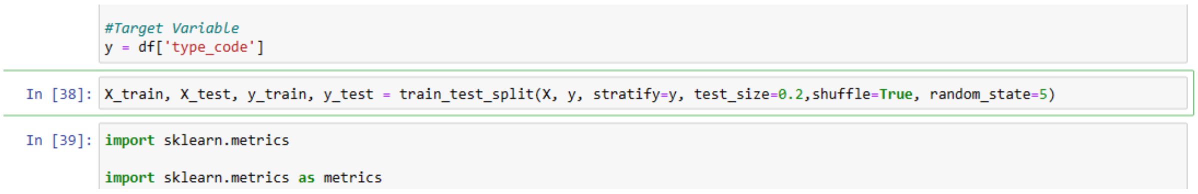
Based on the data analysis, we found that URLs with IP addresses are strong indicators of malware, as they are the only ones with this feature. We also observed that defacement URLs have a higher distribution of the abnormal URL feature, which is useful in identifying them.

The distribution of the suspicious URLs feature emphasizes the need to identify and flag transaction and payment-related keywords to prevent cyber attacks. We noticed that benign URLs have the highest distribution, followed by phishing URLs, suggesting that careful analysis and classification are necessary.

Moreover, our analysis of the short URL feature distribution suggests that benign URLs have the highest number of short URLs. Therefore, it is crucial to closely examine URLs that use URL shortening services. These findings clearly demonstrate the importance of using machine learning models in accurately classifying URLs and safeguarding users against potential cybersecurity threats.

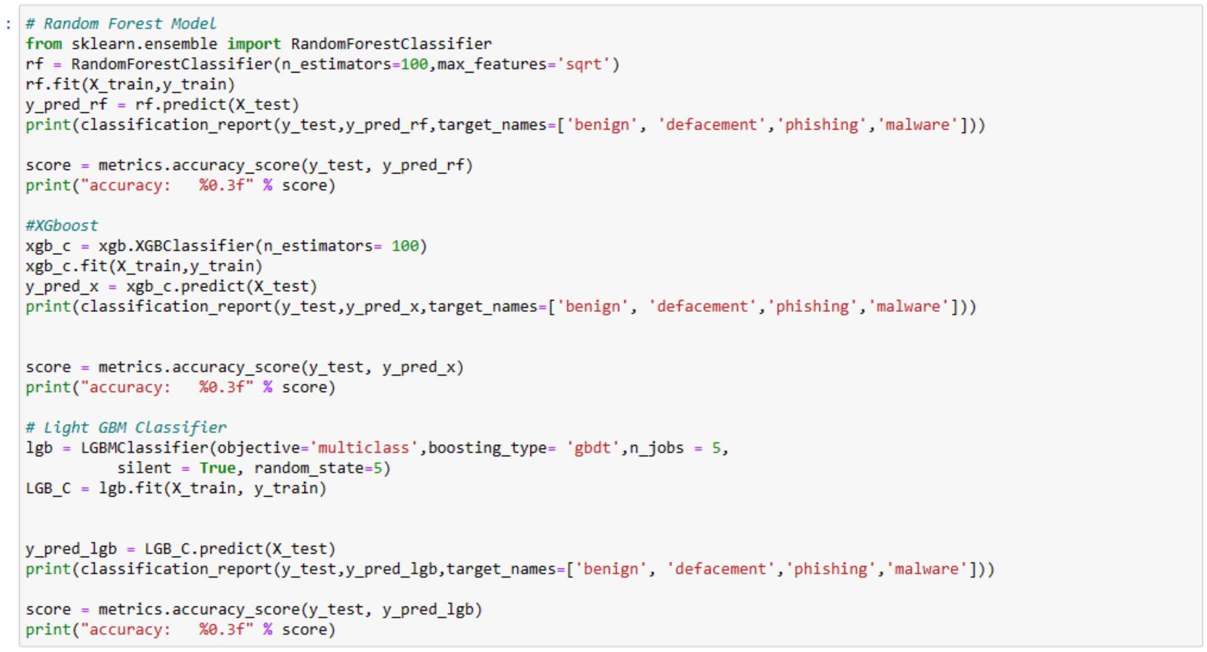
Training & Test Split

In order to ensure that our machine learning models are accurate and reliable, we have split our dataset into a train and test set using an 80:20 ratio. However, as our dataset is imbalanced, with a majority of benign URLs, it is crucial to maintain the same proportion of the target variable in the train and test sets. This not only ensures that the performance of the machine learning model is not impacted but also allows for better predictions on future data. By using the stratify parameter, we can achieve this balance and ensure that the proportion of values in the sample produced is the same as the proportion of values provided to the parameter stratify.



```
        #Target Variable
        y = df['type_code']

In [38]: X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, test_size=0.2,shuffle=True, random_state=5)

In [39]: import sklearn.metrics

         import sklearn.metrics as metrics
```

Model building



```
: # Random Forest Model
  from sklearn.ensemble import RandomForestClassifier
  rf = RandomForestClassifier(n_estimators=100,max_features='sqrt')
  rf.fit(X_train,y_train)
  y_pred_rf = rf.predict(X_test)
  print(classification_report(y_test,y_pred_rf,target_names=['benign', 'defacement','phishing','malware']))

  score = metrics.accuracy_score(y_test, y_pred_rf)
  print("accuracy:   %0.3f" % score)

  #XGboost
  xgb_c = xgb.XGBClassifier(n_estimators= 100)
  xgb_c.fit(X_train,y_train)
  y_pred_x = xgb_c.predict(X_test)
  print(classification_report(y_test,y_pred_x,target_names=['benign', 'defacement','phishing','malware']))


  score = metrics.accuracy_score(y_test, y_pred_x)
  print("accuracy:   %0.3f" % score)

  # Light GBM Classifier
  lgb = LGBMClassifier(objective='multiclass',boosting_type= 'gbdt',n_jobs = 5,
            silent = True, random_state=5)
  LGB_C = lgb.fit(X_train, y_train)


  y_pred_lgb = LGB_C.predict(X_test)
  print(classification_report(y_test,y_pred_lgb,target_names=['benign', 'defacement','phishing','malware']))

  score = metrics.accuracy_score(y_test, y_pred_lgb)
  print("accuracy:   %0.3f" % score)
```

The code appears to be implementing and evaluating three different machine learning models - Random Forest, XGBoost, and LightGBM - for the purpose of classification. Each model is trained and tested on separate datasets, and their performance is evaluated using classification report and accuracy score.

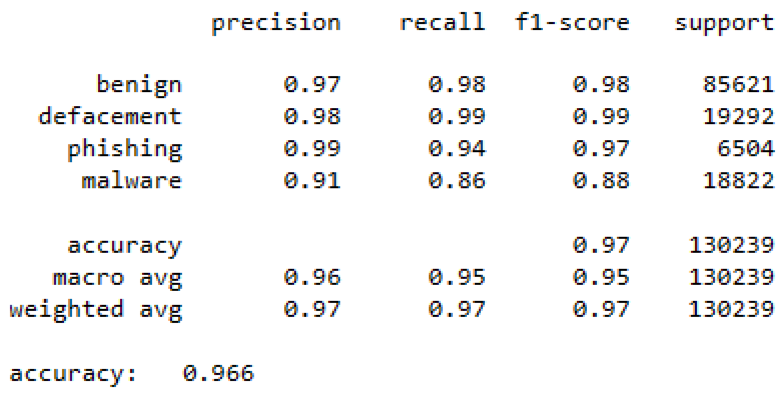
The Random Forest model is initialized with 100 trees and the maximum number of features is set to the square root of the number of features. The model is trained on the training dataset, and predictions are made on the test dataset. The classification report and accuracy score are then printed.

The XGBoost model is initialized with 100 trees, trained on the training dataset, and predictions are made on the test dataset. The classification report and accuracy score are then printed.

The LightGBM model is initialized with specific parameters, trained on the training dataset, and predictions are made on the test dataset. The classification report and accuracy score are then printed.

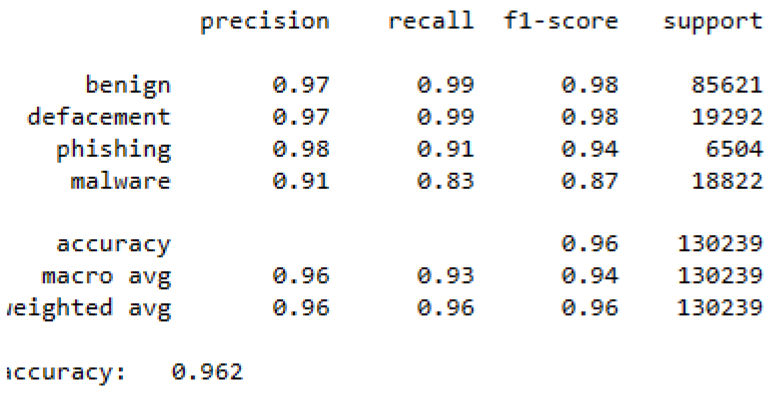
The classification report provides precision, recall, and F1-score for each class, namely 'benign', 'defacement', 'phishing', and 'malware'. The accuracy score represents the overall accuracy of the model.

Random Forest



```
              precision    recall  f1-score   support

      benign       0.97      0.98      0.98     85621
  defacement       0.98      0.99      0.99     19292
    phishing       0.99      0.94      0.97      6504
     malware       0.91      0.86      0.88     18822

    accuracy                           0.97    130239
   macro avg       0.96      0.95      0.95    130239
weighted avg       0.97      0.97      0.97    130239

accuracy:   0.966
```

XG Boost



```
              precision    recall  f1-score   support

      benign       0.97      0.99      0.98     85621
  defacement       0.97      0.99      0.98     19292
    phishing       0.98      0.91      0.94      6504
     malware       0.91      0.83      0.87     18822

    accuracy                           0.96    130239
   macro avg       0.96      0.93      0.94    130239
 eighted avg       0.96      0.96      0.96    130239

ccuracy:   0.962
```

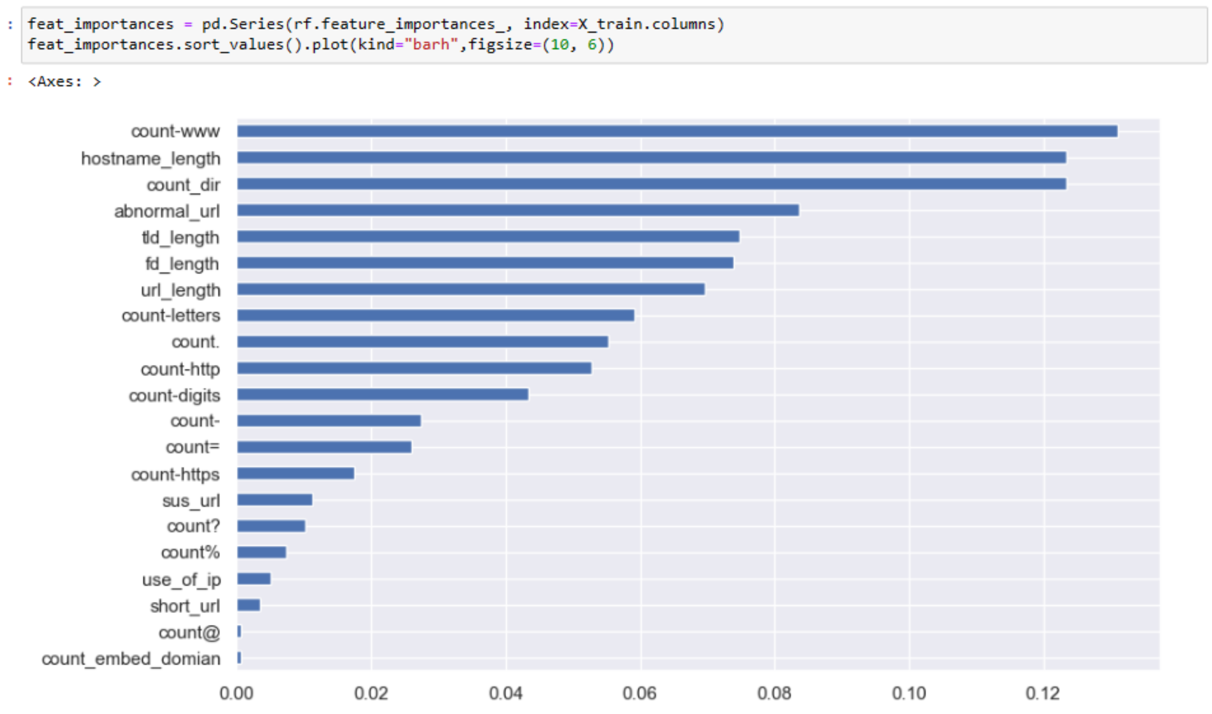
Light GBM

```
              precision    recall  f1-score   support

      benign       0.97      0.99      0.98     85621
  defacement       0.96      0.99      0.98     19292
    phishing       0.97      0.90      0.93      6504
     malware       0.90      0.83      0.86     18822

    accuracy                           0.96    130239
   macro avg       0.95      0.93      0.94    130239
weighted avg       0.96      0.96      0.96    130239

accuracy:   0.959
```

Random Forest shows the best performance in terms of test accuracy as it attains the highest accuracy of 96.6% with a higher detection rate for benign, defacement, phishing, and malware. The Random Forest model shows that the most crucial feature for identifying malicious URLs is the URL length, followed by the presence of the '@' symbol, the number of directories in the URL, and the presence of suspicious words in the URL. Other essential features include the presence of HTTPS, the number of dots in the URL, and the presence of the equal sign. These findings suggest that URL structure plays a vital role in identifying malicious URLs.

Feature Importance

After selecting our model i.e., Random Forest, next, we will be checking highly contributing features. The code for plotting feature importance plot.

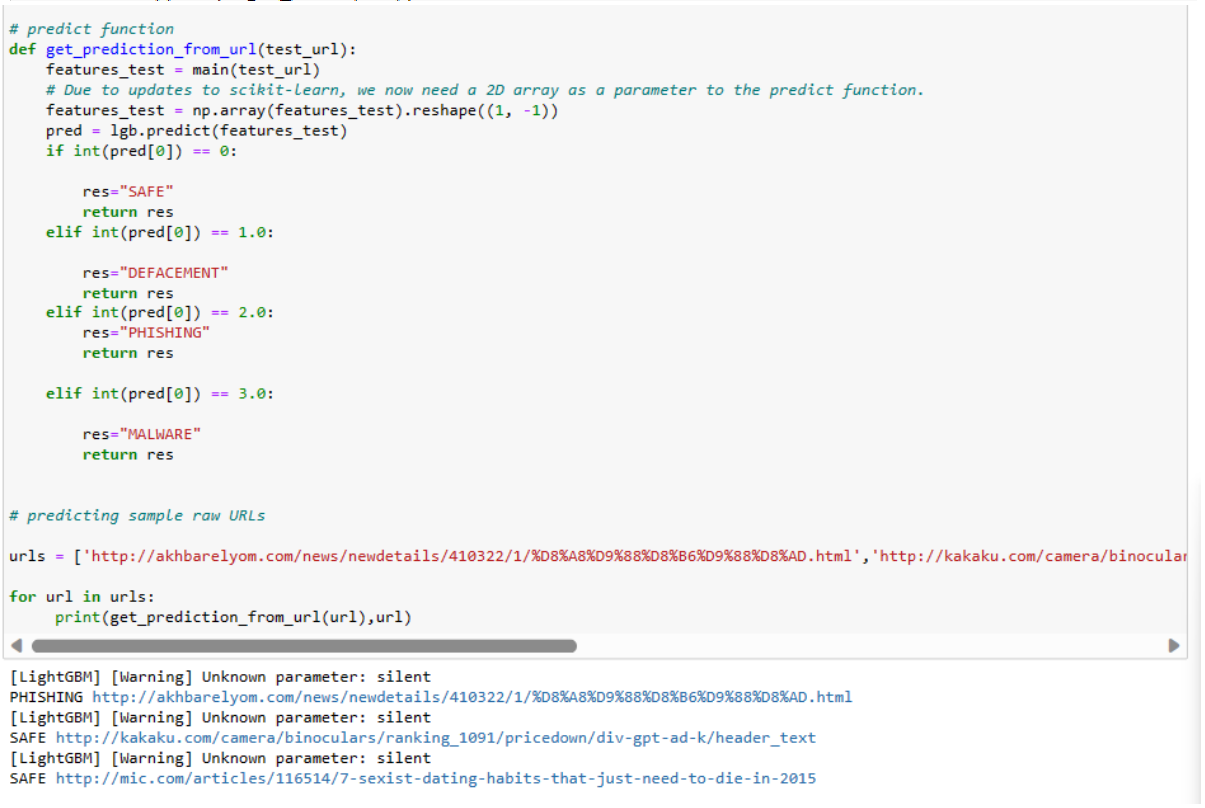


```
: feat_importances = pd.Series(rf.feature_importances_, index=X_train.columns)
  feat_importances.sort_values().plot(kind="barh",figsize=(10, 6))
: <Axes: >
```

From the above plot, we can observe that the top 5 features for detecting malicious URLs are hostname_length, count_dir, count-www, fd_length, and url_length.

Model prediction

```
: def main(url):

      status = []

      status.append(having_ip_address(url))
      status.append(abnormal_url(url))
      status.append(count_dot(url))
      status.append(count_www(url))
      status.append(count_atrate(url))
      status.append(no_of_dir(url))
      status.append(no_of_embed(url))

      status.append(shortening_service(url))
      status.append(count_https(url))
      status.append(count_http(url))

      status.append(count_per(url))
      status.append(count_ques(url))
      status.append(count_hyphen(url))
      status.append(count_equal(url))

      status.append(url_length(url))
      status.append(hostname_length(url))
      status.append(suspicious_words(url))
      status.append(digit_count(url))
```



```
# predict function
def get_prediction_from_url(test_url):
    features_test = main(test_url)
    # Due to updates to scikit-learn, we now need a 2D array as a parameter to the predict function.
    features_test = np.array(features_test).reshape((1, -1))
    pred = lgb.predict(features_test)
    if int(pred[0]) == 0:

        res="SAFE"
        return res
    elif int(pred[0]) == 1.0:

        res="DEFACEMENT"
        return res
    elif int(pred[0]) == 2.0:
        res="PHISHING"
        return res

    elif int(pred[0]) == 3.0:

        res="MALWARE"
        return res


# predicting sample raw URLs

urls = ['http://akhbarelyom.com/news/newdetails/410322/1/%D8%A8%D9%88%D8%B6%D9%88%D8%AD.html','http://kakaku.com/camera/binocular

for url in urls:
     print(get_prediction_from_url(url),url)
```

```
[LightGBM] [Warning] Unknown parameter: silent
PHISHING http://akhbarelyom.com/news/newdetails/410322/1/%D8%A8%D9%88%D8%B6%D9%88%D8%AD.html
[LightGBM] [Warning] Unknown parameter: silent
SAFE http://kakaku.com/camera/binoculars/ranking_1091/pricedown/div-gpt-ad-k/header_text
[LightGBM] [Warning] Unknown parameter: silent
SAFE http://mic.com/articles/116514/7-sexist-dating-habits-that-just-need-to-die-in-2015
```

```
[LightGBM] [Warning] Unknown parameter: silent
PHISHING http://akhbarelyom.com/news/newdetails/410322/1/%D8%A8%D9%88%D8%B6%D9%88%D8%AD.html
[LightGBM] [Warning] Unknown parameter: silent
SAFE http://kakaku.com/camera/binoculars/ranking_1091/pricedown/div-gpt-ad-k/header_text
[LightGBM] [Warning] Unknown parameter: silent
SAFE http://mic.com/articles/116514/7-sexist-dating-habits-that-just-need-to-die-in-2015
```

Conclusion

We've shown how to utilize machine learning to detect malicious URLs. Using raw URLs, we have constructed 22 lexical features and trained three machine-learning models: Random Forest, Light GBM, and XG Boost. In addition, we analyzed the three machine learning models' performances and discovered that Random Forest performed better than the others, achieving the greatest accuracy of 96.6%. The top 5 features for identifying malicious URLs are hostname_length, count_dir, count-www, fd_length, and url_length, according to a plot of Random Forest feature significance. Finally, we have implemented the prediction function that uses our Random Forest stored model to categorize any raw URL.