

# Rapport

## Application CLOUD : Cybersécurité

Projet sur détection d'URL malveillantes à l'aide de  
l'apprentissage automatique



Dans ce projet, nous nous intéressons à l'étude de cas de la détection d'URL malveillantes à l'aide de fonctionnalités lexicales avec une approche d'apprentissage automatique améliorée basée sur une arborescence.

Dans un passé récent, nous avons assisté à une augmentation significative des attaques de cybersécurité telles que les ransomwares, le phishing, l'injection de malwares, etc. sur différents sites Web partout dans le monde. En conséquence, diverses institutions financières, sociétés de commerce électronique et particuliers ont subi d'énormes pertes financières.

Dans un tel type de scénario, une attaque de cybersécurité constitue un défi majeur pour les professionnels de la cybersécurité, car différents types de nouvelles attaques surviennent de jour en jour.

En plus de l'apprentissage automatique réalisée pour détecter les URLs malveillantes, on déploie le projet à l'aide de Docker et FastAPI pour standardiser les opérations de notre projet et de migrer aisément du code en améliorant l'utilisation des ressources.

# Table des matières

## **Chapitre 0 : Préambule**

- Qu'est-ce qu'une URL ?
- Qu'est-ce qu'une URL malveillante ?

## **Chapitre1 : Préparation du modèle de prédiction**

- Énoncé du problème
- Déroulement du projet
- Description du jeu de données
- Word cloud des URLs
- Importation des bibliothèques
- Chargement de base de données
- Ingénierie des caractéristiques
- Analyse exploratoire des données (EDA)
- Encodage des étiquettes
- Séparation des variables caractéristiques et cibles
- Division entraînement/test
- Construction du modèle
- Évaluation du modèle
- Importance des caractéristiques
- Prédiction du modèle

## **Chapitre 2 : déploiement du modèle avec Docker**

- Outils et environnement
- Simulation

## Chapitre 0 : Préambule

### 1- Qu'est-ce qu'une URL ?



L'Uniform Resource Locator (URL) est un format structuré bien défini, une adresse unique permettant d'accéder à des sites Web sur le World Wide Web (WWW).

Généralement, une URL légitime est composée de trois éléments de base :

**Protocole** : Il s'agit essentiellement d'un identifiant qui détermine quel protocole utiliser, par exemple HTTP, HTTPS, etc.

**Nom d'hôte** : Aussi appelé nom de ressource. Il contient l'adresse IP ou le nom de domaine où se trouve la ressource réelle.

**Chemin** : Il spécifie le chemin réel où se trouve la ressource.

Selon la figure, wisdomml.in.edu est le nom de domaine. Le domaine de premier niveau est un autre composant du nom de domaine qui indique la nature du site Web, par exemple commercial (.com), éducatif (.edu), organisationnel (.org), etc.

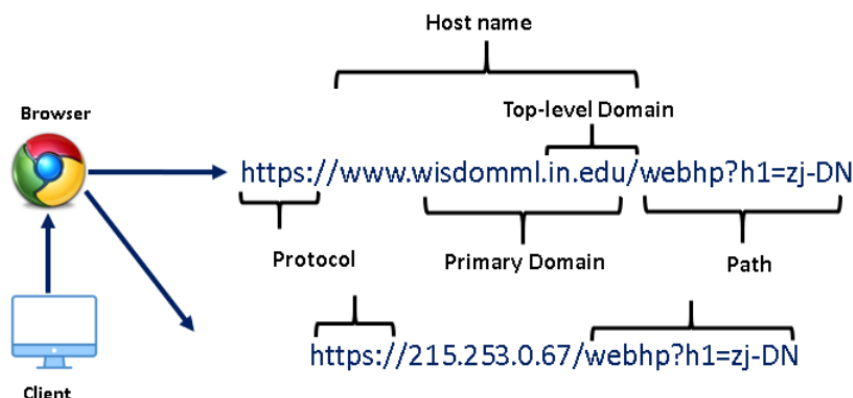
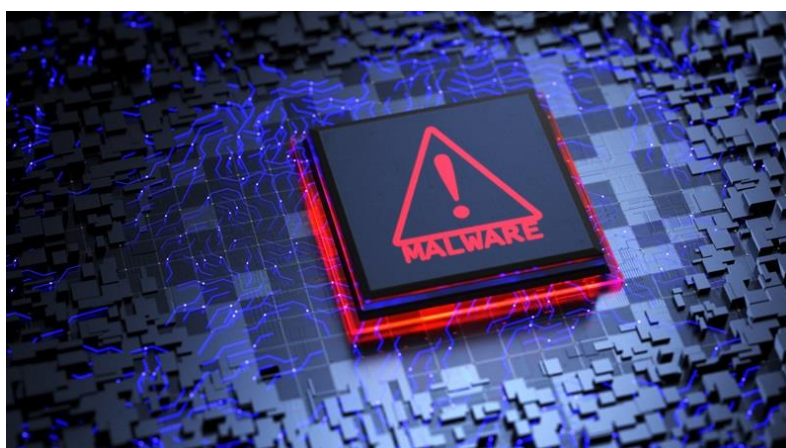


Fig. 1. Components of a URL

## 2- Qu'est-ce qu'une URL malveillante ?



Les URL malveillantes sont des URL modifiées ou compromises utilisées pour des attaques cybernétiques.

Une URL ou un site Web malveillant contient généralement différents types de chevaux de Troie, de logiciels malveillants, de contenu non sollicité sous forme de phishing, de téléchargements indésirables, de spams, etc.

L'objectif principal du site Web malveillant est de frauder ou de voler les informations personnelles ou financières des utilisateurs non méfiants. En raison de la pandémie de COVID-19 en cours, les incidents de cybercriminalité ont considérablement augmenté.

# Chapitre 1 : Préparation du modèle

## 1- Énoncé du problème :

Dans cette étude de cas, nous abordons la détection des URL malveillantes comme un problème de classification multi-classes. Nous classifions les URL brutes en différents types de classes telles que les URLs **benign** ou safe URLs, **phishing** URLs, **malware** URLs, ou **defacement** URLs.

## 2- Déroulement du projet :

Étant donné que les algorithmes d'apprentissage automatique ne prennent en charge que les entrées numériques, nous allons créer des caractéristiques numériques lexicales à partir des URL d'entrée. Ainsi, l'entrée des algorithmes d'apprentissage automatique consistera en les caractéristiques lexicales numériques plutôt que les URL brutes réelles.

Dans cette étude de cas, nous utiliserons le classificateurs d'ensemble d'apprentissage automatique bien connu, à savoir Random Forest.

Par la suite, nous évaluons leur performance et tracerons un graphique de l'importance moyenne des caractéristiques pour comprendre quelles caractéristiques sont importantes dans la prédiction des URL malveillantes.

## 3- Description du Dataset :

Dans cette étude de cas, nous utiliserons un jeu de données d'URLs **malveillantes** comprenant 6,51,191 URLs, parmi lesquelles 4,28,103 sont des URLs **bégnines** ou sûres, 96,457 des URL de **defacement**, 94,111 des URL de **phishing** et 32,520 des URLs de **malware**.

Maintenant, parlons des différents types d'URL dans notre jeu de données, à savoir les URLs **bégnines**, les liens **malveillants**, de **phishing** et de **défiguration**.

**URLs benign:** Ce sont des URLs sécurisées à parcourir.

**URLs de malware :** Ces types d'URLs injectent des liens **malveillants** dans le système de la victime une fois qu'elle visite de telles URLs.

**URLs de defacement :** Les URLs de **défiguration** sont généralement créées par des hackers dans le but de pirater un serveur Web et de remplacer le site Web hébergé par le leur, en utilisant des techniques telles que l'injection de code, les scripts intersites, etc. Les cibles courantes des URLs de **défiguration** sont les sites Web religieux, gouvernementaux, bancaires et d'entreprise.

#### 4- Wordcloud des URLs :

Wordcloud of URLs aide à comprendre le schéma des mots ou des tokens dans des étiquettes cibles particulières.

C'est l'une des techniques les plus attrayantes du **NLP** pour comprendre le schéma de distribution des mots.

Comme on peut le voir dans la figure ci-dessous, le wordcloud des URLs *bénignes* est assez évident avec des tokens fréquents tels que *html*, *com*, *org*, *wiki*, etc. Les URLs de *phishing* ont des tokens fréquents tels que *tools*, *ietf*, *www*, *index*, *battle*, *net* tandis que *html*, *org*, *html* sont des tokens de fréquence plus élevée car ces URLs essaient d'imiter les URLs originales pour tromper les utilisateurs.

Wordcloud des URLs de liens *malveillants* contient des tokens de fréquence plus élevée tels que *exe*, *E7*, *BB*, *MOZI*. Ces tokens sont également évidents car les URLs de logiciels malveillants tentent d'installer des chevaux de Troie sous forme de fichiers exécutables sur le système des utilisateurs une fois que l'utilisateur visite ces URLs.

L'intention des URLs de *défiguration* est de modifier le code du site Web original, c'est pourquoi les tokens dans son wordcloud sont des termes de développement plus courants tels que *index*, *php*, *itemid*, *https*, *option*, etc.





Fig. 2. Word clouds of Benign, Phishing, Malware, and Defacement URLs

## 5- Importation des bibliothèques :

Dans cette étape, nous importerons toutes les bibliothèques *Python* nécessaires qui seront utilisées dans ce projet.

```
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 os
import seaborn as sns
from wordcloud import WordCloud
```

Ensuite, nous chargerons la base de données et vérifierons des enregistrements d'échantillon dans le jeu de données pour comprendre les données.

## 6- Chargement de base de données :

Dataset est importée de *kaggle*.



Dans cette étape, nous importerons le jeu de données à l'aide de la bibliothèque pandas et vérifierons les entrées d'échantillon dans la base de données.

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

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

(30158, 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

Ainsi, d'après la sortie ci-dessus, nous pouvons observer que le jeu de données comporte 6,51,191 enregistrements avec deux colonnes : "**url**" contenant les URL brutes et "**type**" qui est la variable cible.

```
df.type.value_counts()
```

```
type
benign      22105
defacement   5560
phishing     1756
malware       736
Name: count, dtype: int64
```

Ensuite, nous passerons à la partie de l'ingénierie des caractéristiques, où nous créerons des caractéristiques lexicales à partir des URLs brutes.

## 7- Ingénierie des caractéristiques :

Dans cette étape, nous allons extraire les caractéristiques lexicales suivantes à partir des URLs brutes, car ces caractéristiques seront utilisées comme les caractéristiques d'entrée pour

l'entraînement du modèle d'apprentissage automatique. Les caractéristiques suivantes sont créées comme suit :

- **having\_ip\_address** : Généralement, les cyber-attaquants utilisent une adresse IP à la place du nom de domaine pour masquer l'identité du site Web. Cette caractéristique vérifie si l'URL contient une adresse IP ou non.
- **abnormal\_url** : Cette caractéristique peut être extraite de la base de données WHOIS. Pour un site Web légitime, l'identité fait généralement partie de son URL.
- **google\_index** : Dans cette caractéristique, nous vérifions si l'URL est indexée dans la console de recherche Google ou non.
- **Count.** : Les sites Web de *phishing* ou de liens *malveillants* utilisent généralement plus de deux sous-domaines dans l'URL. Chaque domaine est séparé par un point (.). Si une URL contient plus de trois points (.), cela augmente la probabilité d'un site *malveillant*.
- **Count-www** : Généralement, la plupart des sites Web sûrs ont un seul www dans leur URL. Cette caractéristique aide à détecter les sites Web *malveillants* si l'URL n'a pas ou plus d'un www dans son URL.
- **count@** : La présence du symbole "@" dans l'URL ignore tout ce qui précède.
- **Count\_dir** : La présence de plusieurs répertoires dans l'URL indique généralement des sites Web suspects.
- **Count\_embed\_domain** : Le nombre de domaines intégrés peut être utile pour détecter les URL malveillantes. Cela peut être fait en vérifiant l'occurrence de "/" dans l'URL.
- **Mots suspects dans l'URL**: Les URLs *malveillantes* contiennent généralement des mots suspects dans l'URL tels que PayPal, connexion, se connecter, banque, compte, mise à jour, bonus, service, ebayisapi, jeton, etc. Nous avons trouvé la présence de tels mots suspects fréquents dans l'URL comme une variable binaire, c'est-à-dire si de tels mots sont présents dans l'URL ou non.
- **Short\_url** : Cette caractéristique est créée pour identifier si l'URL utilise des services de raccourcissement d'URL comme bit.ly, goo.gl, go2l.in, etc.
- **Count\_https** : Généralement, les URLs *malveillantes* n'utilisent pas de protocoles HTTPS car cela nécessite généralement des informations d'identification utilisateur et garantit que le site Web est sécurisé pour les transactions. Ainsi, la présence ou l'absence du protocole HTTPS dans l'URL est une caractéristique importante.
- **Count\_http** : La plupart du temps, les sites Web de *phishing* ou malveillants ont plus d'un HTTP dans leur URL tandis que les sites sûrs n'ont qu'un seul HTTP.
- **Count%** : Comme nous le savons, les URLs ne peuvent pas contenir d'espaces. L'encodage d'URL remplace normalement les espaces par le symbole (%). Les sites sûrs contiennent généralement moins d'espaces tandis que les sites Web malveillants contiennent généralement plus d'espaces dans leur URL, d'où un nombre plus élevé de .

- **Count ?** : La présence du symbole "?" dans l'URL indique une chaîne de requête contenant les données à transmettre au serveur. Un plus grand nombre de ? dans l'URL indique certainement une URL suspecte.
- **Count-** : Les fraudeurs ou les cybercriminels ajoutent généralement des tirets (-) en préfixe ou en suffixe du nom de marque afin qu'il semble être une URL authentique. Par exemple. [www.flipkart-india.com](http://www.flipkart-india.com).
- **Count=** : La présence du signe égal (=) dans l'URL indique le passage de valeurs de variable d'une page de formulaire à une autre. Cela est considéré comme plus risqué dans l'URL car n'importe qui peut modifier les valeurs pour modifier la page.
- **url\_length** : Les attaquants utilisent généralement des URLs longues pour masquer le nom de domaine. Nous avons trouvé que la longueur moyenne d'une URL sûre est de 74.
- **hostname\_length** : La longueur du nom d'hôte est également une caractéristique importante pour détecter les URLs *malveillantes*.
- **Longueur du premier répertoire** : Cette caractéristique aide à déterminer la longueur du premier répertoire dans l'URL. Ainsi, en recherchant le premier '/', la longueur de l'URL jusqu'à ce point aide à trouver la longueur du premier répertoire de l'URL. Pour accéder aux informations de niveau de répertoire, nous devons installer la bibliothèque python TLD.
- **Longueur des domaines de premier niveau** : Un domaine de premier niveau (TLD) est l'un des domaines au plus haut niveau dans le système hiérarchique de noms de domaine de l'Internet. Par exemple, dans le nom de domaine [www.example.com](http://www.example.com), le domaine de premier niveau est com. Ainsi, la longueur du TLD est également importante pour identifier les URLs *malveillantes*. Comme la plupart des URLs ont une extension .com. Les TLD dans la plage de 2 à 3 indiquent généralement des URLs sûres.
- **Count\_digits** : La présence de chiffres dans l'URL indique généralement des URL suspectes. Les URL sûres n'ont généralement pas de chiffres, donc compter le nombre de chiffres dans l'URL est une caractéristique importante pour détecter les URLs *malveillantes*.
- **Count\_letters** : Le nombre de lettres dans l'URL joue également un rôle significatif dans l'identification des URLs *malveillantes*. Comme les attaquants essaient d'augmenter la longueur de l'URL pour masquer le nom de domaine et cela se fait généralement en augmentant le nombre de lettres et de chiffres dans l'URL.

```
[10] import re
#Use of IP or not in domain
def having_ip_address(url):
    match = re.search(
        '([01]?\\d\\d?[2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?[2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?[2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?[2[0-4]\\d|25[0-5])\\.'
        '([01]?\\d\\d?[2[0-4]\\d|25[0-5])\\.|' # IPv4
        '((0x[0-9a-fA-F]{1,2})\\. (0x[0-9a-fA-F]{1,2})\\. (0x[0-9a-fA-F]{1,2})\\. (0x[0-9a-fA-F]{1,2})\\.|' # IPv6 in hexadecimal
        '(?:[a-fA-F0-9]{1,4}:){7}[a-fA-F0-9]{1,4}', url) # IPv6
    if match:
        # print match.group()
        return 1
    else:
        # print 'No matching pattern found'
        return 0
df['use_of_ip'] = df['url'].apply(lambda i: having_ip_address(i))
```

```
from urllib.parse import urlparse

def abnormal_url(url):
    hostname = urlparse(url).hostname
    hostname = str(hostname)
    match = re.search(hostname, url)
    if match:
        # print match.group()
        return 1
    else:
        # print 'No matching pattern found'
        return 0

df['abnormal_url'] = df['url'].apply(lambda i: abnormal_url(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',
        url)
    if match:
        return 1
    else:
        return 0
```

```
[ ] 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))

#Length of URL
df['url_length'] = df['url'].apply(lambda i: url_length(i))
#Hostname Length

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

```
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

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

df.head()
```

Maintenant, après avoir créé les 22 caractéristiques ci-dessus, le jeu de données ressemble à ce qui suit.

	url	type	use_of_ip	abnormal_url	google_index	count.	count-www	count@	count_dir	count_embed_domian	...	count-http	count%	count?	count-	count=	url_le
0	br-icloud.com.br	phishing	0	0	1	2	0	0	0	0	...	0	0	0	1	0	
1	mp3raid.com/music/knizz_kaliko.html	benign	0	0	1	2	0	0	2	0	...	0	0	0	0	0	
2	bopsecrets.org/rexroth/cr/1.htm	benign	0	0	1	2	0	0	3	0	...	0	0	0	0	0	
3	http://www.garage-pirene.be/index.php?option=...	defacement	0	1	1	3	1	0	1	0	...	1	0	1	1	4	
4	http://adventure-nicaragua.net/index.php?optio...	defacement	0	1	1	2	0	0	1	0	...	1	0	1	1	3	

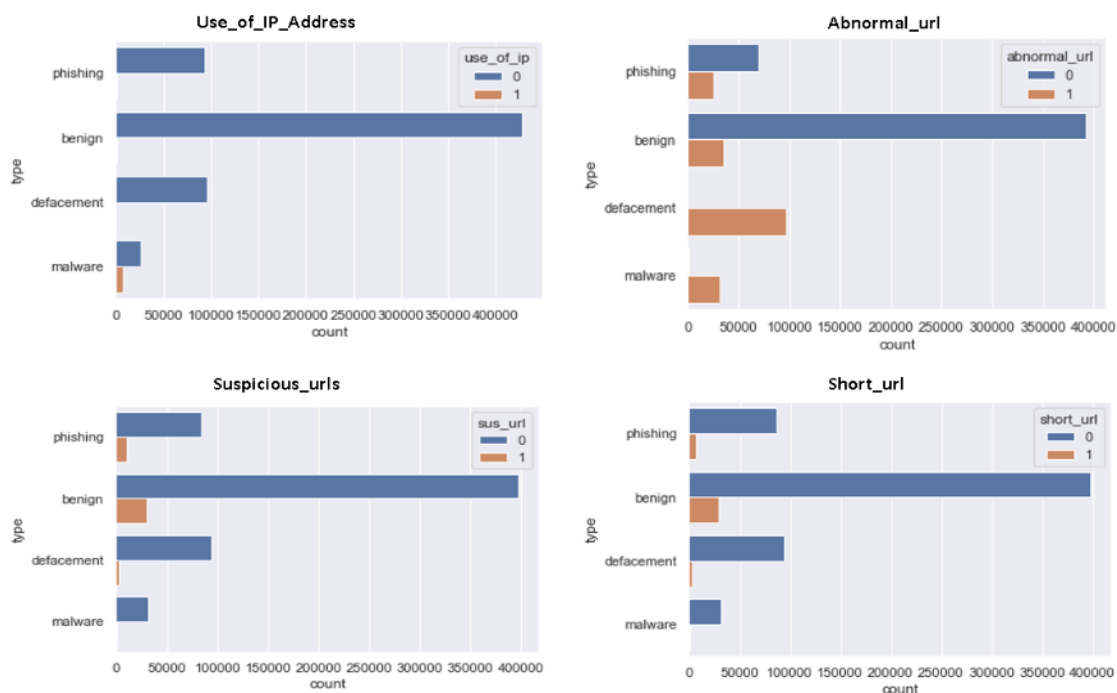
5 rows x 22 columns

Maintenant, dans la prochaine étape, nous supprimons les colonnes **non pertinentes**, c'est-à-dire **URL**, **google\_index** et **tld**.

La raison de la suppression de la colonne URL est que nous avons déjà extrait les caractéristiques pertinentes à partir de celle-ci qui peuvent être utilisées en entrée dans les algorithmes d'apprentissage automatique.

La caractéristique google\_index indique si l'URL est indexée dans la console de recherche Google ou non. Dans ce jeu de données, toutes les URL sont indexées sur Google et ont une valeur de 1.

Analyse exploratoire des données (EDA) Dans cette étape, nous allons vérifier la distribution des différentes caractéristiques pour les quatre classes d'URL.





Comme nous pouvons le constater à partir de la distribution ci-dessus de la caractéristique `use_ip_address`, seules les URLs de liens *malveillants* ont des adresses IP. En ce qui concerne `abnormal_url`, les URLs de *défiguration* ont une distribution plus élevée.

De la distribution des `suspicious_urls`, il est clair que les URLs *benignes* ont la plus forte distribution tandis que les URLs de *phishing* ont une deuxième distribution la plus élevée. Comme les URLs suspectes contiennent des mots-clés liés aux transactions et aux paiements et que généralement les URLs bancaires ou de paiement authentiques contiennent de tels mots-clés, c'est pourquoi les URLs *benignes* ont la plus forte distribution.

En ce qui concerne la distribution des `short_url`, nous pouvons observer que les URLs *benignes* ont les plus courtes URL, car généralement, nous utilisons des services de raccourcissement d'URL pour partager facilement des URLs de grande longueur.

## 8- Encodage des étiquettes :

Après cela, l'étape la plus importante est d'étiqueter et d'encoder la variable cible (*type*) afin de pouvoir la convertir en catégories numériques 0, 1, 2 et 3. Comme les algorithmes d'apprentissage automatique comprennent uniquement les variables cibles numériques.

```
✓ [31] from sklearn.preprocessing import LabelEncoder
0s

lb_make = LabelEncoder()
df["type_code"] = lb_make.fit_transform(df["type"])
df["type_code"].value_counts()

type_code
0      22105
1       5560
3       1756
2        736
4          1
Name: count, dtype: int64
```

## 9- Séparation des variables caractéristiques et cibles :

Dans la prochaine étape, nous avons créé une variable prédicteur et une variable cible. Ici, les variables prédicteur sont les variables indépendantes, c'est-à-dire les caractéristiques de l'URL, et la variable cible est le type.

```
[33] #Predictor Variables
# filtering out google_index as it has only 1 value
X = df[['use_of_ip', 'abnormal_url', 'count.', 'count-www', 'count@',
        'count_dir', 'count_embed_domian', 'short_url', 'count-https',
        'count-http', 'count%', 'count?', 'count-', 'count=', 'url_length',
        'hostname_length', 'sus_url', 'fd_length', 'tld_length', 'count-digits',
        'count-letters']]

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

```
X.head()
```

	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	hostname_
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	

5 rows x 21 columns

## 10- Division entraînement/test :

La prochaine étape consiste à diviser le jeu de données en ensembles d'entraînement et de test. Nous avons divisé le jeu de données selon un ratio de 80:20, c'est-à-dire que 80% des données ont été utilisées pour *entraîner* les modèles d'apprentissage automatique, et les 20% restants ont été utilisés pour *tester* le modèle.

Comme nous le savons, nous avons un jeu de données déséquilibré. La raison en est qu'environ 66% des données sont des URLs bénignes, 5% sont des URLs malveillants, 14% des URLs de phishing et 15% des URLs de défiguration. Ainsi, après avoir divisé aléatoirement le jeu de données en ensembles d'entraînement et de test, il se peut que la distribution des différentes catégories soit perturbée, ce qui affectera grandement les performances du modèle d'apprentissage automatique. Pour maintenir la même proportion de la variable cible, une stratification est nécessaire.

Ce paramètre de stratification effectue une division de telle sorte que la proportion de valeurs dans l'échantillon produit soit la même que la proportion de valeurs fournies au paramètre de stratification.

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

## 11- Construction du modèle :

```
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']))
```

	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	0.96	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

## 12- L'évaluation du modèle :

```
from sklearn import metrics

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

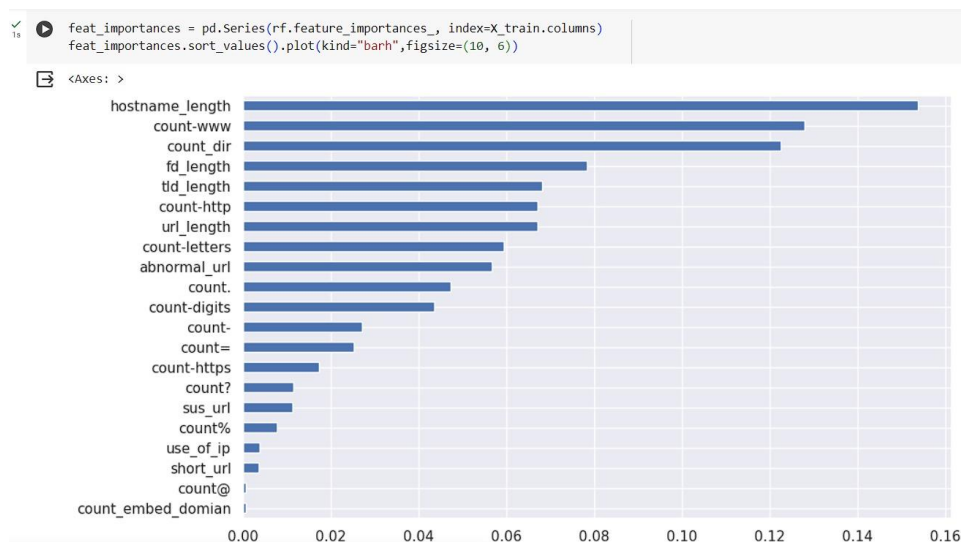
```
accuracy:  0.966
```

D'après les résultats ci-dessus, il est évident que **Random Forest** présente une bonne performance en termes de précision de test, car il atteint la plus haute précision de 96,6% avec un taux de détection plus élevé pour les URL *bénignes*, de *défiguration*, de *phishing* et de liens *malveillants*.

Donc, nous avons sélectionné **Random Forest** comme notre principal modèle pour détecter les URL malveillantes, et dans la prochaine étape, nous allons également tracer le graphique de l'importance des caractéristiques.

## 13- Importance des Caractéristiques :

Après avoir sélectionné notre modèle, c'est-à-dire **Random Forest**, nous allons maintenant vérifier les caractéristiques les plus contributives. Voici le code pour tracer le graphique de l'importance des caractéristiques.



À partir du graphique ci-dessus, nous pouvons observer que les 5 principales caractéristiques pour détecter les URL malveillantes sont la longueur du nom d'hôte, le nombre de répertoires, le nombre de "www", la longueur du premier répertoire et la longueur de l'URL.

## 14- Prédiction du modèle :

Dans cette dernière étape, nous allons prédire les URL malveillantes à l'aide de notre modèle c'est-à-dire **Random Forest**.

```
[ ] 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))
    status.append(letter_count(url))
    status.append(fd_length(url))
    tld = get_tld(url, fail_silently=True)

    status.append(tld_length(tld))

    return status
```

```
[ ] 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 = rf.predict(features_test)
    if int(pred[0]) == 0:
        res="SAFE"
        return res
    elif int(pred[0]) == 1.0:
        res="DEFAACEMENT"
        return res
    elif int(pred[0]) == 2.0:
        res="PHISHING"
        return res
    elif int(pred[0]) == 3.0:
        res="MALWARE"
        return res
```


## 15- Enregistrement du modèle :

Le modèle **Random Forest** que nous avons développé a été enregistré à l'aide de Joblib. Cela permet de sauvegarder le modèle entraîné dans un fichier, ce qui le rend facilement accessible et réutilisable dans d'autres environnements.

```
✓ [41] import joblib
    joblib.dump(rf, 'rf.joblib')

['rf.joblib']
```

Le modèle est généré avec succès :

 rf.joblib

## Chapitre 2 : déploiement du modèle avec Docker

### 1- Outils et environnement :

#### ★ Docker :



Docker est un **outil** qui permet aux développeurs, aux administrateurs système, etc. de **déployer facilement** leurs applications dans un bac à sable (appelé *conteneurs*) pour les exécuter sur le système d'exploitation hôte, c'est-à-dire Linux. Le principal avantage de Docker est qu'il permet aux utilisateurs de regrouper une application avec toutes ses dépendances dans une unité standardisée pour le développement logiciel. Contrairement aux machines virtuelles, les conteneurs n'ont pas de surcharge importante et permettent donc une utilisation plus efficace du système et des ressources sous-jacentes.

#### ★ FastAPI :



FastAPI est un framework Web moderne publié pour la première fois en 2018 pour créer des **API RESTful** en *Python*. Il est utilisé pour créer des API avec *Python 3.8+* basées sur des astuces standard de type *Python*.

FastAPI est basé sur *Pydantic* et utilise des indices de type pour valider, sérialiser et désérialiser les données. Il génère également automatiquement la documentation **OpenAPI** pour les API créées avec lui.

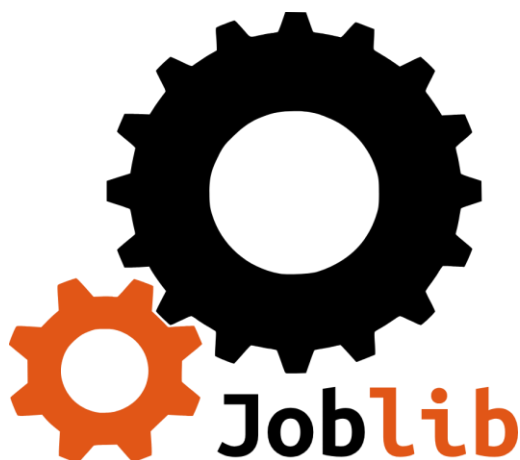


★ Python :



Python est un **langage de programmation** interprété, multiparadigme et multiplateformes. Il favorise la programmation impérative structurée, fonctionnelle et orientée objet. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions.

★ Joblib :



Joblib est un **ensemble d'outils** permettant de fournir un pipeline léger en *Python*. En particulier : La mise en cache transparente des fonctions sur disque, la réévaluation paresseuse (modèle de mémorisation) et le calcul parallèle simple et facile. Joblib est optimisé pour être rapide et robuste sur les données volumineuses en particulier et dispose d'optimisations spécifiques pour les tableaux *numpy*. Il est sous licence BSD.

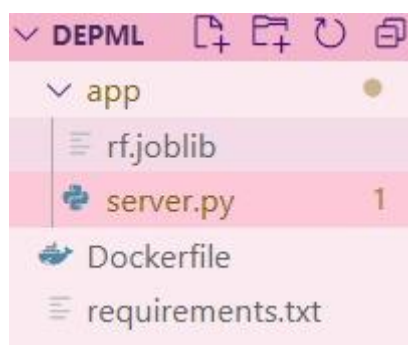
★ Unicorn :



Uvicorn est une **implémentation de serveur Web ASGI** pour *Python*. Jusqu'à récemment, *Python* manquait d'une interface serveur/application minimale de bas niveau pour les frameworks asynchrones. La spécification *ASGI* comble cette lacune et signifie que nous sommes désormais en mesure de commencer à créer un ensemble commun d'outils utilisables dans tous les frameworks asynchrones.

## 2- Simulation :

Pour déployer notre projet, on crée les dossiers et fichiers suivants selon une architecture précise :



Un dossier app qui contient les deux fichiers : **rf.joblib** et **server.py**.

Le fichier **rf.joblib** est le modèle de prédiction généré après l'entraînement mentionné dans chapitre 1.

Le fichier **server.py** contient la création des APIs à l'aide de *FastAPI*. On définit aussi les fonctions utilisées pour identifier les malware URLs avec route `/predict1` qui possède une fonction `get_prediction_from_url(url)` qui prend comme argument un lien et donne comme résultat et selon le modèle entraîné s'il est dangereux ou non.

```

app > server.py > reed_root
1  from fastapi import FastAPI
2  import joblib
3  import numpy as np
4  import re
5  from urllib.parse import urlparse
6  from urllib.parse import urlparse
7  from tld import get_tld
8
9  model = joblib.load('app/rf.joblib')
10
11  class_names = np.array(['benign', 'defacement', 'phishing', 'malware'])
12
13  app = FastAPI()
14
15  > def having_ip_address(url): ...
16
17
18  > def abnormal_url(url): ...
19
20
21  > def count_dot(url): ...
22
23
24  > def count_www(url): ...
25
26
27  > def count_atrate(url): ...
28
29
30  > def no_of_dir(url): ...
31
32
33  > def no_of_embed(url): ...
34
35
36  > def shortening_service(url): ...
37
38
39  > def count_https(url): ...
40
41
42  > def count_http(url): ...
43
44
45  > def count_per(url): ...
46
47
48  > def count_ques(url): ...
49
50
51  > def count_hyphen(url): ...
52
53
54  > def count_equal(url): ...
55
56
57  > def url_length(url): ...
58
59
60  > def hostname_length(url): ...
61
62
63  > def suspicious_words(url): ...
64
65
66  > def digit_count(url): ...
67
68
69  > def letter_count(url): ...
70
71
72  > def fd_length(url): ...
73
74
75  > def tld_length(tld): ...
76
77
78  > def main(url): ...
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169  @app.get('/')
170  def reed_root():
171      return {'message': 'Url safety prediction'}
172
173  @app.post('/predict1')
174  def get_prediction_from_url(url: str):
175      features = main(url)
176      features = np.array(features).reshape(1, -1)
177
178      pred = model.predict(features)
179      class_name = class_names[pred][0]
180      if int(pred[0]) == 0:
181
182
183          return {'predicted_class': class_name}
184      elif int(pred[0]) == 1.0:

```

```

184 elif int(pred[0]) == 1.0:
185
186     return {'predicted_class': class_name}
187 elif int(pred[0]) == 2.0:
188     return {'predicted_class': class_name}
189
190 elif int(pred[0]) == 3.0:
191     return {'predicted_class': class_name}

```

On trouve aussi un fichier **Dockerfile** qui contient les instructions nécessaires à la création d'une image de conteneur appelé dans notre cas *appml*.

```

Dockerfile > ...
1 FROM python:3.11
2
3 WORKDIR /code
4
5 COPY ./requirements.txt /code/requirements.txt
6
7 RUN pip install --no-cache-dir -r /code/requirements.txt
8
9 COPY ./app /code/app
10
11 EXPOSE 8000
12
13 CMD ["uvicorn", "app.server:app", "--host", "0.0.0.0", "--port", "8000"]

```

Et finalement, on a un fichier texte **requirements.txt** où on précise les différentes bibliothèques avec lesquelles on a travaillé pour la création d'image.

```

requirements.txt
1 scikit-learn==1.2.2
2 fastapi
3 numpy
4 uvicorn
5 joblib
6 tld

```

Pour démarrer la création d'image de conteneur on commence tout d'abord par l'exécution de la commande suivante :

```

PS C:\Users\Mouna\OneDrive\Desktop\depML> docker buildx build -t appml .
[+] Building 0.0s (0/0) docker:default
[+] Building 11.1s (9/10)
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 301B
=> [internal] load metadata for docker.io/library/python:3.11
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 5.34kB
=> [1/5] FROM docker.io/library/python:3.11@sha256:e453eb723bc8ecac7a797498f9a5915d13e567620d48dcd3568750bac3b59f31
=> CACHED [2/5] WORKDIR /code
=> CACHED [3/5] COPY ./requirements.txt /code/requirements.txt
=> CACHED [4/5] RUN pip install --no-cache-dir -r /code/requirements.txt
=> [5/5] COPY ./app /code/app

```

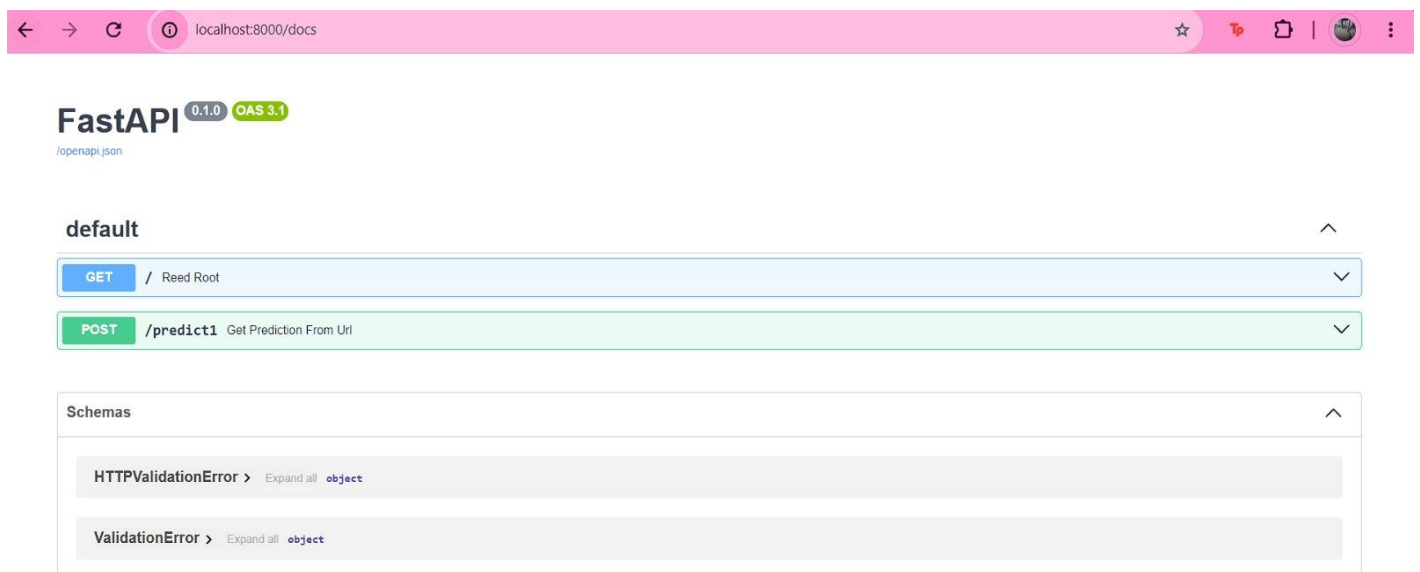
Et après on exécute une instance d'un conteneur Docker appelé *containerdepml* en utilisant une image créée précédemment.

```
PS C:\Users\Mouna\OneDrive\Desktop\depML> docker run --name containerdepml -p 8000:8000 appml
INFO:      Started server process [1]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
INFO:      Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

Le conteneur est exécuté avec succès :



On accède '**localhost:8000/docs**' pour afficher l'interface de l'exploration et de test des APIs à laide de *FastAPI*.



On teste tout d'abord par la saisie d'un lien sécurisé (*benign URL*)

Name	Description
url * required string (query)	mp3raid.com/music/krizz_kaliko.html

Execute Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8000/predict?url=mp3raid.com%2Fmusic%2Fkrizz_kaliko.html' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

```
http://localhost:8000/predict?url=mp3raid.com%2Fmusic%2Fkrizz_kaliko.html
```

Server response

Code	Details
200	<p>Response body</p> <pre>{   "predicted_class": "benign" }</pre> <p>Download</p>

On reçoit un code 200 qui démontre le fonctionnement de modèle avec comme résultat la classe prédite **'benign'**.

On teste encore une fois le cas d'un lien malveillant (*malware URL*) :

Name	Description
url * required string (query)	trtsport.cz

Execute Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8000/predict?url=trtsport.cz' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

```
http://localhost:8000/predict?url=trtsport.cz
```

Server response

Code	Details
200	<p>Response body</p> <pre>{   "predicted_class": "malware" }</pre> <p>Download</p>

On obtient bien que c'est un lien malveillant.

Pour le cas d'un lien de défiguration (*defacement URL*) :



Name	Description
url * required string (query)	<input type="text" value="http://www.natureevent.dk/component/mailto/"/>

Execute
Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8000/predict?url=http%3A%2F%2Fwww.natureevent.dk%2Fcomponent%2Fmailto%2Findex.html%3Ftmpl%3Dcomponent%26link%3DaHR0cDovL3d3dy5uYXR1cmV1dmVudC5kay9ha3Rpdml0ZXR1ci5odG1s' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

```
http://localhost:8000/predict?url=http%3A%2F%2Fwww.natureevent.dk%2Fcomponent%2Fmailto%2Findex.html%3Ftmpl%3Dcomponent%26link%3DaHR0cDovL3d3dy5uYXR1cmV1dmVudC5kay9ha3Rpdml0ZXR1ci5odG1s
```

Server response

Code	Details
200	<p>Response body</p> <pre>{   "predicted_class": "defacement" }</pre> <div> <span>Download</span> </div>

Le cas d'un lien de phishing (*phishing URL*) :

Name	Description
url * required string (query)	<input type="text" value="://drive-google-com.fanalav.com/6a7ec96d6a"/>

Execute
Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8000/predict?url=http%3A%2F%2Fdrive-google-com.fanalav.com%2F6a7ec96d6a' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

```
http://localhost:8000/predict?url=http%3A%2F%2Fdrive-google-com.fanalav.com%2F6a7ec96d6a
```

Server response

Code	Details
200	<p>Response body</p> <pre>{   "predicted_class": "phishing" }</pre> <div> <span>Download</span> </div>

On peut bien remarquer que la sécurité des liens est prédite fidèlement.

## Conclusion

Dans ce projet, nous avons exploré en profondeur le concept des URL et avons examiné leur utilisation potentielle pour des activités malveillantes. Nous avons commencé par préparer un modèle de prédiction pour identifier les URL malveillantes, en décrivant le problème, en collectant et en analysant les données, et en construisant un modèle de prédiction basé sur des caractéristiques pertinentes extraites des URL. Après avoir évalué et validé notre modèle, nous avons entrepris de le déployer à l'aide de Docker, en utilisant les outils appropriés et en simulant le déploiement pour garantir son bon fonctionnement dans un environnement de production.

Ce projet a permis de mettre en lumière l'importance de comprendre les URL et les menaces potentielles associées à leur utilisation malveillante. Il a également démontré l'efficacité de l'apprentissage automatique pour détecter de telles activités et la facilité avec laquelle les modèles peuvent être déployés à grande échelle à l'aide de technologies comme Docker. En conclusion, ce projet représente une étape significative vers la création d'outils de sécurité efficaces pour protéger les utilisateurs contre les menaces en ligne.