Phishing Website Detection by Machine Learning Techniques

1. OBJECTIVE

The objective of this project is to train machine learning models and deep neural networks to predict phishing websites using a dataset containing features extracted from both phishing and benign URLs. The project aims to compare the performance of various models and techniques in accurately identifying phishing websites, thus enhancing cybersecurity measures.

2. SOFTWARE DETAILS

The project utilizes Python as the primary programming language along with popular libraries such as Pandas, NumPy, Seaborn, Matplotlib, and Scikit-learn for data manipulation, visualization, and machine learning tasks. The XGBoost library is employed for implementing the eXtreme Gradient Boosting algorithm.

3. ABSTRACT/ INTRODUCTION

Phishing websites pose a significant threat to internet users by masquerading as legitimate websites to steal sensitive information. A phishing website is a common social engineering method that mimics trustful uniform resource locators (URLs) and webpages. The objective of this project is to train machine learning models and deep neural nets on the dataset created to predict phishing websites.

Both phishing and benign URLs of websites are gathered to form a dataset and from them required URL and website content-based features are extracted. The performance level of each model is measured and compared.

Below mentioned are the steps involved in the completion of this project:

• Collect dataset containing phishing and legitimate websites from the open source

platforms. • Write a code to extract the required features from the URL database.

• Analyze and preprocess the dataset by using EDA techniques.

• Divide the dataset into training and testing sets.
• Run selected machine learning and deep neural network algorithms like SVM, Random Forest, Autoencoder on the dataset.

• Write a code for displaying the evaluation result considering accuracy metrics.

• Compare the obtained results for trained models and specify which is better.

4. BLOCK DIAGRAM / DATASET DESCRIPTION / FEATURE EXTRACTION /CLASSIFIER EXTRACTION

The dataset consists of 10,000 entries with 18 columns, including features such as 'Domain,' 'Have_IP,' 'Have_At,' 'URL_Length,' 'URL_Depth,' and more. These features are extracted from both phishing and benign URLs.

Features Extracted: Features extracted include indicators like whether the URL has an IP address, '@' symbol, URL length, URL depth, redirection, HTTPS domain, TinyURL, prefix/suffix, DNS record, web traffic, domain age, and more.

The following category of features are selected:

• Address Bar based Features

• Domain based Features

• HTML & Javascript based Feature

• Address Bar based Features considered are:

• Domain of URL

• Redirection ‘//’ in URL

• IP Address in URL

• ‘http/https’ in Domain name

• ‘@’ Symbol in URL

• Using URL Shortening Service

• Length of URL

• Prefix or Suffix "-" in Domain

• Depth of URL

Domain based Features considered are:

• HTML and JavaScript based Features considered are:

• All together 17 features are extracted from the dataset.

•DNS Record

•Age of Domain

•Website Traffic

•End Period of Domain

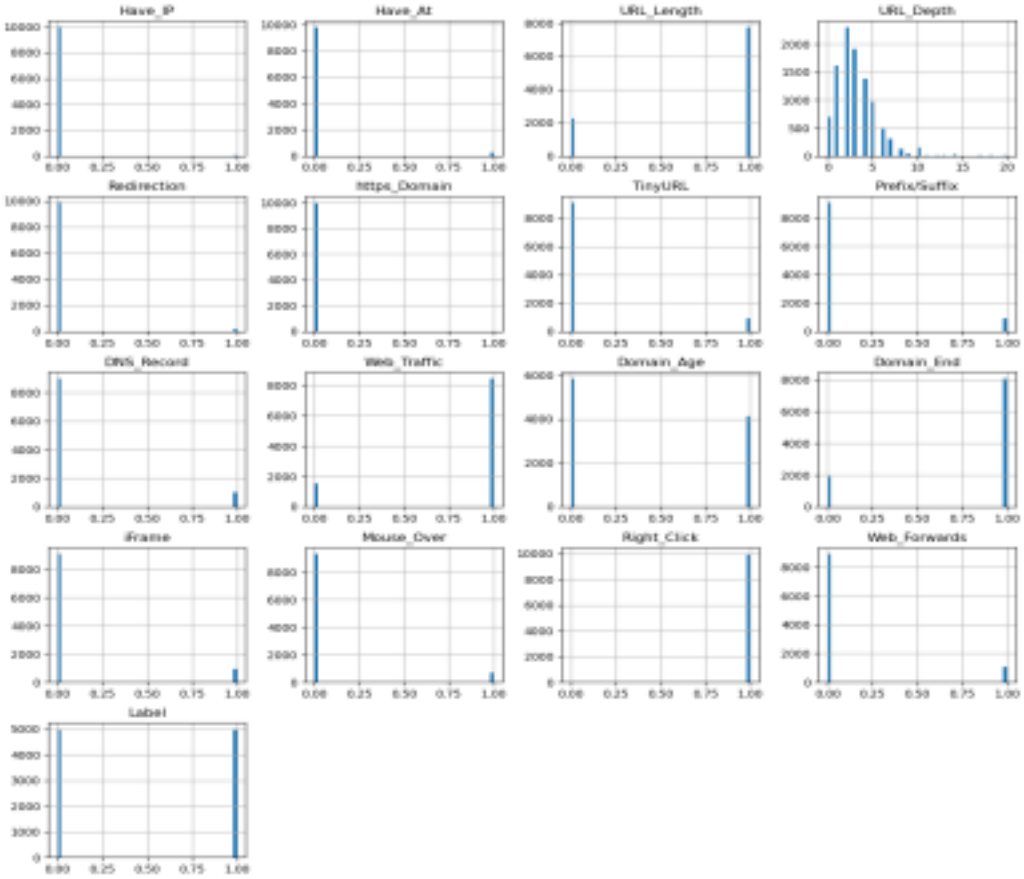
•Iframe Redirection

•Disabling Right Click

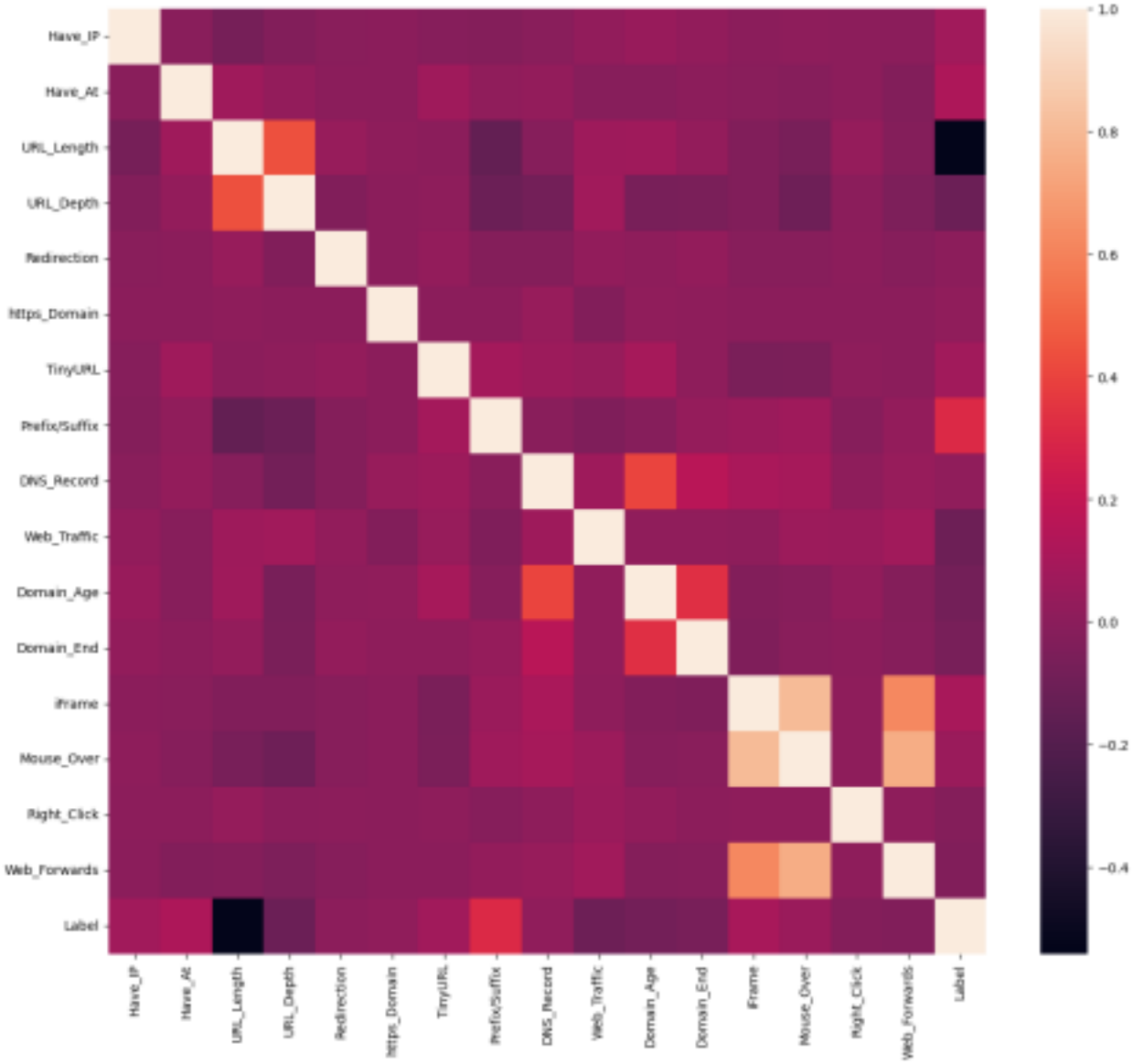
•Status Bar Customization

•Website Forwarding

Features Distribution:



Confusion Matrix/Heat Map Explanation: The confusion matrix and heatmap are used to visualize the performance of the classifier by comparing predicted labels against actual labels. This helps in understanding the model's ability to correctly classify phishing and benign URLs and identify any misclassifications.



5. CODE

```
#importing basic packages
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

data0 = pd.read_csv('5.urldata.csv')
data = data.sample(frac=1).reset_index(drop=True)
data.head()

#Dropping the Domain column
```

```
data = data0.drop(['Domain'], axis = 1).copy()
#checking the data for null or missing values
data.isnull().sum()
# shuffling the rows in the dataset so that when splitting the train and test set are equally
distributed
data = data.sample(frac=1).reset_index(drop=True)
# Separating & assigning features and target columns to X & y
y = data['Label']
X = data.drop('Label',axis=1)
X.shape, y.shape
# Splitting the dataset into train and test sets: 80-20 split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.2, random_state = 12)
X_train.shape, X_test.shape
#importing packages
from sklearn.metrics import accuracy_score
# Creating holders to store the model performance results
ML_Model = []
acc_train = []
acc_test = []
#function to call for storing the results
def storeResults(model, a,b):
ML_Model.append(model)
acc_train.append(round(a, 3))
acc_test.append(round(b, 3))
```

 # XGBoost Classification model

```
from xgboost import XGBClassifier
# instantiate the model
xgb = XGBClassifier(learning_rate=0.4,max_depth=7)
#fit the model
xgb.fit(X_train, y_train)
#predicting the target value from the model for the samples
y_test_xgb = xgb.predict(X_test)
y_train_xgb = xgb.predict(X_train)
#computing the accuracy of the model performance
acc_train_xgb = accuracy_score(y_train,y_train_xgb)
acc_test_xgb = accuracy_score(y_test,y_test_xgb)
print("XGBoost: Accuracy on training Data: {:.3f}".format(acc_train_xgb))
print("XGBoost : Accuracy on test Data: {:.3f}".format(acc_test_xgb)) #storing
the results. The below mentioned order of parameter passing is important.
#Caution: Execute only once to avoid duplications.
```

```
storeResults('XGBoost', acc_train_xgb, acc_test_xgb)
```

 # Autoencoder Neural Network

```
from sklearn.metrics import accuracy_score
ML_Model = []
acc_train = []
acc_test = []
#function to call for storing the results
def storeResults(model, a,b):
ML_Model.append(model)
acc_train.append(round(a, 3))
acc_test.append(round(b, 3))
#importing required packages
import keras
from keras.layers import Input, Dense
from keras import regularizers
import tensorflow as tf
from keras.models import Model
from sklearn import metrics
#building autoencoder model
input_dim = X_train.shape[1]
encoding_dim = input_dim
input_layer = Input(shape=(input_dim, ))
encoder = Dense(encoding_dim, activation="relu",
activity_regularizer=regularizers.l1(10e-4))(input_layer)
encoder = Dense(int(encoding_dim), activation="relu")(encoder)
encoder = Dense(int(encoding_dim-2), activation="relu")(encoder)
code = Dense(int(encoding_dim-4), activation='relu')(encoder)
decoder = Dense(int(encoding_dim-2), activation='relu')(code)
decoder = Dense(int(encoding_dim), activation='relu')(encoder)
decoder = Dense(input_dim, activation='relu')(decoder)
autoencoder = Model(inputs=input_layer, outputs=decoder)
#compiling the model
autoencoder.compile(optimizer='adam',
loss='binary_crossentropy',
metrics=['accuracy'])
#Training the model
history = autoencoder.fit(X_train, X_train, epochs=10, batch_size=64, shuffle=True,
validation_split=0.2)
acc_train_auto = autoencoder.evaluate(X_train, X_train)[1]
acc_test_auto = autoencoder.evaluate(X_test, X_test)[1]
print('\nAutoencoder: Accuracy on training Data: {:.3f}' .format(acc_train_auto))
print('Autoencoder: Accuracy on test Data: {:.3f}' .format(acc_test_auto))
```

```
#storing the results. The below mentioned order of parameter passing is important.
#Caution: Execute only once to avoid duplications.
storeResults('AutoEncoder', acc_train_auto, acc_test_auto)
```

 # Multilayer Perceptrons

```
#importing packages
from sklearn.metrics import accuracy_score

# Creating holders to store the model performance results
ML_Model = []
acc_train = []
acc_test = []
#function to call for storing the results
def storeResults(model, a,b):
ML_Model.append(model)
acc_train.append(round(a, 3))
acc_test.append(round(b, 3)

# Multilayer Perceptrons model
from sklearn.neural_network import MLPClassifier
# instantiate the model
mlp = MLPClassifier(alpha=0.001, hidden_layer_sizes=([100,100,100]))
# fit the model
mlp.fit(X_train, y_train)
#predicting the target value from the model for the samples
y_test_mlp = mlp.predict(X_test)
y_train_mlp = mlp.predict(X_train)
#computing the accuracy of the model performance
acc_train_mlp = accuracy_score(y_train,y_train_mlp)
acc_test_mlp = accuracy_score(y_test,y_test_mlp)
print("Multilayer Perceptrons: Accuracy on training Data:
{:.3f}".format(acc_train_mlp)) print("Multilayer Perceptrons: Accuracy on test Data:
{:.3f}".format(acc_test_mlp)) storeResults('Multilayer Perceptrons', acc_train_mlp,
acc_test_mlp)
results = pd.DataFrame({ 'ML Model': ML_Model,
 'Train Accuracy': acc_train,
 'Test Accuracy': acc_test})
results
```

OUTPUT / RESULT:

	ML Model	Train Accuracy	Test Accuracy
1	XGBoost	0.866	0.865
0	Multilayer Perceptrons	0.862	0.861
2	AutoEncoder	0.827	0.844

Upon comparison, we have deduced that the XGBoost model gives better

performance.

7. CONCLUSION

In conclusion, the project successfully demonstrated the effectiveness of machine learning, specifically the XGBoost classifier, in predicting phishing websites based on extracted features from URLs. The model achieved high accuracy on both training and test datasets, indicating its robustness in identifying malicious URLs. Such techniques can be instrumental in enhancing cybersecurity measures and protecting users from falling victim to phishing attacks.

NUPUR CHATTERJEE