PHISHING WEBSITE DETECTOR

The internet is now a core part of our daily lives, but it also opens the door to cyber threats like phishing. In phishing attacks, hackers try to trick people by creating fake websites or using social engineering to steal sensitive information such as usernames, passwords, or account details. Although many solutions have been developed to detect phishing websites, attackers keep evolving their techniques to avoid detection. One of the most effective ways to identify phishing is through **Machine Learning**, since phishing sites usually share certain recognizable patterns.

Steps in this project:

1. Load the dataset
2. Explore and understand the data (EDA)
3. Visualize the features
4. Split the dataset into training and testing sets
5. Train different machine learning models
6. Compare model performance
7. Draw conclusions

1. Loading the Data

We use a dataset from Kaggle: Phishing Website Detector.
It contains over **11,000 website URLs**, each with **30 features** (like domain details, URL patterns, etc.) and a **class label** that marks the site as *phishing* (1) or *legitimate* (-1).
In total, the dataset has **11,054 samples with 32 columns**.

```
#Loading data into dataframe

data = pd.read_csv("phishing.csv")
data.head()
```

2. Exploring the Data (EDA)
At this stage, we get familiar with the dataset by:

- Checking the shape (rows × columns)
- Listing all feature names
- Looking at basic dataset info (datatypes, null values, etc.)
- Checking the uniqueness of values in each column

```
#droping index column

data = data.drop(['Index'],axis = 1)

#description of dataset
data.describe().T
```

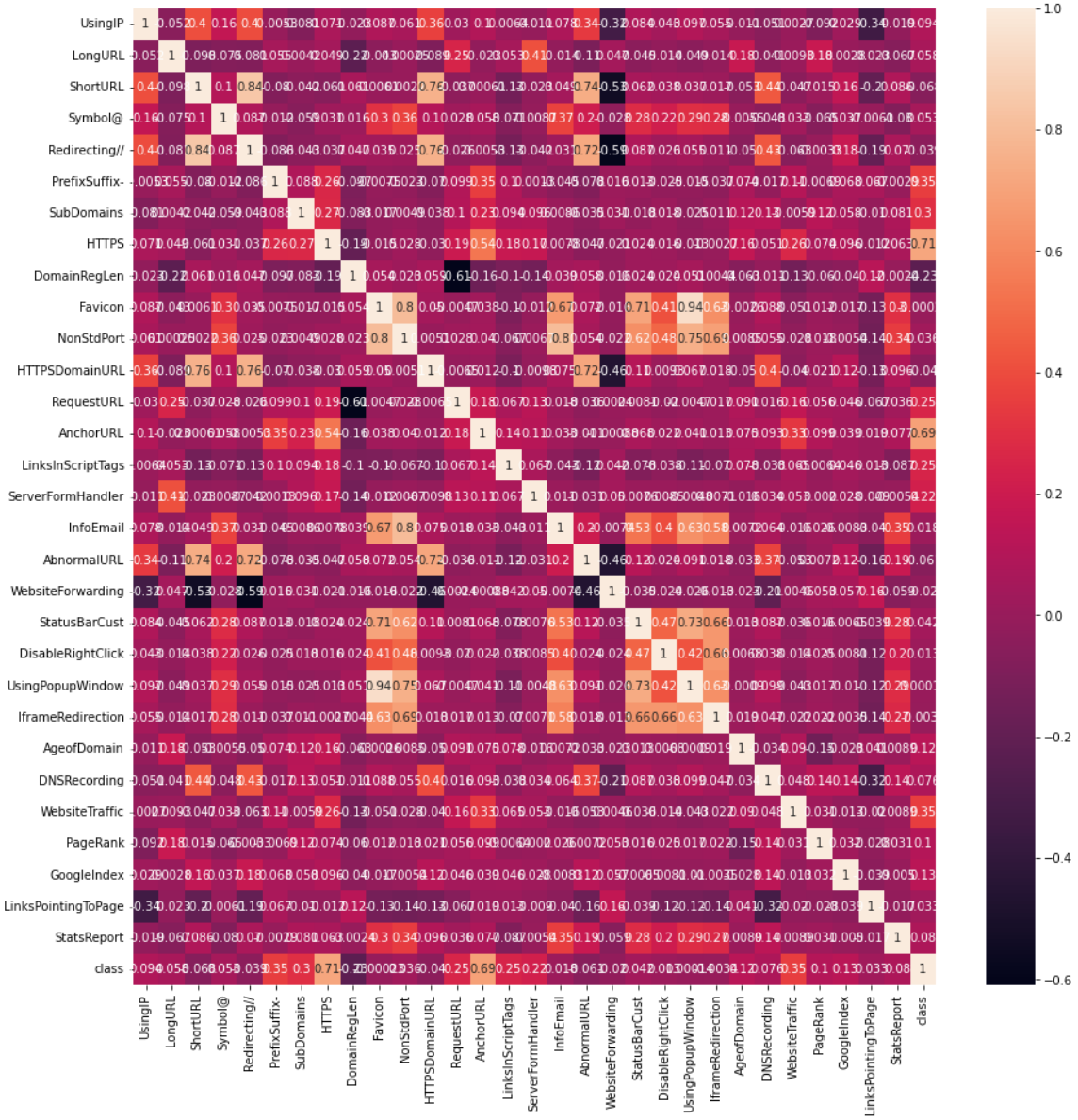
data_set.append(9 OBSERVATIONS:

1. There are 11054 instances and 31 fearures in dataset.
2. Out of which 30 are independent features where as 1 is dependent feature.
3. Each feature is in int datatype, so there is no need to use LabelEncoder.
4. There is no outlier present in dataset.
5. There is no missing value in dataset.

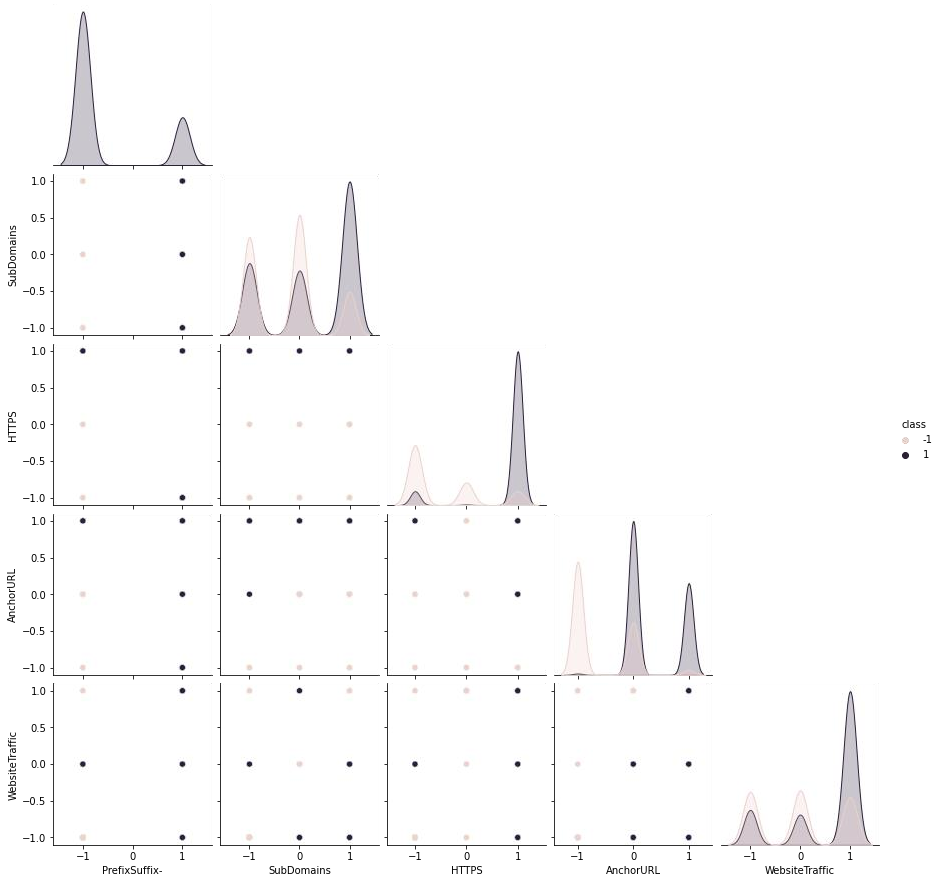
2. Data Visualization
Graphs and plots are used to understand how features are distributed and how phishing websites differ from legitimate ones. Visualizations help identify patterns that models can later use.

```
#Correlation heatmap

plt.figure(figsize=(15,15))
sns.heatmap(data.corr(), annot=True)
```

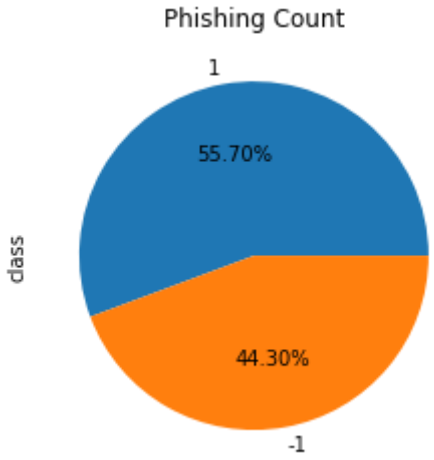
```
plt.show()
```




```
#pairplot for particular features

df = data[['PrefixSuffix-', 'SubDomains',
'HTTPS','AnchorURL','WebsiteTraffic','class']]
sns.pairplot(data = df,hue="class",corner=True);
```




```
# Phishing Count in pie chart

data['class'].value_counts().plot(kind='pie',autopct='%1.2f%%')
plt.title("Phishing Count")
plt.show()
```



3. Splitting the Data

The dataset is split into **training data** (to teach the model) and **testing data** (to evaluate its performance). This ensures the model doesn't just memorize but actually learns general rules.

```
# Splitting the dataset into dependant and independant fetature

X = data.drop(["class"],axis =1)
y = data["class"]

# 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 = 42)
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

5. Model Building & Training

Supervised machine learning is one of the most widely used and effective approaches in AI. It is applied when we want to predict a specific outcome (or label) based on a set of input features, and we already have examples of feature–label pairs to learn from. Using these examples, we train a machine learning model that can then make predictions on new, unseen data.

There are two main types of supervised learning problems: **classification** and **regression**. In classification, the goal is to predict discrete categories (like "spam" or "not spam"), while in regression, the goal is to predict continuous values (like house prices or temperatures).

In this project, our dataset is treated as a **regression problem**, since the target value (suicide rate) is a continuous number rather than a fixed category.

To train and evaluate the dataset, we considered the following supervised learning models:

1. Logistic Regression
2. k-Nearest Neighbors (k-NN)
3. Support Vector Classifier (SVC)
4. Naive Bayes
5. Decision Tree
6. Random Forest
7. Gradient Boosting
8. CatBoost
9. XGBoost
10. Multilayer Perceptrons (Neural Networks)

To measure how well these models perform, we used **Accuracy** and **F1 Score** as evaluation metrics.

```
# Creating holders to store the model performance results
ML_Model = []
accuracy = []
f1_score = []
recall = []
precision = []

#function to call for storing the results
def storeResults(model, a,b,c,d):
  ML_Model.append(model)
  accuracy.append(round(a, 3))
  f1_score.append(round(b, 3))
  recall.append(round(c, 3))
  precision.append(round(d, 3))
```

5.1 Logistic Regression

Logistic Regression is a machine learning algorithm used when the outcome we want to predict belongs to a **category** (for example: "phishing" vs "legitimate", or "yes" vs "no"). Unlike Linear Regression, which predicts continuous numerical values, Logistic Regression is designed specifically for **classification problems**.

In simple terms:

- **Linear Regression** → predicts numbers (like prices, scores, or temperatures).
- **Logistic Regression** → predicts categories (like spam/not spam, true/false, etc.).

Although the name includes "regression," Logistic Regression is actually a **classification algorithm**.

```
# Linear regression model
from sklearn.linear_model import LogisticRegression
#from sklearn.pipeline import Pipeline

# instantiate the model
log = LogisticRegression()

# fit the model
log.fit(X_train,y_train)
```

```
#predicting the target value from the model for the samples

y_train_log = log.predict(X_train)
y_test_log = log.predict(X_test)
```

```
#computing the accuracy, f1_score, Recall, precision of the model performance

acc_train_log = metrics.accuracy_score(y_train,y_train_log)
acc_test_log = metrics.accuracy_score(y_test,y_test_log)
print("Logistic Regression : Accuracy on training Data:
{:.3f}".format(acc_train_log))
print("Logistic Regression : Accuracy on test Data:
{:.3f}".format(acc_test_log))
print()

f1_score_train_log = metrics.f1_score(y_train,y_train_log)
f1_score_test_log = metrics.f1_score(y_test,y_test_log)
print("Logistic Regression : f1_score on training Data:
{:.3f}".format(f1_score_train_log))
print("Logistic Regression : f1_score on test Data:
{:.3f}".format(f1_score_test_log))
print()

recall_score_train_log = metrics.recall_score(y_train,y_train_log)
recall_score_test_log = metrics.recall_score(y_test,y_test_log)
print("Logistic Regression : Recall on training Data:
{:.3f}".format(recall_score_train_log))
print("Logistic Regression : Recall on test Data:
{:.3f}".format(recall_score_test_log))
print()

precision_score_train_log = metrics.precision_score(y_train,y_train_log)
precision_score_test_log = metrics.precision_score(y_test,y_test_log)
print("Logistic Regression : precision on training Data:
{:.3f}".format(precision_score_train_log))
print("Logistic Regression : precision on test Data:
{:.3f}".format(precision_score_test_log))

#computing the classification report of the model
print(metrics.classification_report(y_test, y_test_log))
```

	precision	recall	f1-score	support
-1	0.94	0.91	0.92	976
1	0.93	0.95	0.94	1235
accuracy			0.93	2211
macro avg	0.93	0.93	0.93	2211
weighted avg	0.93	0.93	0.93	2211

```
#storing the results. The below mentioned order of parameter passing is
important.

storeResults('Logistic Regression',acc_test_log,f1_score_test_log,
             recall_score_train_log,precision_score_train_log)
```

5.2 K-Nearest Neighbors (KNN) Classifier

The **K-Nearest Neighbors (KNN)** algorithm is one of the simplest and most intuitive machine learning methods under supervised learning. It works on the idea that **similar things exist close to each other**.

When a new data point needs to be classified, KNN looks at the "K" closest data points (neighbors) from the training set and assigns the new point to the category that most of its neighbors belong to.

In simple words, KNN classifies a new case based on **majority voting** from its nearest neighbors.

```
# K-Nearest Neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

# instantiate the model
knn = KNeighborsClassifier(n_neighbors=1)

# fit the model
knn.fit(X_train,y_train)
```

```
#predicting the target value from the model for the samples
y_train_knn = knn.predict(X_train)
y_test_knn = knn.predict(X_test)
```

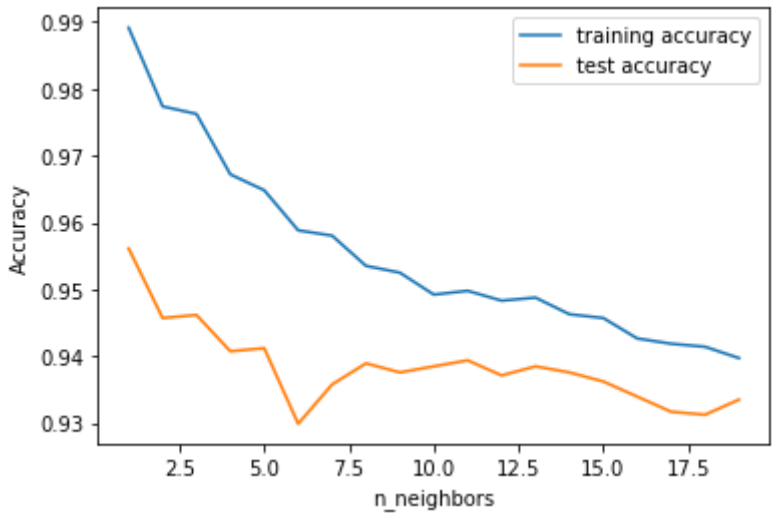
```python
#computing the accuracy,f1_score,Recall,precision of the model performance

acc_train_knn = metrics.accuracy_score(y_train,y_train_knn)
acc_test_knn = metrics.accuracy_score(y_test,y_test_knn)
print("K-Nearest Neighbors : Accuracy on training Data:
{:.3f}".format(acc_train_knn))
print("K-Nearest Neighbors : Accuracy on test Data:
{:.3f}".format(acc_test_knn))
print()

f1_score_train_knn = metrics.f1_score(y_train,y_train_knn)
f1_score_test_knn = metrics.f1_score(y_test,y_test_knn)
print("K-Nearest Neighbors : f1_score on training Data:
{:.3f}".format(f1_score_train_knn))
print("K-Nearest Neighbors : f1_score on test Data:
{:.3f}".format(f1_score_test_knn))
print()

recall_score_train_knn = metrics.recall_score(y_train,y_train_knn)
recall_score_test_knn = metrics.recall_score(y_test,y_test_knn)
print("K-Nearest Neighborsn : Recall on training Data:
{:.3f}".format(recall_score_train_knn))
print("Logistic Regression : Recall on test Data:
{:.3f}".format(recall_score_test_knn))
print()

precision_score_train_knn = metrics.precision_score(y_train,y_train_knn)
precision_score_test_knn = metrics.precision_score(y_test,y_test_knn)
print("K-Nearest Neighbors : precision on training Data:
{:.3f}".format(precision_score_train_knn))
print("K-Nearest Neighbors : precision on test Data:
{:.3f}".format(precision_score_test_knn))
```

```
K-Nearest Neighbors : Accuracy on training Data: 0.989
K-Nearest Neighbors : Accuracy on test Data: 0.956

K-Nearest Neighbors : f1_score on training Data: 0.990
K-Nearest Neighbors : f1_score on test Data: 0.961

K-Nearest Neighborsn : Recall on training Data: 0.991
Logistic Regression : Recall on test Data: 0.962

K-Nearest Neighbors : precision on training Data: 0.989
K-Nearest Neighbors : precision on test Data: 0.960
```

```
#computing the classification report of the model

print(metrics.classification_report(y_test, y_test_knn))
```

```
              precision    recall  f1-score   support

          -1       0.95      0.95      0.95       976
           1       0.96      0.96      0.96      1235

    accuracy                           0.96      2211
   macro avg       0.96      0.96      0.96      2211
weighted avg       0.96      0.96      0.96      2211
```

```
training_accuracy = []
test_accuracy = []
# try max_depth from 1 to 20
depth = range(1,20)
for n in depth:
    knn = KNeighborsClassifier(n_neighbors=n)

    knn.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(knn.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(knn.score(X_test, y_test))


#plotting the training & testing accuracy for n_estimators from 1 to 20
plt.plot(depth, training_accuracy, label="training accuracy")
plt.plot(depth, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_neighbors")
plt.legend();
```



5.3 Support Vector Machine (SVM) Classifier

The **Support Vector Machine (SVM)** is a powerful and widely used supervised learning algorithm that can handle both **classification** and **regression** tasks.

For classification, SVM tries to find the **best possible boundary (also called a hyperplane)** that separates different classes in the data. This boundary is chosen so that the gap (or margin) between the classes is as wide as possible, making the classification more accurate and reliable.

In simpler terms, SVM draws a line (in 2D) or a plane (in higher dimensions) that clearly divides data into categories, so that when new data points appear, we can easily assign them to the correct class.

```
# Support Vector Classifier model
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# defining parameter range
param_grid = {'gamma': [0.1],'kernel': ['rbf','linear']}

svc = GridSearchCV(SVC(), param_grid)

# fitting the model for grid search
svc.fit(X_train, y_train)
```

```
#predicting the target value from the model for the samples
y_train_svc = svc.predict(X_train)
y_test_svc = svc.predict(X_test)
```

```
#computing the accuracy, f1_score, Recall, precision of the model performance
```

```
acc_train_svc = metrics.accuracy_score(y_train,y_train_svc)
acc_test_svc = metrics.accuracy_score(y_test,y_test_svc)
print("Support Vector Machine : Accuracy on training Data:
{:.3f}".format(acc_train_svc))
print("Support Vector Machine : Accuracy on test Data:
{:.3f}".format(acc_test_svc))
print()

f1_score_train_svc = metrics.f1_score(y_train,y_train_svc)
f1_score_test_svc = metrics.f1_score(y_test,y_test_svc)
print("Support Vector Machine : f1_score on training Data:
{:.3f}".format(f1_score_train_svc))
print("Support Vector Machine : f1_score on test Data:
{:.3f}".format(f1_score_test_svc))
print()

recall_score_train_svc = metrics.recall_score(y_train,y_train_svc)
recall_score_test_svc = metrics.recall_score(y_test,y_test_svc)
print("Support Vector Machine : Recall on training Data:
{:.3f}".format(recall_score_train_svc))
print("Support Vector Machine : Recall on test Data:
{:.3f}".format(recall_score_test_svc))
print()

precision_score_train_svc = metrics.precision_score(y_train,y_train_svc)
precision_score_test_svc = metrics.precision_score(y_test,y_test_svc)
print("Support Vector Machine : precision on training Data:
{:.3f}".format(precision_score_train_svc))
print("Support Vector Machine : precision on test Data:
{:.3f}".format(precision_score_test_svc))
```

```
#computing the classification report of the model

print(metrics.classification_report(y_test, y_test_svc))
```

```
              precision    recall  f1-score   support

          -1       0.97      0.94      0.96       976
           1       0.96      0.98      0.97      1235

    accuracy                           0.96      2211
   macro avg       0.97      0.96      0.96      2211
weighted avg       0.96      0.96      0.96      2211
```

5.4 Naïve Bayes Classifier

The **Naïve Bayes** algorithm is a simple yet highly effective supervised learning method used for classification. It is based on **Bayes' Theorem**, which uses probabilities to make predictions about which class a data point belongs to.

The term *"naïve"* comes from the assumption that all features in the dataset are independent of each other — which is rarely true in real-world data, but the algorithm still works surprisingly well.

Naïve Bayes is especially popular for tasks like **text classification** (e.g., spam detection, sentiment analysis) and **image classification**, particularly when working with large, high-dimensional datasets.

Its biggest strength is that it is **fast, efficient, and easy to implement**, making it an excellent choice when quick predictions are needed.

```
# Naive Bayes Classifier Model
from sklearn.naive_bayes import GaussianNB
from sklearn.pipeline import Pipeline

# instantiate the model
nb=  GaussianNB()

# fit the model
nb.fit(X_train,y_train)
```

```
#computing the accuracy, f1_score, Recall, precision of the model performance

acc_train_nb = metrics.accuracy_score(y_train,y_train_nb)
acc_test_nb = metrics.accuracy_score(y_test,y_test_nb)
print("Naive Bayes Classifier : Accuracy on training Data:
{:.3f}".format(acc_train_nb))
print("Naive Bayes Classifier : Accuracy on test Data:
{:.3f}".format(acc_test_nb))
print()

f1_score_train_nb = metrics.f1_score(y_train,y_train_nb)
f1_score_test_nb = metrics.f1_score(y_test,y_test_nb)
print("Naive Bayes Classifier : f1_score on training Data:
{:.3f}".format(f1_score_train_nb))
print("Naive Bayes Classifier : f1_score on test Data:
{:.3f}".format(f1_score_test_nb))
print()

recall_score_train_nb = metrics.recall_score(y_train,y_train_nb)
```

```
recall_score_test_nb = metrics.recall_score(y_test,y_test_nb)
print("Naive Bayes Classifier : Recall on training Data:
{:.3f}".format(recall_score_train_nb))
print("Naive Bayes Classifier : Recall on test Data:
{:.3f}".format(recall_score_test_nb))
print()

precision_score_train_nb = metrics.precision_score(y_train,y_train_nb)
precision_score_test_nb = metrics.precision_score(y_test,y_test_nb)
print("Naive Bayes Classifier : precision on training Data:
{:.3f}".format(precision_score_train_nb))
print("Naive Bayes Classifier : precision on test Data:
{:.3f}".format(precision_score_test_nb))
```

```
#computing the classification report of the model

print(metrics.classification_report(y_test, y_test_svc))
```

```
              precision    recall  f1-score   support

          -1       0.97      0.94      0.96       976
           1       0.96      0.98      0.97      1235

    accuracy                           0.96      2211
   macro avg       0.97      0.96      0.96      2211
weighted avg       0.96      0.96      0.96      2211
```

5.5 Decision Tree Classifier

A **Decision Tree** is a supervised learning algorithm that is commonly used for both **classification** and **regression** problems. It works by splitting the dataset into smaller and smaller subsets based on feature values, forming a tree-like structure of decisions.

Each internal node of the tree represents a **test on a feature** (for example: "Is the URL length greater than 50?"), each branch represents the **outcome of the test**, and each leaf node represents the **final decision or class label**.

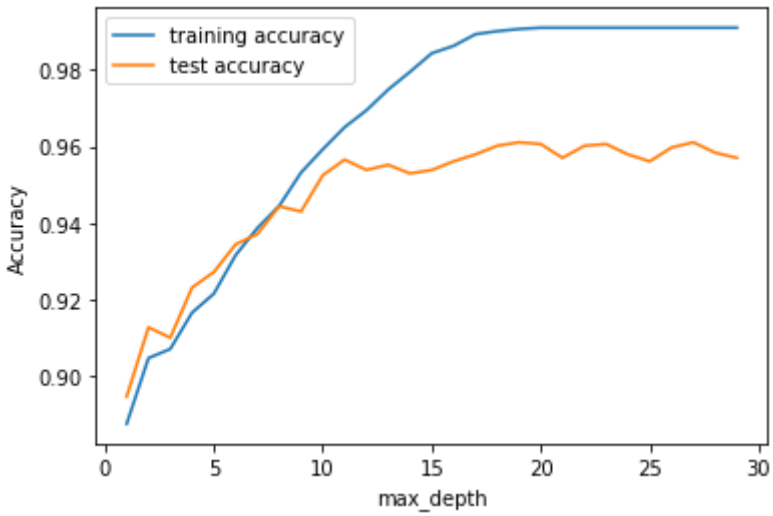
Decision Trees are easy to **understand, visualize, and interpret**, which makes them very popular. However, they can sometimes become overly complex and **overfit** the data unless controlled with techniques like pruning or depth limitation.

```
# Decision Tree Classifier model
from sklearn.tree import DecisionTreeClassifier

# instantiate the model
tree = DecisionTreeClassifier(max_depth=30)

# fit the model
tree.fit(X_train, y_train)
```

```
#computing the classification report of the model

print(metrics.classification_report(y_test, y_test_tree))
```

```
              precision    recall  f1-score   support

          -1       0.96      0.96      0.96       976
           1       0.97      0.96      0.97      1235

    accuracy                           0.96      2211
   macro avg       0.96      0.96      0.96      2211
weighted avg       0.96      0.96      0.96      2211
```

```
training_accuracy = []
test_accuracy = []
# try max_depth from 1 to 30
depth = range(1,30)
for n in depth:
    tree_test = DecisionTreeClassifier(max_depth=n)

    tree_test.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(tree_test.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(tree_test.score(X_test, y_test))


#plotting the training & testing accuracy for max_depth from 1 to 30
plt.plot(depth, training_accuracy, label="training accuracy")
plt.plot(depth, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("max_depth")
plt.legend();
```



5.6 Random Forest Classifier

The **Random Forest** algorithm is an advanced version of Decision Trees that improves both accuracy and stability. Instead of relying on a single decision tree, Random Forest builds **multiple trees** on different random subsets of the data and then combines their results (through majority voting in classification or averaging in regression).

This "ensemble" approach makes Random Forest more **robust and reliable**, reducing the chances of overfitting that a single decision tree might suffer from.

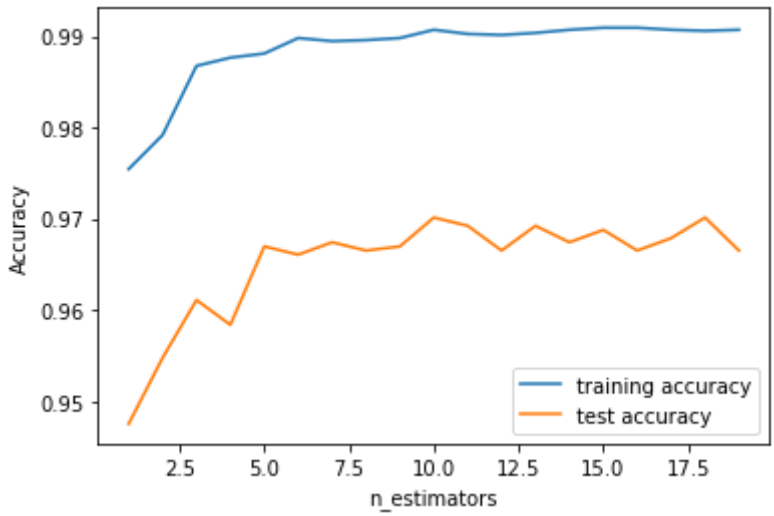
In simple words, Random Forest is like asking the opinion of many decision trees and then choosing the most common answer, which usually leads to better predictions.

```
# Random Forest Classifier Model
from sklearn.ensemble import RandomForestClassifier

# instantiate the model
forest = RandomForestClassifier(n_estimators=10)

# fit the model
forest.fit(X_train,y_train)
```

```
#computing the classification report of the model

print(metrics.classification_report(y_test, y_test_forest))
```

	precision	recall	f1-score	support
-1	0.96	0.96	0.96	976
1	0.97	0.97	0.97	1235
accuracy			0.97	2211
macro avg	0.97	0.97	0.97	2211
eighted avg	0.97	0.97	0.97	2211

```
training_accuracy = []
test_accuracy = []
# try max_depth from 1 to 20
depth = range(1,20)
for n in depth:
    forest_test =  RandomForestClassifier(n_estimators=n)

    forest_test.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(forest_test.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(forest_test.score(X_test, y_test))


#plotting the training & testing accuracy for n_estimators from 1 to 20
plt.figure(figsize=None)
plt.plot(depth, training_accuracy, label="training accuracy")
plt.plot(depth, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_estimators")
plt.legend();
```



5.7 Gradient Boosting Classifier

Gradient Boosting is a powerful machine learning technique that builds models in a **step-by-step (sequential) manner**. Instead of creating all trees independently like in Random Forest, Gradient Boosting builds one tree at a time, where each new tree tries to **fix the errors made by the previous ones**.

It works by combining multiple "weak learners" (usually small decision trees) into a **strong predictive model**. Over time, the algorithm keeps improving by focusing more on the difficult-to-predict cases.

Gradient Boosting is known for delivering **high accuracy** and is widely used in competitions and real-world applications, but it can be **computationally expensive** and requires careful tuning to avoid overfitting.

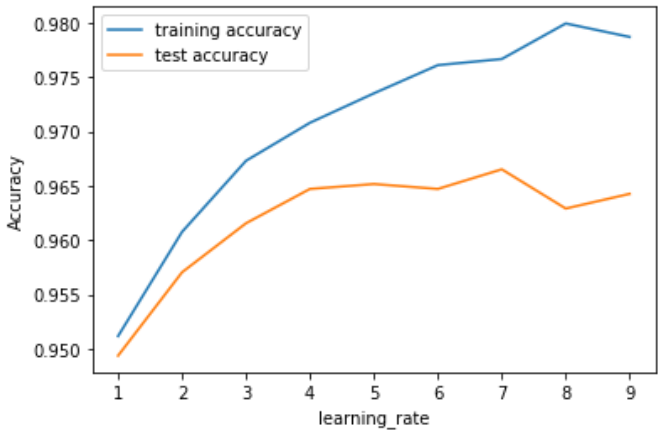
```
# Gradient Boosting Classifier Model
from sklearn.ensemble import GradientBoostingClassifier

# instantiate the model
gbc = GradientBoostingClassifier(max_depth=4,learning_rate=0.7)

# fit the model
gbc.fit(X_train,y_train)
```

```
#computing the classification report of the model

print(metrics.classification_report(y_test, y_test_gbc))
```

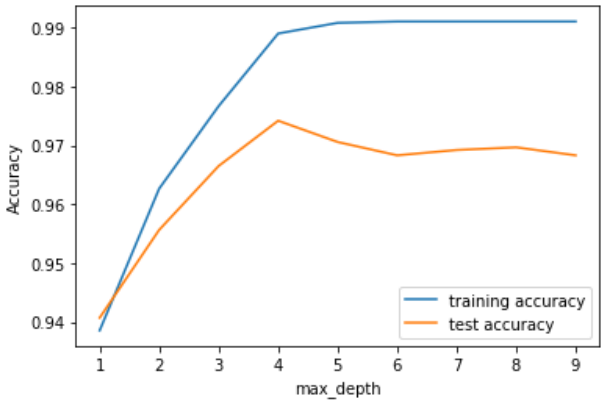
	precision	recall	f1-score	support
-1	0.99	0.96	0.97	976
1	0.97	0.99	0.98	1235
accuracy			0.97	2211
macro avg	0.98	0.97	0.97	2211
weighted avg	0.97	0.97	0.97	2211

```
training_accuracy = []
test_accuracy = []
# try learning_rate from 0.1 to 0.9
depth = range(1,10)
for n in depth:
    forest_test =  GradientBoostingClassifier(learning_rate = n*0.1)

    forest_test.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(forest_test.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(forest_test.score(X_test, y_test))


#plotting the training & testing accuracy for n_estimators from 1 to 50
plt.figure(figsize=None)
plt.plot(depth, training_accuracy, label="training accuracy")
plt.plot(depth, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
```

```
plt.xlabel("learning_rate")
plt.legend();
```




```
training_accuracy = []
test_accuracy = []
# try learning_rate from 0.1 to 0.9
depth = range(1,10,1)
for n in depth:
    forest_test =  GradientBoostingClassifier(max_depth=n,learning_rate = 0.7)

    forest_test.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(forest_test.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(forest_test.score(X_test, y_test))


#plotting the training & testing accuracy for n_estimators from 1 to 50
plt.figure(figsize=None)
plt.plot(depth, training_accuracy, label="training accuracy")
plt.plot(depth, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("max_depth")
plt.legend();
```



5.8 CatBoost Classifier

CatBoost is a high-performance gradient boosting algorithm developed by Yandex. It is especially designed to handle **categorical features** (like country names, product IDs, or categories) efficiently, without needing a lot of preprocessing or manual feature engineering.

Just like other boosting algorithms, CatBoost builds models step by step, where each new tree corrects the mistakes of the previous ones. What makes it stand out is that it is **fast, accurate, and user-friendly**, while also reducing the chances of overfitting.

CatBoost is widely used in real-world applications because it often works **out of the box** with minimal tuning, and it performs really well on both classification and regression problems.

```
#  catboost Classifier Model
from catboost import CatBoostClassifier

# instantiate the model
cat = CatBoostClassifier(learning_rate  = 0.1)

# fit the model
cat.fit(X_train,y_train)
```

```
0:      learn: 0.5487232        total: 235ms    remaining: 3m 54s
1:      learn: 0.4349357        total: 260ms    remaining: 2m 9s
2:      learn: 0.3609236        total: 275ms    remaining: 1m 31s
3:      learn: 0.3050829        total: 290ms    remaining: 1m 12s
4:      learn: 0.2766620        total: 303ms    remaining: 1m
5:      learn: 0.2475476        total: 318ms    remaining: 52.6s
6:      learn: 0.2286637        total: 333ms    remaining: 47.2s
7:      learn: 0.2138754        total: 347ms    remaining: 43s
8:      learn: 0.2013643        total: 356ms    remaining: 39.2s
9:      learn: 0.1896378        total: 365ms    remaining: 36.2s
10:     learn: 0.1819539        total: 375ms    remaining: 33.7s
11:     learn: 0.1767867        total: 382ms    remaining: 31.5s
12:     learn: 0.1727735        total: 388ms    remaining: 29.5s
13:     learn: 0.1682578        total: 394ms    remaining: 27.8s
14:     learn: 0.1641759        total: 400ms    remaining: 26.3s
15:     learn: 0.1614218        total: 407ms    remaining: 25s
16:     learn: 0.1558968        total: 413ms    remaining: 23.9s
17:     learn: 0.1535881        total: 420ms    remaining: 22.9s
18:     learn: 0.1514228        total: 427ms    remaining: 22s
19:     learn: 0.1482580        total: 433ms    remaining: 21.2s
20:     learn: 0.1452536        total: 439ms    remaining: 20.5s
21:     learn: 0.1426992        total: 445ms    remaining: 19.8s
22:     learn: 0.1405068        total: 450ms    remaining: 19.1s
23:     learn: 0.1381617        total: 455ms    remaining: 18.5s
24:     learn: 0.1363558        total: 461ms    remaining: 18s
...
177:    learn: 0.0530595        total: 1.29s    remaining: 5.97s
178:    learn: 0.0529470        total: 1.3s     remaining: 5.96s
179:    learn: 0.0527691        total: 1.3s     remaining: 5.95s
180:    learn: 0.0526404        total: 1.31s    remaining: 5.93s
```

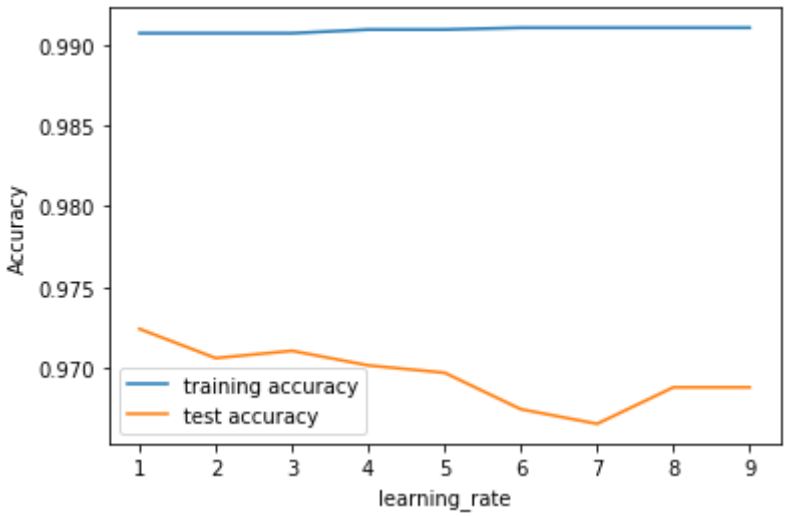
```
#predicting the target value from the model for the samples
y_train_cat = cat.predict(X_train)
y_test_cat = cat.predict(X_test)
```

```
#computing the classification report of the model

print(metrics.classification_report(y_test, y_test_cat))
```

	precision	recall	f1-score	support
-1	0.98	0.96	0.97	976
1	0.97	0.98	0.98	1235
accuracy			0.97	2211
macro avg	0.97	0.97	0.97	2211
weighted avg	0.97	0.97	0.97	2211

```
training_accuracy = []
test_accuracy = []
# try learning_rate from 0.1 to 0.9
depth = range(1,10)
for n in depth:
    forest_test =  CatBoostClassifier(learning_rate = n*0.1)

    forest_test.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(forest_test.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(forest_test.score(X_test, y_test))
```

```
#plotting the training & testing accuracy for n_estimators from 1 to 50
plt.figure(figsize=None)
plt.plot(depth, training_accuracy, label="training accuracy")
plt.plot(depth, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("learning_rate")
plt.legend();
```



5.9 XGBoost Classifier

XGBoost (Extreme Gradient Boosting) is one of the most popular and powerful machine learning algorithms. It is an optimized version of Gradient Boosting that is designed to be **fast, efficient, and highly accurate**.

XGBoost builds models sequentially, where each new tree corrects the mistakes of the previous ones, but it adds clever techniques like **regularization** (to prevent overfitting) and efficient handling of missing values. These improvements make it one of the top choices in both research and industry.

It has become famous in machine learning competitions (like Kaggle) because it consistently delivers **state-of-the-art performance** while being scalable to very large datasets.

```
#  XGBoost Classifier Model
from xgboost import XGBClassifier

# instantiate the model
xgb = XGBClassifier()

# fit the model
xgb.fit(X_train,y_train)
```

```
#predicting the target value from the model for the samples
y_train_xgb = xgb.predict(X_train)
y_test_xgb = xgb.predict(X_test)
#storing the results. The below mentioned order of parameter passing is
important.

storeResults('XGBoost Classifier',acc_test_xgb,f1_score_test_xgb,
             recall_score_train_xgb,precision_score_train_xgb)
```

5.10 Multilayer Perceptrons (Neural Networks)

A **Multilayer Perceptron (MLP)** is a type of **artificial neural network** made up of layers of interconnected nodes (called neurons). These neurons are arranged in an **input layer**, one or more **hidden layers**, and an **output layer**.

Each neuron processes the input it receives, applies a mathematical function (called an activation function), and passes the result forward through the network. By adjusting the weights of these connections during training, the MLP gradually learns complex patterns in the data.

MLPs are powerful because they can capture **non-linear relationships** that simpler models might miss. They are widely used in various domains like image recognition, natural language processing, and predictive analytics.

However, they require more data and computational power compared to traditional algorithms, and they may take longer to train.

```
# Multi-layer Perceptron Classifier Model
from sklearn.neural_network import MLPClassifier

# instantiate the model
mlp = MLPClassifier()
#mlp = GridSearchCV(mlpc, parameter_space)

# fit the model
mlp.fit(X_train,y_train)
```

```
#predicting the target value from the model for the samples
y_train_mlp = mlp.predict(X_train)
y_test_mlp = mlp.predict(X_test)
```

6. Comparing Models
Each trained model is evaluated using metrics like **accuracy, precision, recall, and F1-score**. Comparing them shows which algorithm works best for this problem.

```
#creating dataframe
result = pd.DataFrame({ 'ML Model' : ML_Model,
                        'Accuracy' : accuracy,
                        'f1_score' : f1_score,
                        'Recall'   : recall,
                        'Precision': precision,
                      })
# dispalying total result
result
```

	ML Model	Accuracy	f1_score	Recall	Precision
0	Logistic Regression	0.934	0.941	0.943	0.927
1	K-Nearest Neighbors	0.956	0.961	0.991	0.989
2	Support Vector Machine	0.964	0.968	0.980	0.965
3	Naive Bayes Classifier	0.605	0.454	0.292	0.997
4	Decision Tree	0.961	0.965	0.991	0.993
5	Random Forest	0.967	0.970	0.992	0.991
6	Gradient Boosting Classifier	0.974	0.977	0.994	0.986
7	CatBoost Classifier	0.972	0.975	0.994	0.989
8	XGBoost Classifier	0.969	0.973	0.993	0.984
9	Multi-layer Perceptron	0.971	0.974	0.992	0.985

```
# dispalying total result
sorted_result
```

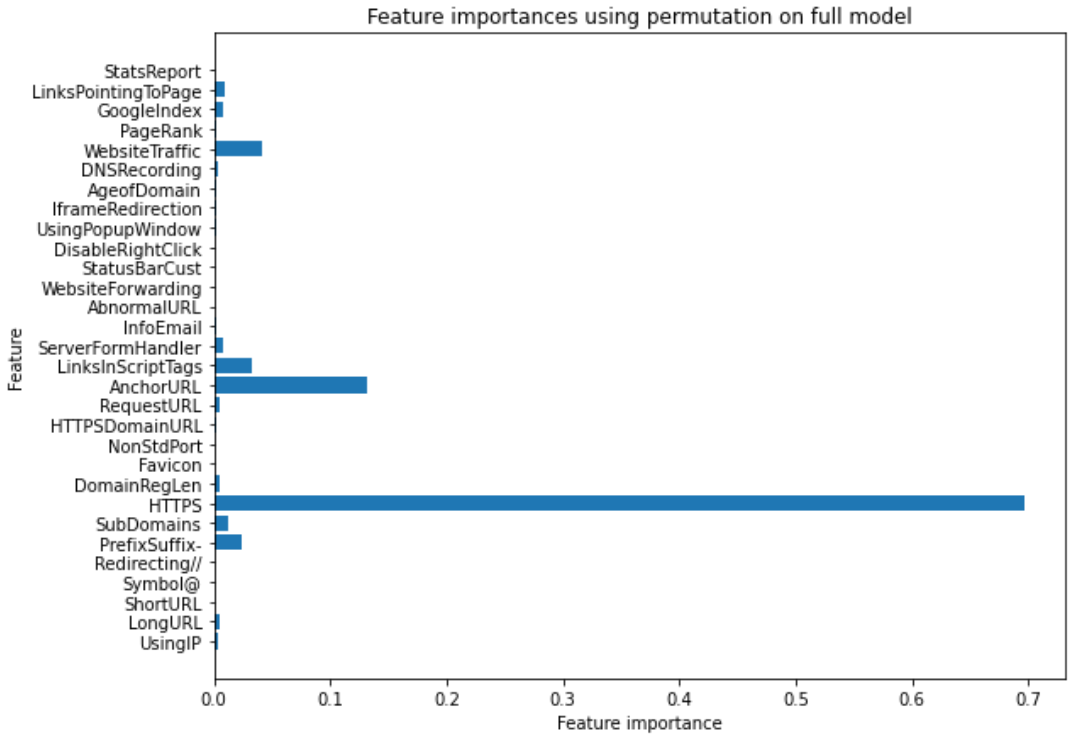
	ML Model	Accuracy	f1_score	Recall	Precision
0	Gradient Boosting Classifier	0.974	0.977	0.994	0.986
1	CatBoost Classifier	0.972	0.975	0.994	0.989
2	Multi-layer Perceptron	0.971	0.974	0.992	0.985
3	XGBoost Classifier	0.969	0.973	0.993	0.984
4	Random Forest	0.967	0.970	0.992	0.991
5	Support Vector Machine	0.964	0.968	0.980	0.965
6	Decision Tree	0.961	0.965	0.991	0.993
7	K-Nearest Neighbors	0.956	0.961	0.991	0.989
8	Logistic Regression	0.934	0.941	0.943	0.927
9	Naive Bayes Classifier	0.605	0.454	0.292	0.997

Storing Best Model:

```
#  XGBoost Classifier Model
from xgboost import XGBClassifier

# instantiate the model
gbc = GradientBoostingClassifier(max_depth=4,learning_rate=0.7)

# fit the model
gbc.fit(X_train,y_train)
```

```
import pickle

# dump information to that file
pickle.dump(gbc, open('pickle/model.pkl', 'wb'))
```

```
#checking the feature improtance in the model
plt.figure(figsize=(9,7))
n_features = X_train.shape[1]
plt.barh(range(n_features), gbc.feature_importances_, align='center')
plt.yticks(np.arange(n_features), X_train.columns)
plt.title("Feature importances using permutation on full model")
plt.xlabel("Feature importance")
plt.ylabel("Feature")
plt.show()
```



Conclusion

1. The key takeaway from this project is the opportunity to explore a variety of machine learning models, perform **exploratory data analysis (EDA)** on a phishing dataset, and gain a deeper understanding of the features that influence phishing detection.
2. Building this notebook not only enhanced my knowledge of how different features impact phishing detection, but also gave me hands-on experience with **model tuning** and understanding how fine-tuning parameters can significantly affect model performance.
3. From the analysis, it was observed that certain features such as **"HTTPS," "AnchorURL," and "WebsiteTraffic"** play a particularly important role in determining whether a URL is safe or phishing.
4. Among all models tested, the **Gradient Boosting Classifier** delivered the best performance, correctly classifying URLs with an accuracy of around **97.4%**. This high accuracy makes it a strong candidate for reducing the risk of malicious websites and attachments.