

PhiUSIIL: A Machine Learning-Based Phishing URL Classifier

DSML Capstone Project

Problem Description :

With the increasing reliance on the internet for financial transactions, personal communications, and business operations, phishing attacks have emerged as a significant cybersecurity threat. Phishing websites impersonate legitimate entities to deceive users into revealing sensitive information, such as passwords, credit card details, and personal data.

Detecting phishing websites accurately and efficiently is a crucial challenge in cybersecurity. Traditional methods, such as blacklists, fail to detect new phishing sites in real time. Therefore, machine learning-based approaches using URL and webpage source code features have gained prominence in identifying phishing threats proactively.

In this project, we utilize the PhiUSIIL Phishing URL Dataset, which contains 134,850 legitimate URLs and 100,945 phishing URLs, to develop a robust phishing detection model. The dataset includes features extracted from the webpage source code and URL structure, such as CharContinuationRate, URLTitleMatchScore, URLCharProb, and TLDLegitimateProb, which are derived from existing URL-based attributes.

Objective :

To analyze these features and implement machine learning techniques to classify URLs as legitimate or phishing, improving the accuracy and efficiency of phishing detection systems. Through this project, we aim to contribute to the development of automated security solutions that help protect users from online fraud and cyber threats.

DataSet :

(<https://archive.ics.uci.edu/dataset/967/phiusiil+phishing+url+dataset>)

Understand the Dataset

```
In [6]: import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.feature_selection import SelectKBest
import xgboost as xgb
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import warnings
warnings.filterwarnings('ignore')
```

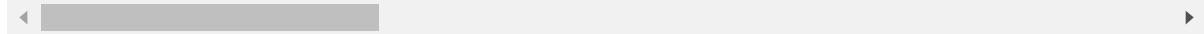
Loading

In [8]: `data=pd.read_csv("PhiUSIIL_Phishing_URL_Dataset.csv")`

In [9]: `data.head()`

	FILENAME	URL	URLLength	Domain	Do
0	521848.txt	https://www.southbankmosaics.com	31	www.southbankmosaics.com	
1	31372.txt	https://www.uni-mainz.de	23	www.uni-mainz.de	
2	597387.txt	https://www.voicefmradio.co.uk	29	www.voicefmradio.co.uk	
3	554095.txt	https://www.sfnmjurnal.com	26	www.sfnmjurnal.com	
4	151578.txt	https://www.rewindingargentina.org	33	www.rewindingargentina.org	

5 rows × 56 columns



In [10]: `data.shape`

Out[10]: `(235795, 56)`

In [11]: `data.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 235795 entries, 0 to 235794
Data columns (total 56 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   FILENAME        235795 non-null   object  
 1   URL             235795 non-null   object  
 2   URLLength       235795 non-null   int64  
 3   Domain          235795 non-null   object  
 4   DomainLength    235795 non-null   int64  
 5   IsDomainIP     235795 non-null   int64  
 6   TLD             235795 non-null   object  
 7   URLSimilarityIndex  235795 non-null   float64 
 8   CharContinuationRate  235795 non-null   float64 
 9   TLDLegitimateProb  235795 non-null   float64 
 10  URLCharProb    235795 non-null   float64 
 11  TLDLength       235795 non-null   int64  
 12  NoOfSubDomain   235795 non-null   int64  
 13  Has0bfuscation  235795 non-null   int64  
 14  No0f0bfuscatedChar  235795 non-null   int64  
 15  ObfuscationRatio  235795 non-null   float64 
 16  No0fLettersInURL  235795 non-null   int64  
 17  LetterRatioInURL  235795 non-null   float64 
 18  No0fDigitsInURL  235795 non-null   int64  
 19  DigitRatioInURL  235795 non-null   float64 
 20  No0fEqualsInURL  235795 non-null   int64  
 21  No0fQMarkInURL  235795 non-null   int64  
 22  No0fAmpersandInURL  235795 non-null   int64  
 23  No0fOtherSpecialCharsInURL  235795 non-null   int64  
 24  SpacialCharRatioInURL  235795 non-null   float64 
 25  IsHTTPS          235795 non-null   int64  
 26  Line0fCode        235795 non-null   int64  
 27  LargestLineLength  235795 non-null   int64  
 28  HasTitle          235795 non-null   int64  
 29  Title             235795 non-null   object  
 30  DomainTitleMatchScore  235795 non-null   float64 
 31  URLTitleMatchScore  235795 non-null   float64 
 32  HasFavicon        235795 non-null   int64  
 33  Robots            235795 non-null   int64  
 34  IsResponsive      235795 non-null   int64  
 35  No0fURLRedirect   235795 non-null   int64  
 36  No0fSelfRedirect   235795 non-null   int64  
 37  HasDescription     235795 non-null   int64  
 38  No0fPopup          235795 non-null   int64  
 39  No0fiFrame         235795 non-null   int64  
 40  HasExternalFormSubmit  235795 non-null   int64  
 41  HasSocialNet       235795 non-null   int64  
 42  HasSubmitButton     235795 non-null   int64  
 43  HasHiddenFields     235795 non-null   int64  
 44  HasPasswordField    235795 non-null   int64  
 45  Bank               235795 non-null   int64  
 46  Pay                235795 non-null   int64  
 47  Crypto              235795 non-null   int64  
 48  HasCopyrightInfo    235795 non-null   int64  
 49  No0fImage          235795 non-null   int64  
 50  No0fCSS            235795 non-null   int64
```

```

51  NoOfJS           235795 non-null int64
52  NoOfSelfRef      235795 non-null int64
53  NoOfEmptyRef     235795 non-null int64
54  NoOfExternalRef   235795 non-null int64
55  label             235795 non-null int64
dtypes: float64(10), int64(41), object(5)
memory usage: 100.7+ MB

```

In [12]: `data.describe()`

	URLLength	DomainLength	IsDomainIP	URLSimilarityIndex	CharContinuation
count	235795.000000	235795.000000	235795.000000	235795.000000	235795.000000
mean	34.573095	21.470396	0.002706	78.430778	0.84!
std	41.314153	9.150793	0.051946	28.976055	0.216
min	13.000000	4.000000	0.000000	0.155574	0.000
25%	23.000000	16.000000	0.000000	57.024793	0.680
50%	27.000000	20.000000	0.000000	100.000000	1.000
75%	34.000000	24.000000	0.000000	100.000000	1.000
max	6097.000000	110.000000	1.000000	100.000000	1.000

8 rows × 51 columns

In [13]: `data.groupby("label").describe()`

	label	URLLength										DomainLength	
		count	mean	std	min	25%	50%	75%	max	count	mean		
0	100945.0	45.720293	61.145523	13.0	26.0	34.0	48.0	6097.0	100945.0	24.465144			
1	134850.0	26.228610	4.815612	15.0	23.0	26.0	29.0	57.0	134850.0	19.228610			

2 rows × 400 columns

Preprocessing and EDA

checking for null values

In [16]: `data.isnull().sum()`

```
Out[16]: FILENAME          0
          URL              0
          URLLength         0
          Domain           0
          DomainLength      0
          IsDomainIP        0
          TLD               0
          URLSimilarityIndex 0
          CharContinuationRate 0
          TLDLegitimateProb   0
          URLCharProb        0
          TLDLength          0
          NoOfSubDomain       0
          HasObfuscation      0
          NoOfObfuscatedChar    0
          ObfuscationRatio     0
          NoOfLettersInURL     0
          LetterRatioInURL     0
          NoOfDigitsInURL      0
          DigitRatioInURL      0
          NoOfEqualsInURL       0
          NoOfQMarkInURL       0
          NoOfAmpersandInURL    0
          NoOfOtherSpecialCharsInURL 0
          SpacialCharRatioInURL 0
          IsHTTPS             0
          LineOfCode          0
          LargestLineLength     0
          HasTitle            0
          Title               0
          DomainTitleMatchScore 0
          URLTitleMatchScore    0
          HasFavicon          0
          Robots              0
          IsResponsive         0
          NoOfURLRedirect      0
          NoOfSelfRedirect      0
          HasDescription        0
          NoOfPopup            0
          NoOfFrame            0
          HasExternalFormSubmit   0
          HasSocialNet          0
          HasSubmitButton        0
          HasHiddenFields        0
          HasPasswordField      0
          Bank                 0
          Pay                  0
          Crypto               0
          HasCopyrightInfo      0
          NoOfImage            0
          NoOfCSS              0
          NoOfJS               0
          NoOfSelfRef           0
          NoOfEmptyRef          0
          NoOfExternalRef        0
```

```
label          0  
dtype: int64
```

Checking for duplicates

```
In [18]: data.duplicated().sum()
```

```
Out[18]: 0
```

No null values or duplicates founded!

Check class distribution for finding imbalance

```
In [21]: print(data['label'].value_counts())
```

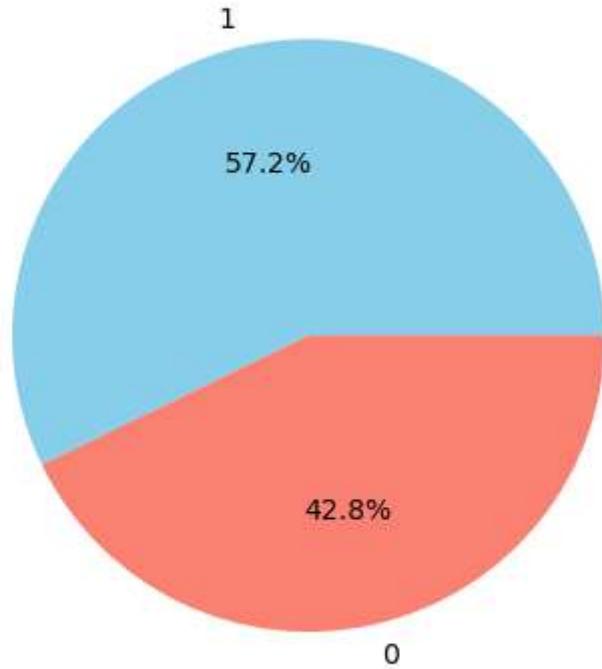
```
label  
1    134850  
0    100945  
Name: count, dtype: int64
```

```
In [22]: #Calculate Class Imbalance Ratio  
maj_cls=data['label'].value_counts().max()  
min_cls=data['label'].value_counts().min()  
ratio=maj_cls/min_cls  
print(f"Class Imbalance Ratio: {ratio:.2f}")
```

```
Class Imbalance Ratio: 1.34
```

```
In [23]: data['label'].value_counts().plot(kind='pie', autopct='%1.1f%%', colors=['skyblue',  
plt.title("Class Distribution")  
plt.ylabel("")  
plt.show()
```

Class Distribution



Having a mild imbalance (ratio : 1.34), which does not affect the model accuracy.

Univariate Analysis

Finding Categorical features for EDA.

```
In [27]: categorical=data.select_dtypes(include='object').columns  
categorical
```

```
Out[27]: Index(['FILENAME', 'URL', 'Domain', 'TLD', 'Title'], dtype='object')
```

Column 'FILENAME' can be ignored.

```
In [29]: #making a copy  
df=data.copy()
```

```
In [30]: df.drop(columns='FILENAME', inplace=True)
```

```
In [31]: updated_categorical=df.select_dtypes(include='object').columns  
updated_categorical
```

```
Out[31]: Index(['URL', 'Domain', 'TLD', 'Title'], dtype='object')
```

Finding numerical features

```
In [33]: numerical=df.select_dtypes(include='number').columns
```

Exclude Binary (0/1) Columns

```
In [35]: # Identify binary columns (only contain 0 and 1)
binary_cols=[col for col in numerical if df[col].nunique()==2]

## Exclude binary columns from numerical analysis
univariate_cols= [col for col in numerical if df[col].nunique()!=2]

print("Numerical columns for univariate analysis:", univariate_cols)
print (f"There are {len(univariate_cols)} numerical features for univariate analysis")
```

Numerical columns for univariate analysis: ['URLLength', 'DomainLength', 'URLSimilarityIndex', 'CharContinuationRate', 'TLDLegitimateProb', 'URLCharProb', 'TLDLength', 'NoOfSubDomain', 'NoOfObfuscatedChar', 'ObfuscationRatio', 'NoOfLettersInURL', 'LetterRatioInURL', 'NoOfDigitsInURL', 'DigitRatioInURL', 'NoOfEqualsInURL', 'NoOfQMarkInURL', 'NoOfAmpersandInURL', 'NoOfOtherSpecialCharsInURL', 'SpacialCharRatioInURL', 'LineOfCode', 'LargestLineLength', 'DomainTitleMatchScore', 'URLTitleMatchScore', 'NoOfPopup', 'NoOfFrame', 'NoOfImage', 'NoOfCSS', 'NoOfJS', 'NoOfSelfRef', 'NoOfEmptyRef', 'NoOfExternalRef']

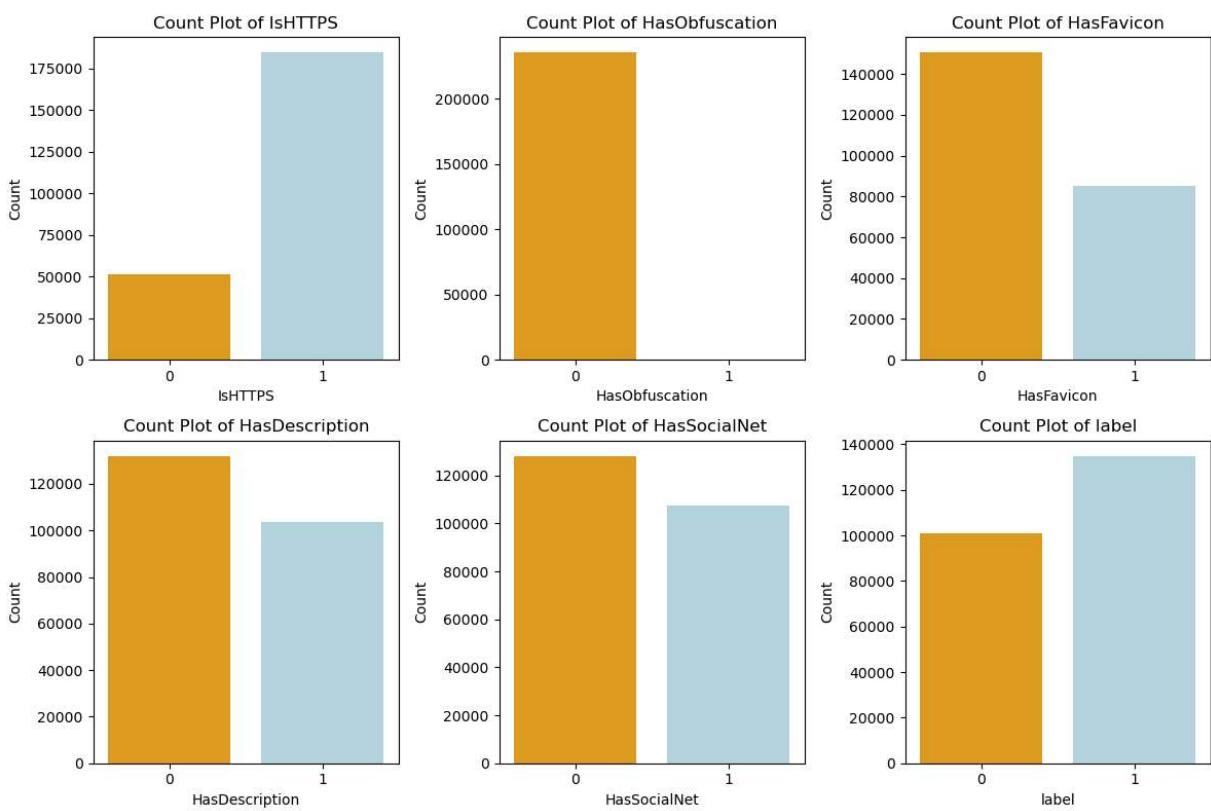
There are 31 numerical features for univariate analysis.

Lets do some Analysis (Dataset before removing skewness)

```
In [37]: # List of relevant binary features
binary_features = ['IsHTTPS', 'HasObfuscation', 'HasFavicon', 'HasDescription', 'Ha

plt.figure(figsize=(12, 8))
for i, col in enumerate(binary_features, 1):
    plt.subplot(2, 3, i) # 2 rows, 3 columns
    sns.countplot(x=df[col], palette=['orange', 'lightblue'])
    plt.title(f"Count Plot of {col}")
    plt.xlabel(col)
    plt.ylabel("Count")

plt.tight_layout()
plt.show()
```

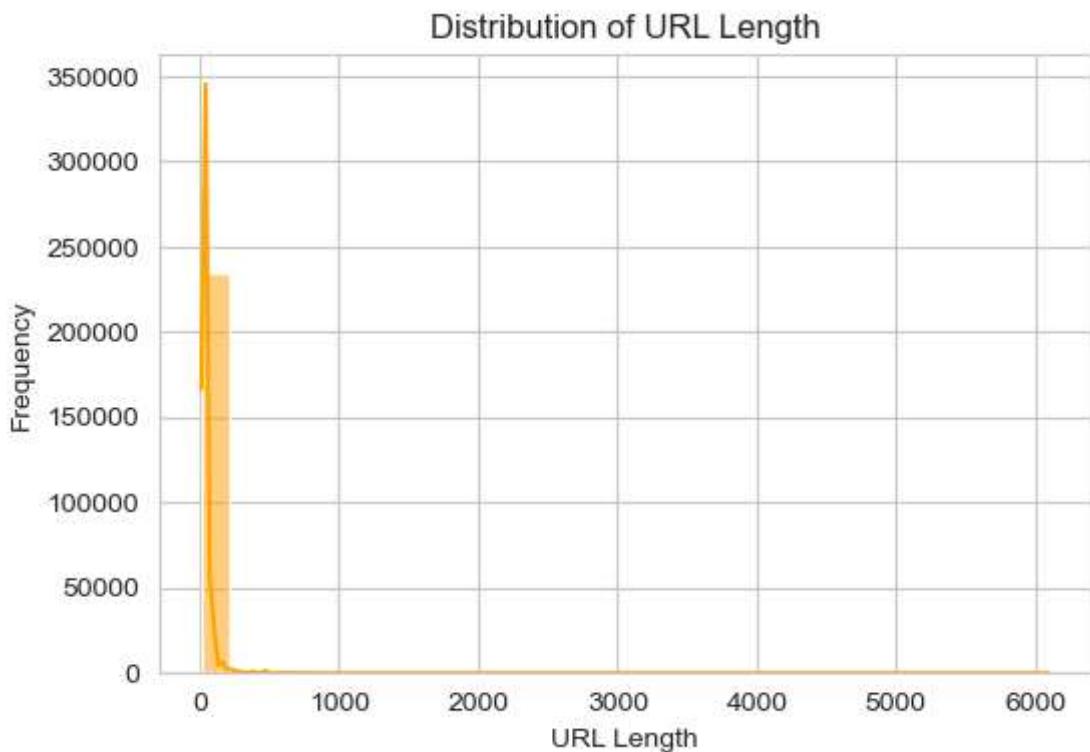


Insights

- IsHTTPS: Most URLs use HTTPS, indicating that HTTPS alone is not a strong indicator of legitimacy.
- HasObfuscation: Very few URLs use obfuscation, suggesting it is not a common technique for phishing.
- HasFavicon: Many phishing URLs lack favicons, making it a potential indicator for phishing detection.
- HasDescription: Many URLs lack a meta description, which could be a potential phishing indicator.
- HasSocialNet: A significant number of URLs do not contain social network links, suggesting that legitimate sites may include them more often.
- Label: The dataset contains more phishing URLs than legitimate ones, indicating an imbalance that may affect model performance.

```
In [39]: # Set plot style
sns.set_style("whitegrid")

# Histogram: URL Length Distribution
plt.figure(figsize=(6, 4))
sns.histplot(df['URLLength'], bins=30, kde=True, color='orange')
plt.title("Distribution of URL Length")
plt.xlabel("URL Length")
plt.ylabel("Frequency")
plt.show()
```



Insights :

The histogram reveals a right-skewed distribution of URL lengths, with most URLs being relatively short, typically under 500 characters. However, the presence of a few extremely long URLs, potentially over 6000 characters, could indicate obfuscation or suspicious activity, often seen in phishing or malicious websites.

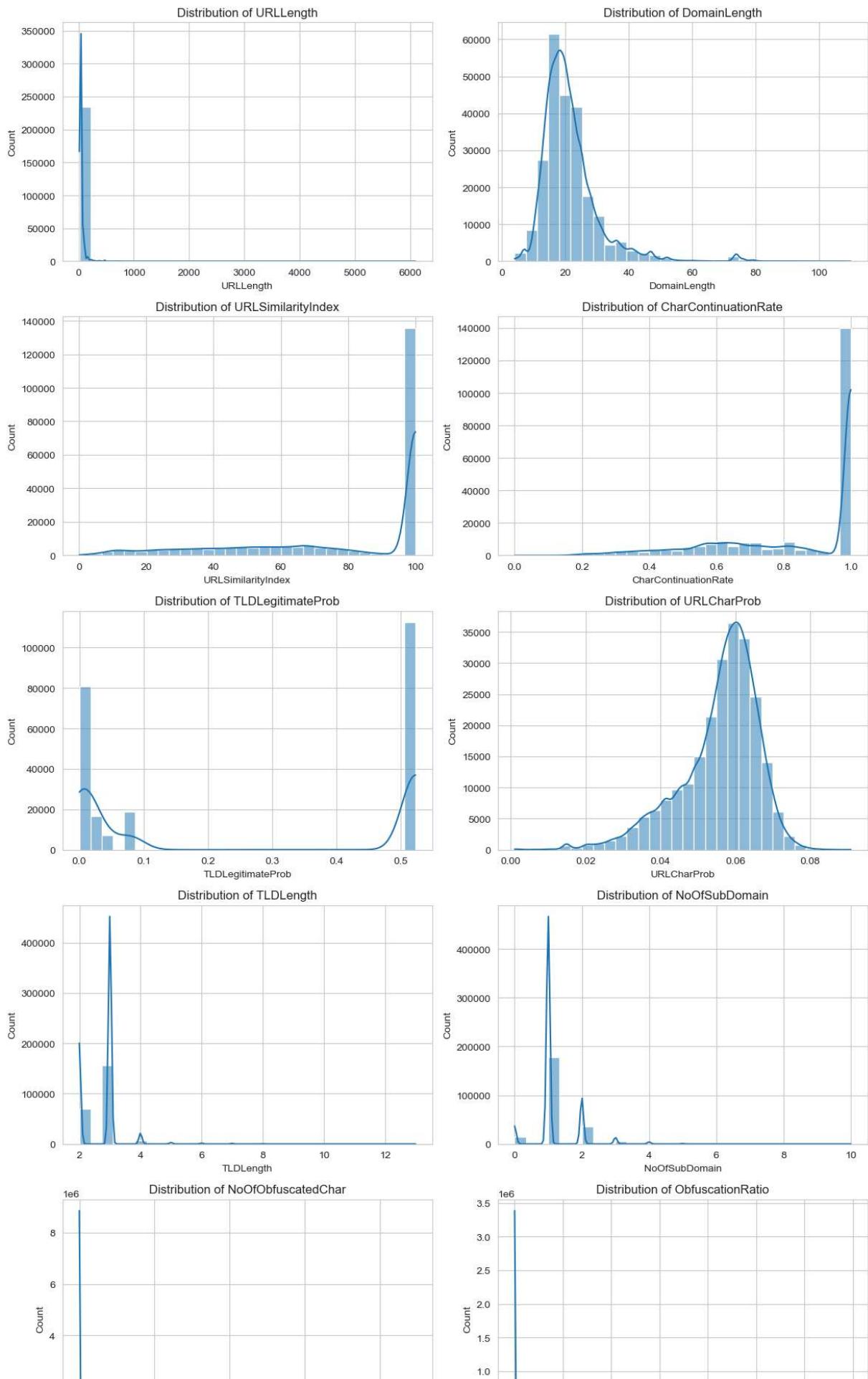
```
In [41]: # Assuming df is your DataFrame and univariate_numerical_cols is your list of columns
num_cols = len(univariate_numerical_cols)
rows = (num_cols + 1) // 2 # Calculate required rows (ceil division for odd number)

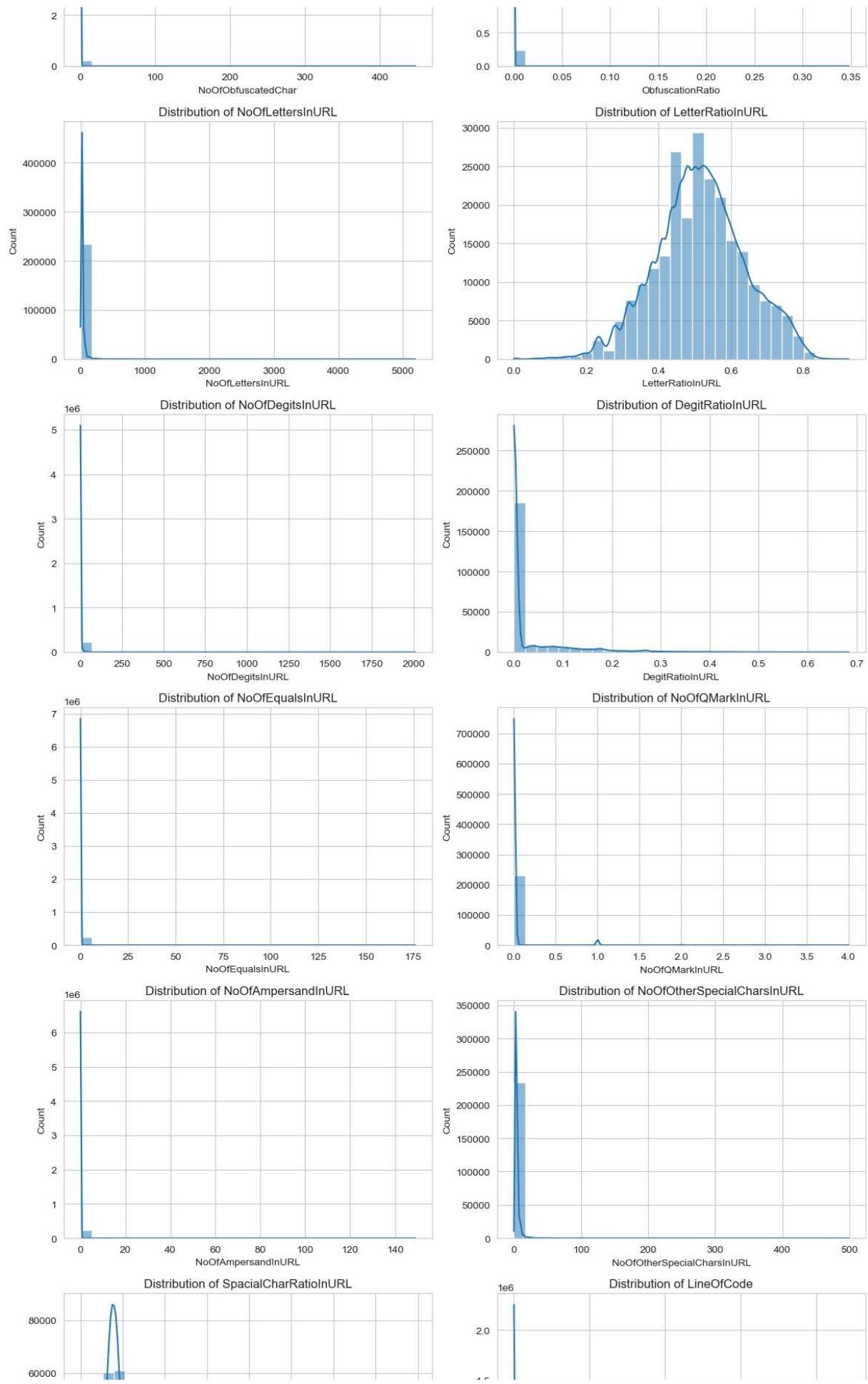
fig, axes = plt.subplots(rows, 2, figsize=(12, 4 * rows)) # Adjust size accordingly
axes = axes.flatten() # Flatten in case of multiple rows

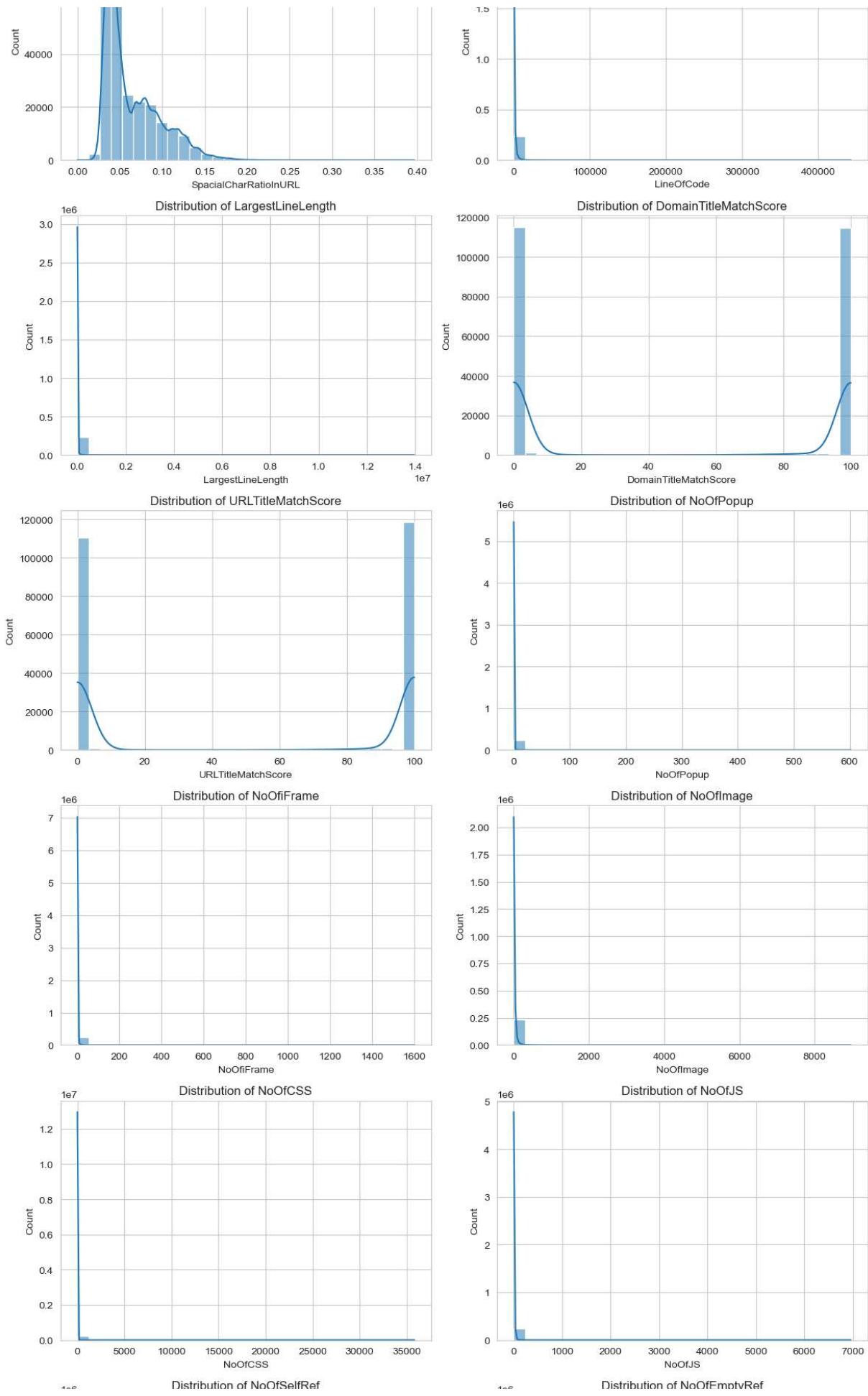
for i, col in enumerate(univariate_numerical_cols):
    sns.histplot(df[col], kde=True, bins=30, ax=axes[i])
    axes[i].set_title(f'Distribution of {col}')

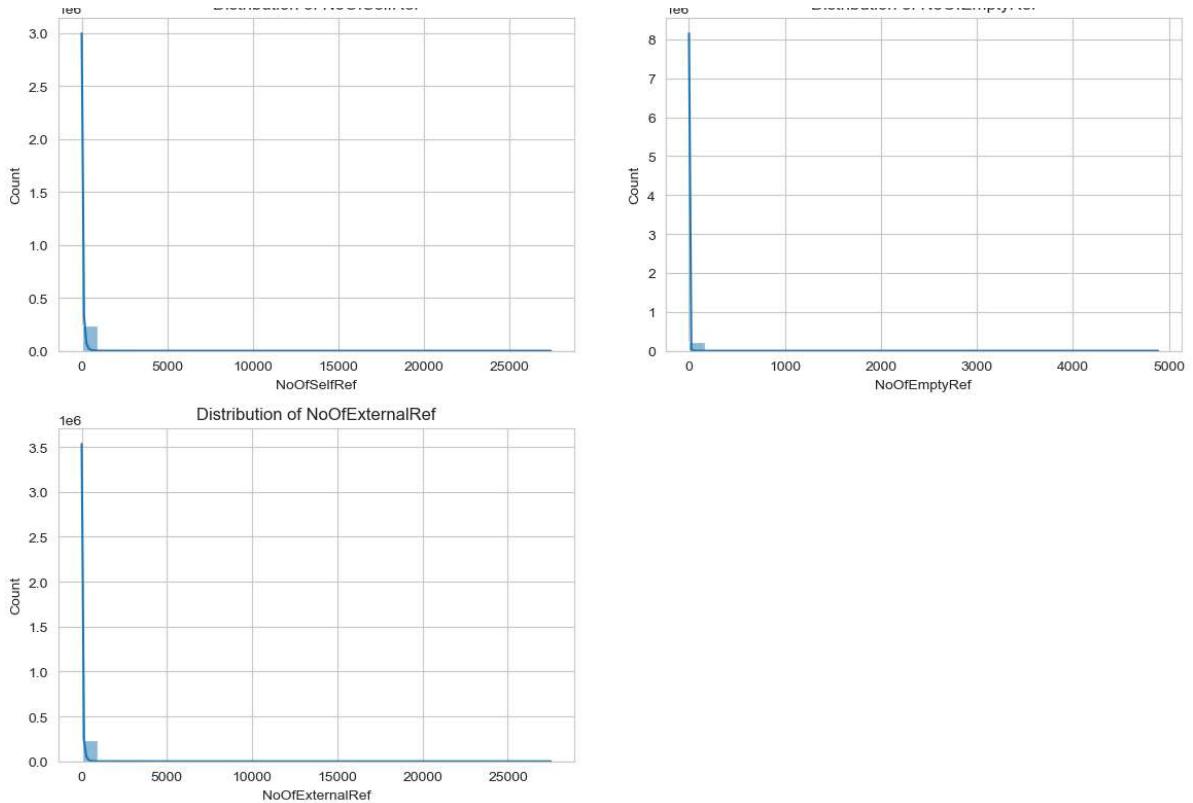
# Hide any unused subplots
for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout() # Adjust Layout for better spacing
plt.show()
```









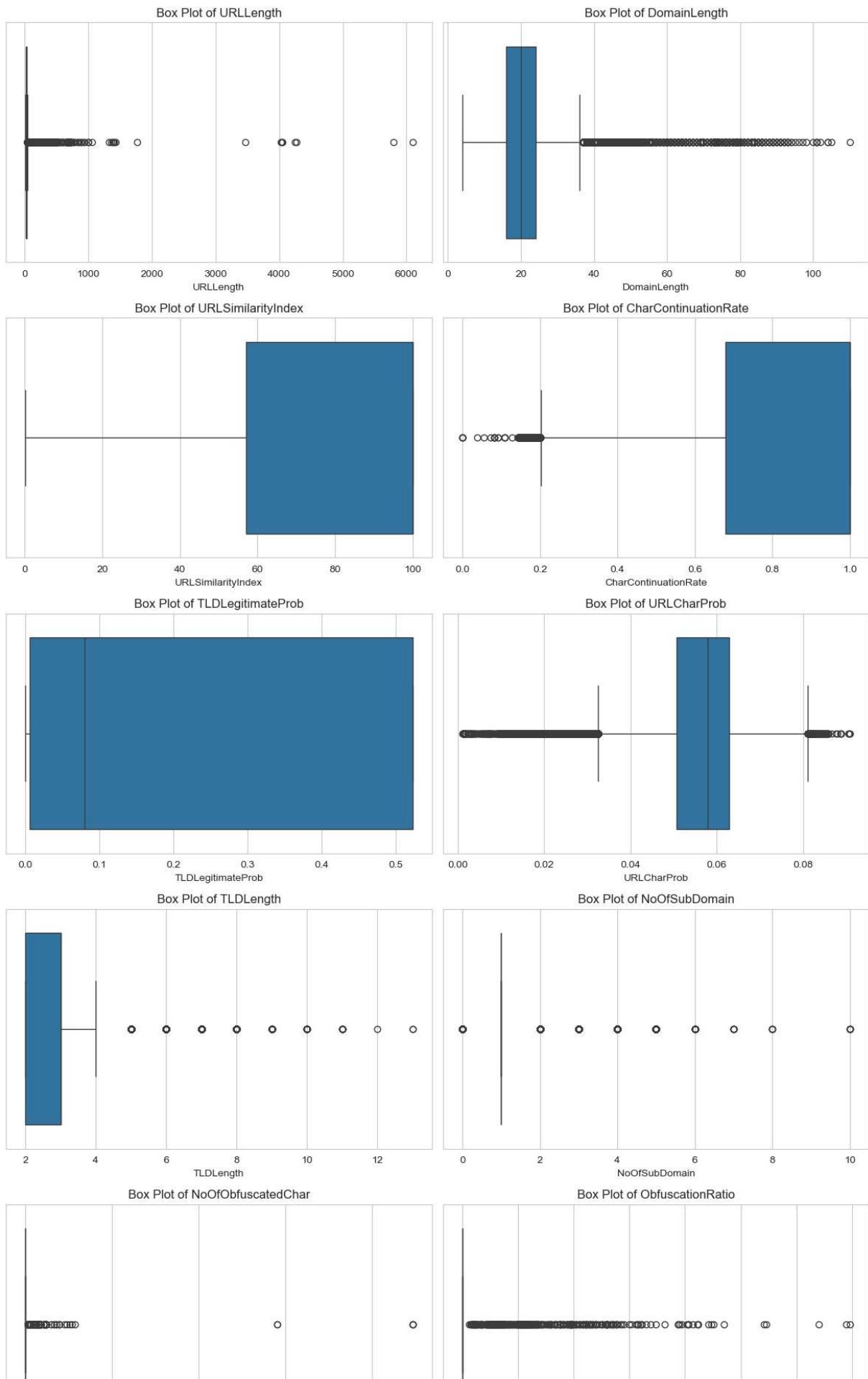
```
In [42]: # Assuming df is your DataFrame and univariate_numerical_cols is your list of columns
num_cols = len(univariate_numerical_cols)
rows = (num_cols + 1) // 2 # Calculate required rows (ceil division for odd number)

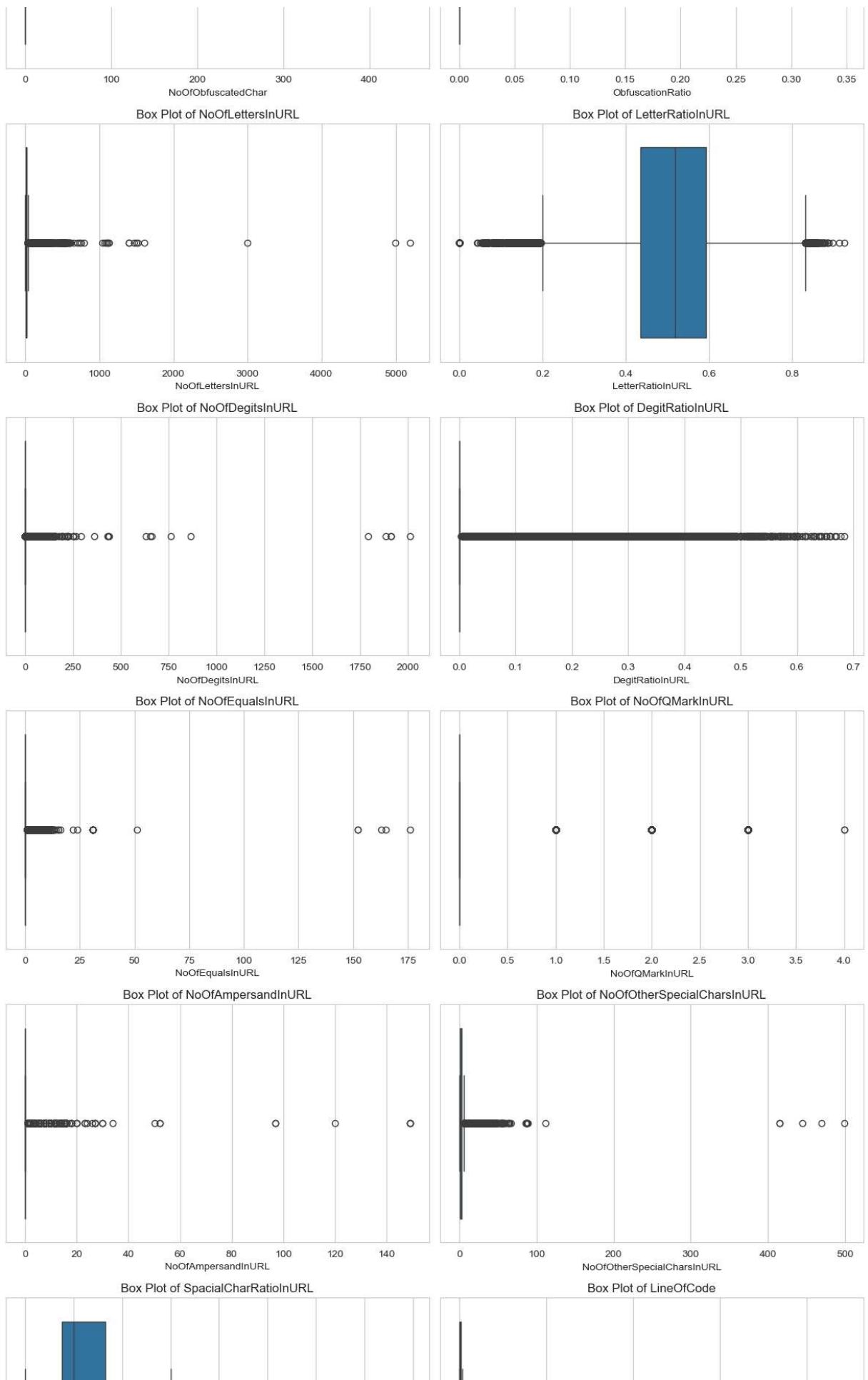
fig, axes = plt.subplots(rows, 2, figsize=(12, 4 * rows)) # Adjust size accordingly
axes = axes.flatten() # Flatten in case of multiple rows

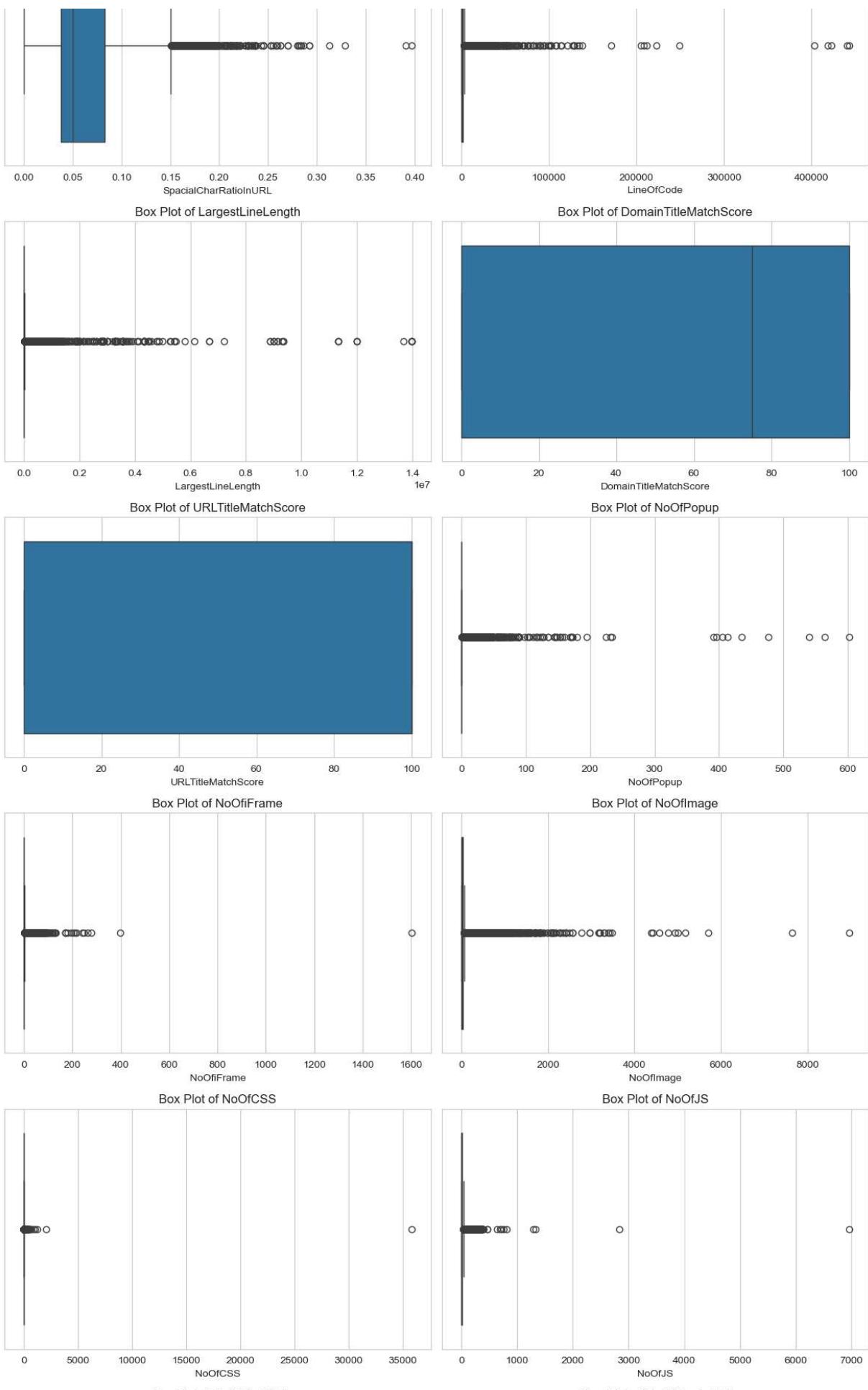
for i, col in enumerate(univariate_numerical_cols):
    sns.boxplot(x=df[col], ax=axes[i])
    axes[i].set_title(f'Box Plot of {col}')

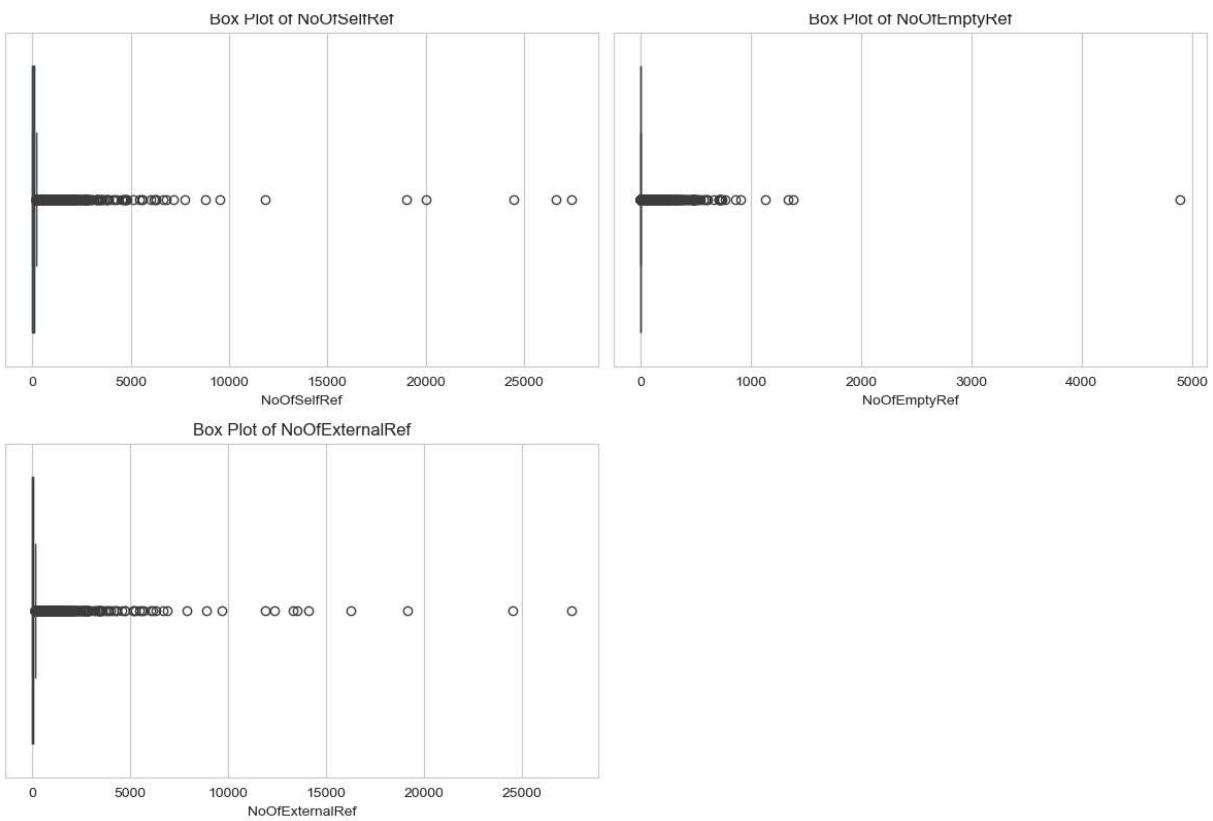
# Hide any unused subplots
for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout() # Adjust Layout for better spacing
plt.show()
```









Checking Skewness and Kurtosis

Skewness

```
In [44]: df[univariate_cols].skew()
```

```
Out[44]: URLLength          53.321737
DomainLength           2.513397
URLSimilarityIndex    -0.989903
CharContinuationRate   -1.128591
TLDLegitimateProb     0.073090
URLCharProb            -1.064291
TLDLength              1.663518
NoOfSubDomain          1.809400
NoOfObfuscatedChar     204.684335
ObfuscationRatio       40.118612
NoOfLettersInURL      58.402581
LetterRatioInURL       -0.066358
NoOfDigitsInURL        94.877574
DigitRatioInURL        3.244523
NoOfEqualsInURL        114.985479
NoOfQMarkInURL         8.160618
NoOfAmpersandInURL    106.717256
NoOfOtherSpecialCharsInURL 47.428522
SpacialCharRatioInURL 1.184134
LineOfCode              53.072297
LargestLineLength       48.179926
DomainTitleMatchScore   -0.008735
URLTitleMatchScore      -0.089085
NoOfPopup               84.952694
NoOfFrame               97.677347
NoOfImage               28.230253
NoOfCSS                 464.404031
NoOfJS                  140.435761
NoOfSelfRef              60.381378
NoOfEmptyRef             106.826225
NoOfExternalRef          65.900247
dtype: float64
```

Kurtosis

```
In [46]: df[univariate_cols].kurtosis()
```

```
Out[46]: URLLength           5914.008979
DomainLength          10.660247
URLSimilarityIndex      -0.408037
CharContinuationRate       0.056455
TLDLegitimateProb        -1.973777
URLCharProb             1.461184
TLDLength                13.738667
NoOfSubDomain            7.788128
NoOfObfuscatedChar       45903.366997
ObfuscationRatio         2125.333536
NoOfLettersInURL         8587.893558
LetterRatioInURL          0.015907
NoOfDigitsInURL          14318.789145
DigitRatioInURL           12.236281
NoOfEqualsInURL          19379.093750
NoOfQMarkInURL            84.597833
NoOfAmpersandInURL       16435.636935
NoOfOtherSpecialCharsInURL 5657.868346
SpacialCharRatioInURL      0.976044
LineOfCode                 5597.197274
LargestLineLength          3212.606584
DomainTitleMatchScore      -1.992655
URLTitleMatchScore         -1.984526
NoOfPopup                  10053.849746
NoOfFrame                  25457.057666
NoOfImage                  1785.807339
NoOfCSS                     222089.376914
NoOfJS                      41054.711733
NoOfSelfRef                  7648.047533
NoOfEmptyRef                  25638.891828
NoOfExternalRef              8289.647199
dtype: float64
```

High Skewness and Kurtosis founded!!

Outlier Removal

I decided to skip the outlier removal step to prevent overfitting, as removing outliers was significantly impacting the correlation between features and the target variable. I experimented with various methods, including the IQR method with different thresholds and the Z-score method, but all yielded similar results. Given this, I chose to proceed without outlier removal and instead focused on addressing skewness to improve model performance.

```
In [56]: df.shape
```

```
Out[56]: (235795, 55)
```

For Right-Skewed Data (Positive Skewness):

Use log, square root, or box-cox transformation.

```
In [58]: #To find right skewed columns
right_skewed_cols = df[univariate_cols].skew()[df[univariate_cols].skew() > 1].index
df[right_skewed_cols].skew()
```

```
Out[58]: URLLength           53.321737
DomainLength          2.513397
TLDLength             1.663518
NoOfSubDomain         1.809400
NoOfObfuscatedChar   204.684335
ObfuscationRatio     40.118612
NoOfLettersInURL    58.402581
NoOfDigitsInURL      94.877574
DigitRatioInURL      3.244523
NoOfEqualsInURL      114.985479
NoOfQMarkInURL       8.160618
NoOfAmpersandInURL  106.717256
NoOfOtherSpecialCharsInURL 47.428522
SpacialCharRatioInURL 1.184134
LineOfCode            53.072297
LargestLineLength    48.179926
NoOfPopup             84.952694
NoOfFrame             97.677347
NoOfImage              28.230253
NoOfCSS                464.404031
NoOfJS                 140.435761
NoOfSelfRef            60.381378
NoOfEmptyRef           106.826225
NoOfExternalRef        65.900247
dtype: float64
```

```
In [59]: df1=df.copy()
from scipy.stats import boxcox
for col in right_skewed_cols:
    skewness = df1[col].skew()
    if skewness > 1: # Highly skewed
        df1[col], _ = boxcox(df1[col] + 1) #using boxcox method for removing right
df1[right_skewed_cols].skew()
```

```
Out[59]: URLLength           0.095034
DomainLength          -0.046918
TLDLength             -0.014492
NoOfSubDomain         0.035998
NoOfObfuscatedChar   21.981435
ObfuscationRatio     21.981435
NoOfLettersInURL    -0.108138
NoOfDigitsInURL      1.390798
DigitRatioInURL      1.457255
NoOfEqualsInURL      6.378214
NoOfQMarkInURL       5.957591
NoOfAmpersandInURL  16.212384
NoOfOtherSpecialCharsInURL 0.497004
SpacialCharRatioInURL 0.279246
LineOfCode            -0.103982
LargestLineLength    0.013555
NoOfPopup              3.826578
NoOfFrame              0.802904
NoOfImage              0.057420
NoOfCSS                0.140952
NoOfJS                 0.017867
NoOfSelfRef            0.066672
NoOfEmptyRef           0.929639
NoOfExternalRef        0.061736
dtype: float64
```

For Left-Skewed Data (Negative Skewness):

Use power transformation (square or cube root) or negation.

```
In [61]: df2=df.copy()
left_skewed_cols = df2[univariate_cols].skew()[df2[univariate_cols].skew() <-1 ].index
df2[left_skewed_cols].skew()
```

```
Out[61]: CharContinuationRate   -1.128591
URLCharProb               -1.064291
dtype: float64
```

```
In [62]: # # Apply square transformation
for col in left_skewed_cols:
    df2[col] = df2[col]**2
df2[left_skewed_cols].skew()
```

```
Out[62]: CharContinuationRate   -0.832204
URLCharProb                 -0.399044
dtype: float64
```

```
In [63]: # Select only numeric columns
numeric_cols = df1[right_skewed_cols].select_dtypes(include=[ 'number'])

# Compute skewness of numerical columns
skewness = numeric_cols.skew()

# Identify columns with skewness > 1
cols_to_drop = skewness[skewness > 1].index
```

```
# Remove highly skewed columns
df1_new = df1[right_skewed_cols].drop(columns=cols_to_drop).copy()
```

A cleaned dataset is creating from df1 and df2.

```
In [65]: # Create an empty DataFrame for the cleaned data
df_cleaned = pd.DataFrame()

# Now define columns_right_skewed after creating df1_new
columns_right_skewed = df1_new.columns

# List the necessary columns
columns_needed = list(binary_cols) + list(left_skewed_cols) + list(columns_right_skewed)
df_cleaned = pd.DataFrame(columns=columns_needed) # Create DataFrame with required

# Assign the cleaned values to the corresponding columns
df_cleaned[binary_cols] = df[binary_cols]
df_cleaned[left_skewed_cols] = df2[left_skewed_cols]
df_cleaned[columns_right_skewed] = df1_new[columns_right_skewed]
df_cleaned[updated_categorical] = df[updated_categorical]
```

```
In [66]: df_cleaned.head(2)
```

	IsDomainIP	HasObfuscation	IsHTTPS	HasTitle	HasFavicon	Robots	IsResponsive	NoC
0	0	0	1	1	0	1	1	1
1	0	0	1	1	1	1	0	

2 rows × 42 columns

```
In [67]: numerical=df_cleaned.select_dtypes(include='number').columns
## Exclude binary columns from numerical analysis
univariate_cols= [col for col in numerical if df_cleaned[col].nunique()!=2]

# Check skewness only for existing columns
df_cleaned[univariate_cols].skew()
```

```
Out[67]: CharContinuationRate      -0.832204
          URLCharProb            -0.399044
          URLLength                0.095034
          DomainLength              -0.046918
          TLDLength                 -0.014492
          NoOfSubDomain             0.035998
          NoOfLettersInURL          -0.108138
          NoOfOtherSpecialCharsInURL 0.497004
          SpacialCharRatioInURL      0.279246
          LineOfCode                  -0.103982
          LargestLineLength           0.013555
          NoOfIFrame                  0.802904
          NoOfImage                   0.057420
          NoOfCSS                      0.140952
          NoOfJS                       0.017867
          NoOfSelfRef                  0.066672
          NoOfEmptyRef                 0.929639
          NoOfExternalRef               0.061736
          dtype: float64
```

Now a cleaned data with removed skewness is ready in df_cleaned !!

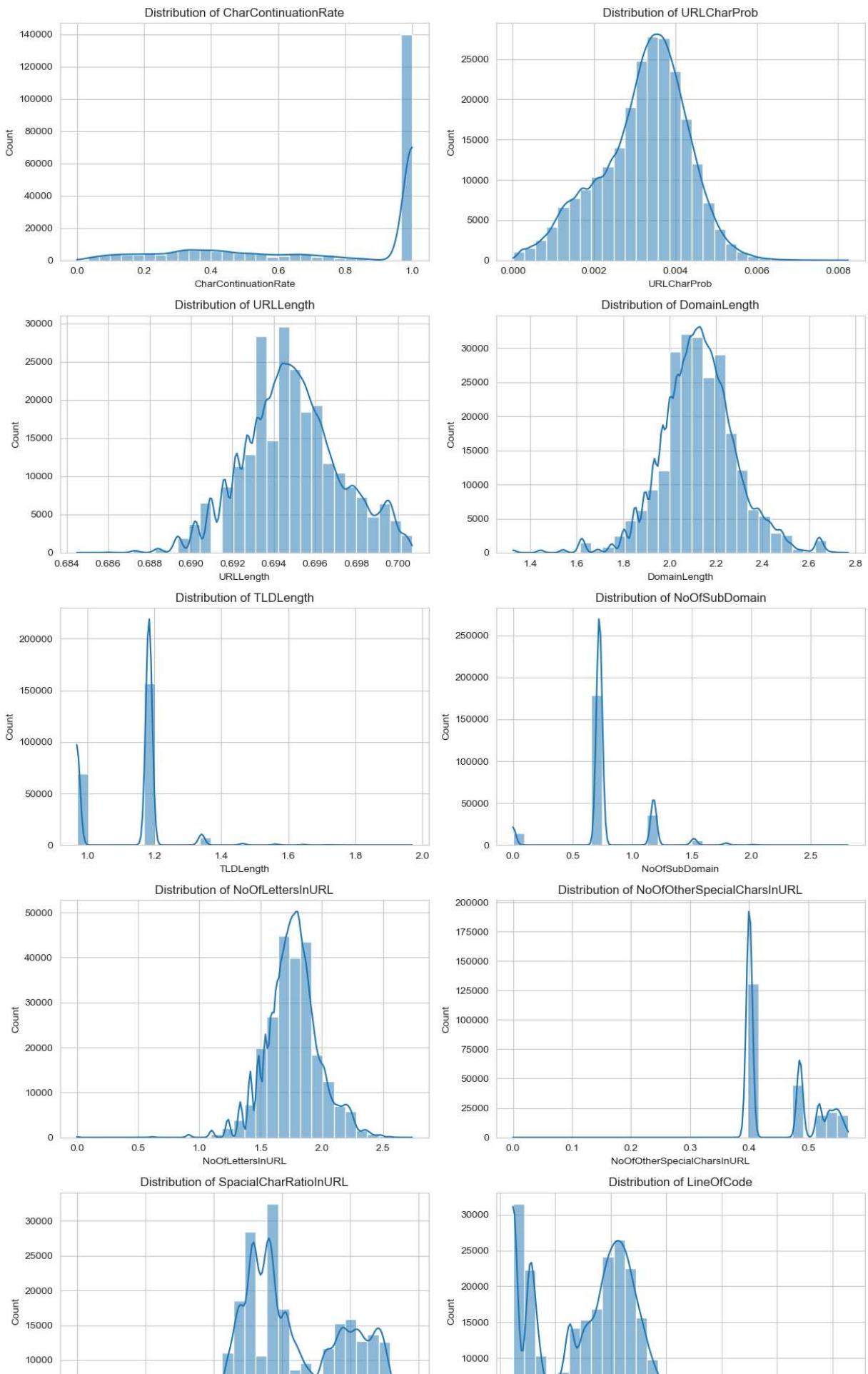
```
In [69]: # Assuming df is your DataFrame and univariate_numerical_cols is your list of column
num_cols = len(univariate_cols)
rows = (num_cols + 1) // 2 # Calculate required rows (ceil division for odd number

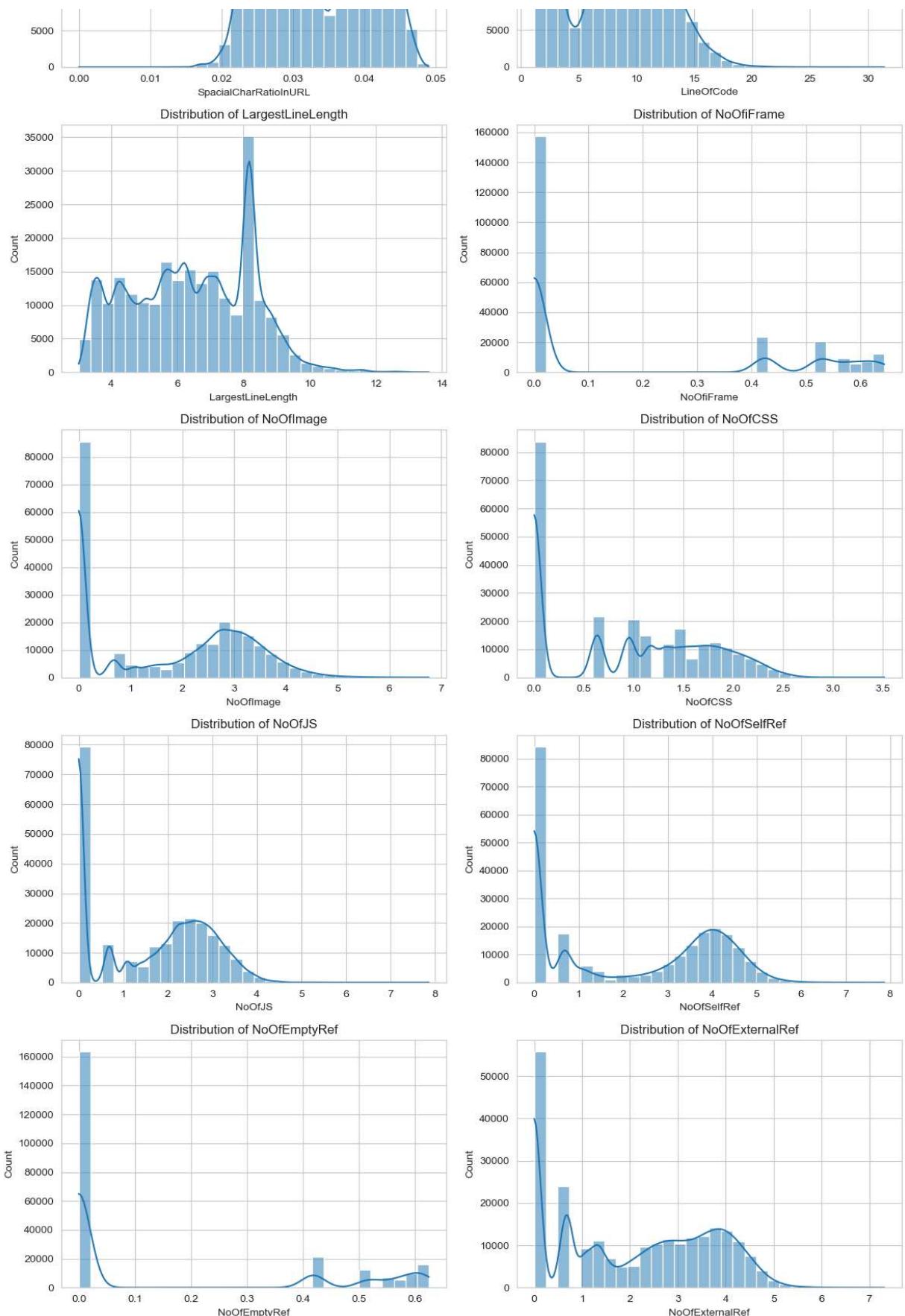
fig, axes = plt.subplots(rows, 2, figsize=(12, 4 * rows)) # Adjust size accordingly
axes = axes.flatten() # Flatten in case of multiple rows

for i, col in enumerate(univariate_cols):
    sns.histplot(df_cleaned[col], kde=True, bins=30, ax=axes[i])
    axes[i].set_title(f'Distribution of {col}')

# Hide any unused subplots
for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout() # Adjust layout for better spacing
plt.show()
```

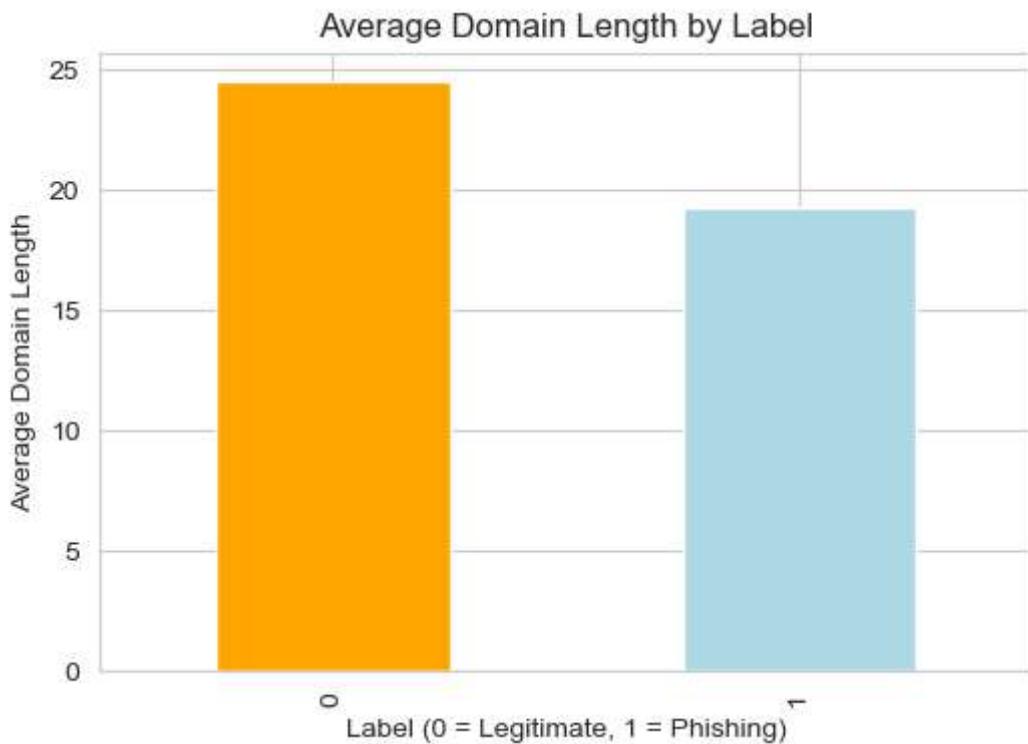




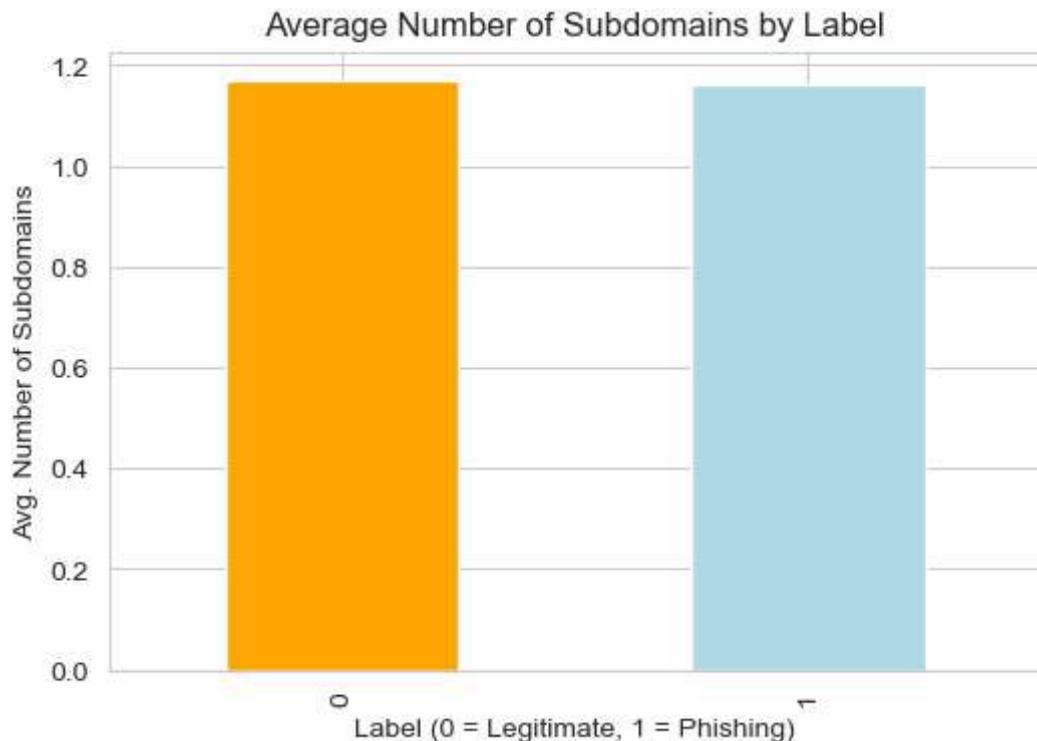
```
In [70]: #saving the cleaned dataset
# df_cleaned.to_csv("cleaned_url_data.csv", index=False)
```

Bivariate Analysis (Before removing skewness)

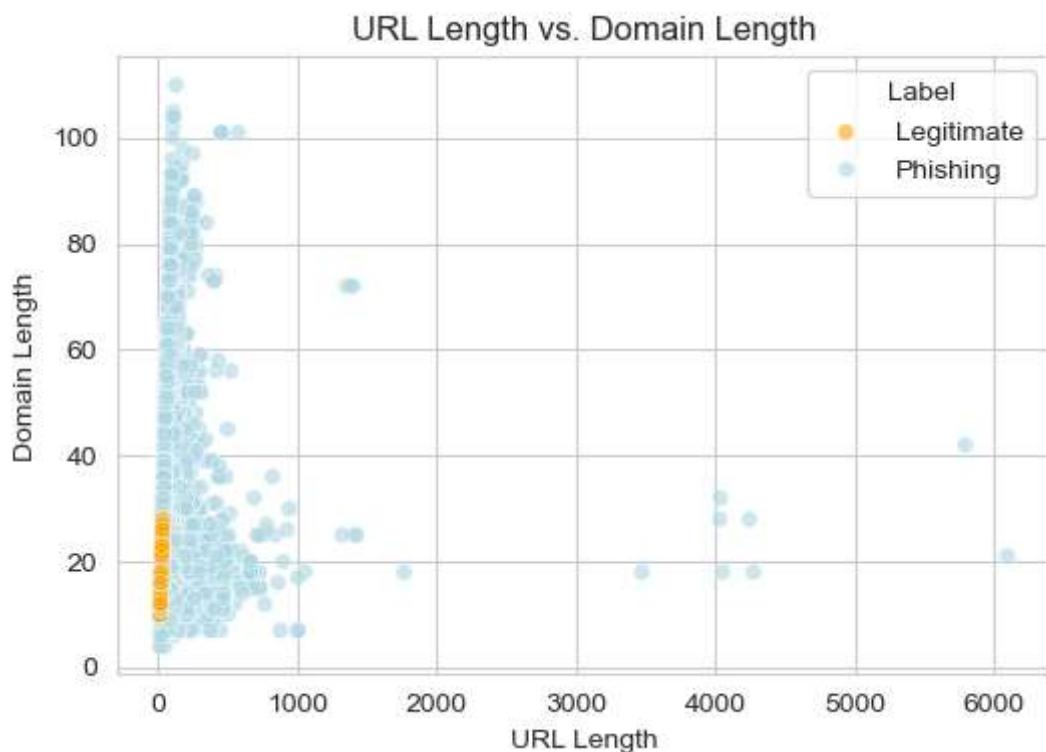
```
In [72]: plt.figure(figsize=(6, 4))
df.groupby('label')['DomainLength'].mean().plot(kind='bar', color=['orange', 'lightblue'])
plt.title("Average Domain Length by Label")
plt.xlabel("Label (0 = Legitimate, 1 = Phishing)")
plt.ylabel("Average Domain Length")
plt.show()
```



```
In [73]: # Bar Chart: Average Number of Subdomains by Label
plt.figure(figsize=(6, 4))
df.groupby('label')['NoOfSubDomain'].mean().plot(kind='bar', color=['orange', 'lightblue'])
plt.title("Average Number of Subdomains by Label")
plt.xlabel("Label (0 = Legitimate, 1 = Phishing)")
plt.ylabel("Avg. Number of Subdomains")
plt.show()
```



```
In [74]: # Scatter Plot: URL Length vs. Domain Length
plt.figure(figsize=(6, 4))
sns.scatterplot(x=df['URLLength'], y=df['DomainLength'], hue=df['label'], palette=[plt.title("URL Length vs. Domain Length")
plt.xlabel("URL Length")
plt.ylabel("Domain Length")
plt.legend(title="Label", labels=["Legitimate", "Phishing"])
plt.show()
```



Key Takeaway:

- Long URLs are a strong indicator of phishing attempts.
- Legitimate URLs tend to be shorter and more structured.
- Domain length alone is not a clear distinguishing factor but can contribute to a phishing detection model.

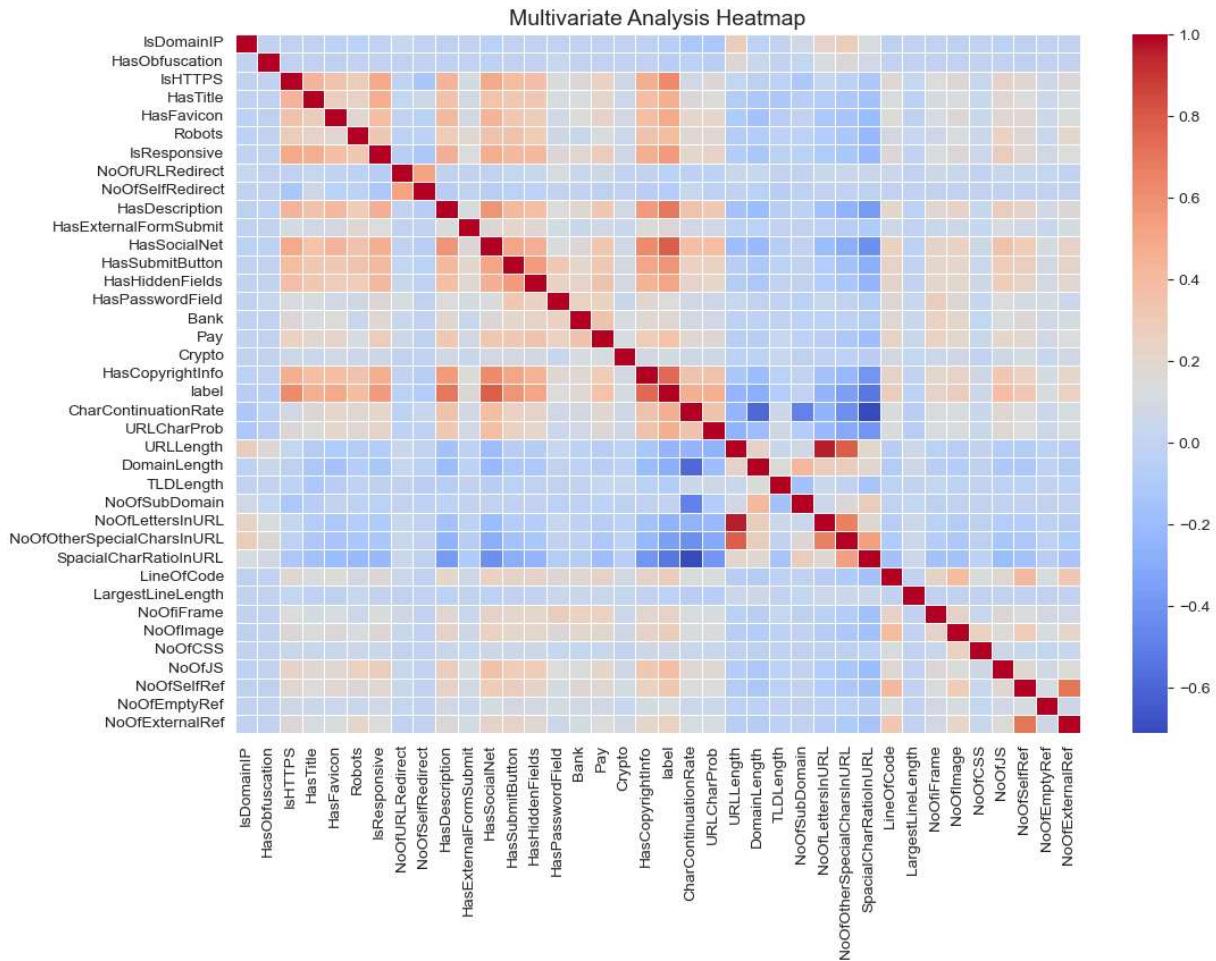
Multivariate Analysis

```
In [77]: # Compute the correlation matrix
corr_matrix = df[numerical].corr()

# Set figure size
plt.figure(figsize=(12, 8))

# Generate heatmap
sns.heatmap(corr_matrix, annot=False, cmap="coolwarm", linewidths=0.5)

# Show plot
plt.title("Multivariate Analysis Heatmap", fontsize=14)
plt.show()
```



```
In [78]: #Dropping highly correlated features (Either 'URL Length' or 'NoOfLettersInURL' )
df_cleaned.drop(columns='NoOfLettersInURL', inplace=True)
```

Encoding categorical values

Label encoding for ordinal and One hot for Nominal.

```
In [80]: categorical_cols=df_cleaned.select_dtypes(include=['object']).columns
categorical_cols
```

```
Out[80]: Index(['URL', 'Domain', 'TLD', 'Title'], dtype='object')
```

```
In [81]: df_cleaned.select_dtypes(include=['object']).head()
```

	URL	Domain	TLD	T
0	https://www.southbankmosaics.com	www.southbankmosaics.com	com	à,à¹à,à²à,§à,
1	https://www.uni-mainz.de	www.uni-mainz.de	de	johannes gutenbe universitÃ¤t ma
2	https://www.voicefmradio.co.uk	www.voicefmradio.co.uk	uk	voice fm southamp
3	https://www.sfnmjurnal.com	www.sfnmjurnal.com	com	home page: seminar fetal and neonatal me
4	https://www.rewillingargentina.org	www.rewillingargentina.org	org	fundaciÃ³n rewild argent

◀ ▶

```
In [82]: df_cleaned['URL'].nunique()
```

```
Out[82]: 235370
```

```
In [83]: # Drop URL
# Why :
''' It's a unique identifier and won't help in modeling directly. The URL column has
Label Encoding would assign arbitrary numbers, but the model wouldn't learn meaning

df_cleaned.drop(columns=['URL'], inplace=True)
```

```
In [84]: df_cleaned['Domain'].nunique()
```

```
Out[84]: 220086
```

```
In [85]: # Drop Domain
#Why :
'''Avoids high cardinality, reduces memory usage.Label encoding adds no meaningful

df_cleaned.drop(columns=['Domain'], inplace=True)
```

```
In [86]: # Target Encoding for TLD
'''Why: Target Encoding (also known as Mean Encoding) is preferred for TLD (Top-Level
mean target value → keeps a single column → reduces complexity.'''
df_encoded=df_cleaned.copy()

tld_target_mean = df_encoded.groupby('TLD')['label'].mean()
df_encoded['TLD'] = df_encoded['TLD'].map(tld_target_mean)

# TF-IDF for Title
'''Why: Title is a text feature (not categorical), so it cannot be directly used in
TF-IDF converts text into numerical vectors by capturing the importance of words.
Instead of simple counting (Bag-of-Words approach), TF-IDF weighs words based on how
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer(max_features=500)

# Transform 'Title' column
title_features = tfidf.fit_transform(df_encoded['Title']).toarray()

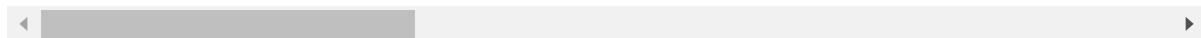
# Replace 'Title' with mean TF-IDF score for each row
df_encoded['Title'] = np.mean(title_features, axis=1)

df_encoded.head()
```

Out[86]:

	IsDomainIP	HasObfuscation	IsHTTPS	HasTitle	HasFavicon	Robots	IsResponsive	NoC
0	0	0	1	1	0	1	1	1
1	0	0	1	1	1	1	1	0
2	0	0	1	1	0	1	1	1
3	0	0	1	1	0	1	1	1
4	0	0	1	1	0	1	1	1

5 rows × 39 columns



```
In [87]: # Compute correlation of all features with the target
correlation = df_encoded.corr()['label'].sort_values(ascending=False)

print(correlation)
```

```

label           1.000000
NoOfSelfRef    0.872972
NoOfImage       0.867367
NoOfExternalRef 0.863774
LineOfCode      0.847514
NoOfJS          0.845733
NoOfCSS         0.805665
HasSocialNet    0.784255
HasCopyrightInfo 0.743358
HasDescription   0.690232
IsHTTPS         0.609132
HasSubmitButton 0.578561
TLD             0.554166
IsResponsive    0.548608
LargestLineLength 0.530157
NoOfFiFrame     0.523463
NoOfEmptyRef    0.518144
HasHiddenFields 0.507731
HasFavicon       0.493711
URLCharProb     0.470568
CharContinuationRate 0.470041
HasTitle         0.459725
Robots           0.392620
Pay              0.359747
Bank              0.188959
HasExternalFormSubmit 0.167574
HasPasswordField 0.138183
Crypto            0.099610
NoOfSubDomain    0.073378
NoOfURLRedirect -0.046456
HasObfuscation   -0.052473
IsDomainIP       -0.060202
TLDLength         -0.062608
NoOfSelfRedirect -0.076463
DomainLength      -0.206693
Title              -0.252750
URLLength         -0.409786
SpacialCharRatioInURL -0.534947
NoOfOtherSpecialCharsInURL -0.612807
Name: label, dtype: float64

```

Choosing 50000 instances for easy execution of the model before splitting the data.

```
In [89]: X = df_encoded.drop(columns=['label'])
y = df_encoded['label']
```

```
In [90]: # Select 50,000 random indices
random_indices = np.random.choice(X.shape[0], 50000, replace=False)

# Use NumPy indexing directly
X_subset = X.iloc[random_indices]
y_subset = y.iloc[random_indices]
```

Train test split

For "Random Forest", "XGBoost", "Decision Tree" we use unscaled data.

For "KNN", "SVM", "Logistic Regression" we use scaled data

```
In [92]: X_train, X_test, y_train, y_test = train_test_split(X_subset, y_subset, test_size=0.)
```

Feature Selection using Select KBest method

Because of large number of features, We need to choose the best set and doing further analysis.

Use the Elbow Method (Optimal Feature Selection)

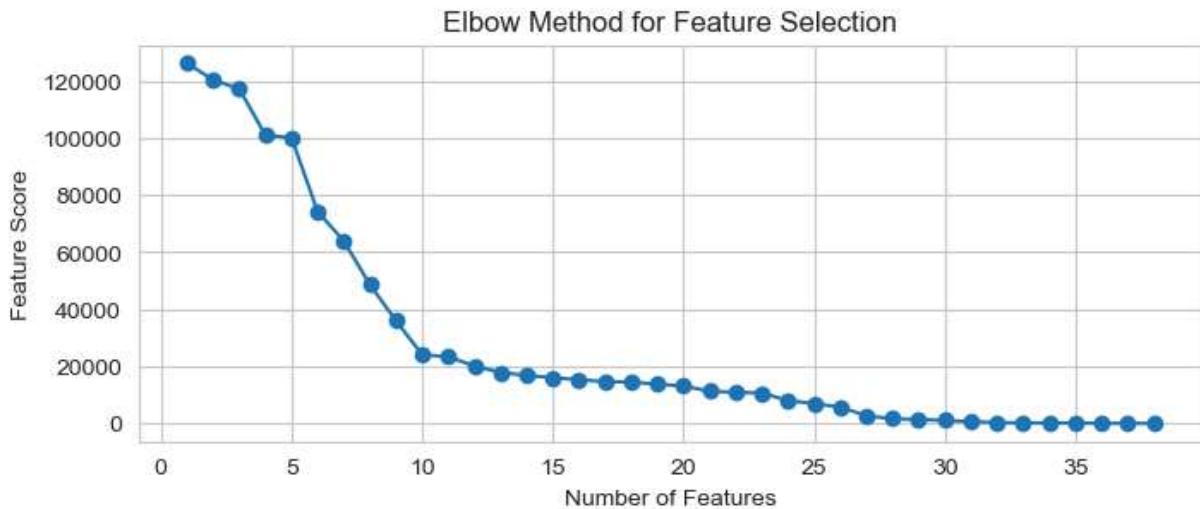
For choosing k value for SelectKBest method. We are using Elbow method.

```
In [95]: from sklearn.feature_selection import SelectKBest, f_classif

# Step 1: Compute feature scores only using training data
selector = SelectKBest(score_func=f_classif, k='all')
selector.fit(X_train, y_train)
scores = selector.scores_

# Step 2: Plot feature scores to find the elbow
sorted_scores = np.sort(scores)[::-1]

plt.figure(figsize=(8, 3))
plt.plot(range(1, len(scores) + 1), sorted_scores, marker='o')
plt.xlabel("Number of Features")
plt.ylabel("Feature Score")
plt.title("Elbow Method for Feature Selection")
plt.show()
```



choosing 15 as k value

```
In [97]: # Using Select Kbest method
# Step 3: Select best features based on training data only
selector = SelectKBest(score_func=f_classif, k=15)
```

```
X_train_selected = selector.fit_transform(X_train, y_train)

# Step 4: Apply same transformation to test data
X_test_selected = selector.transform(X_test)

# Get selected feature names
selected_feature_names = X.columns[selector.get_support()]
print("Selected Features:", selected_feature_names)
```

Selected Features: Index(['IsHTTPS', 'IsResponsive', 'HasDescription', 'HasSocialNet',
 'HasSubmitButton', 'HasCopyrightInfo', 'NoOfOtherSpecialCharsInURL',
 'SpacialCharRatioInURL', 'LineOfCode', 'NoOfImage', 'NoOfCSS', 'NoOfJS',
 'NoOfSelfRef', 'NoOfExternalRef', 'TLD'],
 dtype='object')

We got the features with best correlation with the target for training!!

Feature Scaling

In [100...]

```
# Initialize the scaler
scaler = StandardScaler()

# Fit and transform training data
X_train_scaled = scaler.fit_transform(X_train_selected)

# Transform test data using the same scaler
X_test_scaled = scaler.transform(X_test_selected)

# Convert back to DataFrame
X_train_scaled_df = pd.DataFrame(X_train_scaled, columns=selected_feature_names)
X_test_scaled_df = pd.DataFrame(X_test_scaled, columns=selected_feature_names)

# Display the first few rows
X_train_scaled_df.head()
```

Out[100...]

	IsHTTPS	IsResponsive	HasDescription	HasSocialNet	HassubmitButton	HasCopyrightInfo
0	-1.909388	-1.294997	-0.890459	-0.920036	-0.843952	-0.972
1	0.523728	0.772202	1.123016	1.086914	-0.843952	1.028
2	-1.909388	-1.294997	1.123016	-0.920036	-0.843952	-0.972
3	0.523728	0.772202	1.123016	1.086914	1.184901	1.028
4	0.523728	0.772202	-0.890459	-0.920036	1.184901	-0.972

Applying Dimensionality reduction using PCA

Only applying for "KNN", "SVM", "Logistic Regression"

```
In [102...]: from sklearn.decomposition import PCA

#Dimensionality Reduction
pca = PCA(n_components=10) # Adjust based on explained variance
X_train_pca = pca.fit_transform(X_train_scaled_df)
X_test_pca = pca.fit_transform(X_test_scaled_df)
```

Model fitting

```
In [104...]: # Step 2: Define models
models = {
    "KNN": KNeighborsClassifier(),
    "SVM": SVC(),
    "Logistic Regression": LogisticRegression(),
    "Random Forest": RandomForestClassifier(),
    "XGBoost": XGBClassifier(),
    "Decision Tree": DecisionTreeClassifier(),
}

# Step 3: Train models and store predictions
predictions = {}

for name, model in models.items():
    if name in ["KNN", "SVM", "Logistic Regression"]:
        model.fit(X_train_pca, y_train)
        y_pred = model.predict(X_test_pca)
    else:
        model.fit(X_train_selected, y_train)
        y_pred = model.predict(X_test_selected)

    predictions[name] = y_pred
    print(f" {name} trained and predictions stored!")

# Check stored predictions
print("\n Models trained successfully! Predictions are stored in 'predictions' dict")
```

KNN trained and predictions stored!
 SVM trained and predictions stored!
 Logistic Regression trained and predictions stored!
 Random Forest trained and predictions stored!
 XGBoost trained and predictions stored!
 Decision Tree trained and predictions stored!

Models trained successfully! Predictions are stored in 'predictions' dictionary.

```
In [105...]: # Dictionary to store evaluation results
evaluation_results = {}

# Loop through models and calculate metrics
for name, y_pred in predictions.items():
    evaluation_results[name] = {
        "Accuracy": accuracy_score(y_test, y_pred),
        "Precision": precision_score(y_test, y_pred, average='weighted'),
```

```

        "Recall": recall_score(y_test, y_pred, average='weighted'),
        "F1-score": f1_score(y_test, y_pred, average='weighted'),
        "ROC-AUC": roc_auc_score(y_test, y_pred) if len(set(y_test)) == 2 else "N/A"
    }

# Convert results to a DataFrame for better visualization
eval_df = pd.DataFrame(evaluation_results).T

# Optional: Sort by Accuracy for better visualization
eval_df = eval_df.sort_values(by="Accuracy", ascending=False)

# Display sorted results
print("\n🔍 Sorted Model Performance (by Accuracy):")
print(eval_df)

```

🔍 Sorted Model Performance (by Accuracy):

	Accuracy	Precision	Recall	F1-score	ROC-AUC
XGBoost	0.9993	0.999300	0.9993	0.9993	0.999273
Random Forest	0.9989	0.998900	0.9989	0.9989	0.998837
SVM	0.9981	0.998101	0.9981	0.9981	0.997967
Decision Tree	0.9980	0.998000	0.9980	0.9980	0.997962
KNN	0.9977	0.997700	0.9977	0.9977	0.997587
Logistic Regression	0.9975	0.997502	0.9975	0.9975	0.997300

Hyper parameter tuning

For SVM

In [108...]

```

from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform
param_dist = {
    'C': [0.1, 1, 10], # Fewer values
    'kernel': ['linear', 'rbf'], # Remove 'poly' (it's computationally expensive)
    'gamma': ['scale'] # Use only 'scale' (recommended for large datasets)
}

random_search_svm = RandomizedSearchCV(SVC(), param_dist, n_iter=5, cv=3, scoring='accuracy')
random_search_svm.fit(X_train_pca, y_train)

best_svm = random_search_svm.best_estimator_

# : Make predictions
y_pred_svm = best_svm.predict(X_test_pca)

# Step 6: Calculate all performance metrics
accuracy_svm = accuracy_score(y_test, y_pred_svm)
precision_svm = precision_score(y_test, y_pred_svm, average='weighted') # Use 'macro'
recall_svm = recall_score(y_test, y_pred_svm, average='weighted')
f1_svm = f1_score(y_test, y_pred_svm, average='weighted')
roc_auc_svm = roc_auc_score(y_test, y_pred_svm)

# Step 7: Print results
print("Best parameters for Logistic Regression:", random_search_svm.best_params_)
print(f"Test Accuracy: {accuracy_svm:.4f}")

```

```

print(f"Precision: {precision_svm:.4f}")
print(f"Recall: {recall_svm:.4f}")
print(f"F1 Score: {f1_svm:.4f}")
print(f"ROC-AUC Score: {roc_auc_svm:.4f}")

```

Best parameters for Logistic Regression: {'kernel': 'rbf', 'gamma': 'scale', 'C': 10}
Test Accuracy: 0.9983
Precision: 0.9983
Recall: 0.9983
F1 Score: 0.9983
ROC-AUC Score: 0.9982

For Logistic Regression

In [110...]

```

# Step 1: Define the hyperparameter grid
param_grid_logreg = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100], # Regularization strength
    'penalty': ['l1', 'l2'], # Type of regularization (L1 or L2)
    'solver': ['liblinear', 'saga'] # Solvers that support L1 and L2
}

# Step 2: Initialize Logistic Regression model
logreg = LogisticRegression()

# Step 3: Perform hyperparameter tuning using RandomizedSearchCV
random_search_logreg = RandomizedSearchCV(
    logreg, param_grid_logreg, cv=5, scoring='accuracy', n_jobs=-1, random_state=42
)
random_search_logreg.fit(X_train_pca, y_train)

# Step 4: Get the best model
best_logreg = random_search_logreg.best_estimator_

# Step 5: Make predictions on the test set
y_pred_logreg = best_logreg.predict(X_test_pca)

# Step 6: Calculate all performance metrics
accuracy_logreg = accuracy_score(y_test, y_pred_logreg)
precision_logreg = precision_score(y_test, y_pred_logreg, average='weighted') # Us
recall_logreg = recall_score(y_test, y_pred_logreg, average='weighted')
f1_logreg = f1_score(y_test, y_pred_logreg, average='weighted')
roc_auc_logreg = roc_auc_score(y_test, y_pred_logreg)

# Step 7: Print results
print("Best parameters for Logistic Regression:", random_search_logreg.best_params_)
print(f"Test Accuracy: {accuracy_logreg:.4f}")
print(f"Precision: {precision_logreg:.4f}")
print(f"Recall: {recall_logreg:.4f}")
print(f"F1 Score: {f1_logreg:.4f}")
print(f"ROC-AUC Score: {roc_auc_logreg:.4f}")

```

```
Best parameters for Logistic Regression: {'solver': 'liblinear', 'penalty': 'l1',
'C': 10}
Test Accuracy: 0.9975
Precision: 0.9975
Recall: 0.9975
F1 Score: 0.9975
ROC-AUC Score: 0.9973
```

For KNN

In [112...]

```
# Step 1: Define hyperparameter grid
param_grid_knn = {
    'n_neighbors': [3, 5, 7, 9, 11, 15, 20], # Number of neighbors
    'weights': ['uniform', 'distance'], # Weight function
    'metric': ['euclidean', 'manhattan', 'minkowski'] # Distance metric
}

# Step 2: Initialize KNN model
knn = KNeighborsClassifier()

# Step 3: Perform RandomizedSearchCV for hyperparameter tuning
random_search_knn = RandomizedSearchCV(
    knn, param_grid_knn, cv=5, scoring='accuracy', n_jobs=-1, random_state=42
)
random_search_knn.fit(X_train_pca, y_train)

# Step 4: Get the best model
best_knn = random_search_knn.best_estimator_

# Step 5: Make predictions on the test set
y_pred_knn = best_knn.predict(X_test_pca)

# Step 6: Calculate all performance metrics
accuracy_knn = accuracy_score(y_test, y_pred_knn)
precision_knn = precision_score(y_test, y_pred_knn, average='weighted') # Use 'macro'
recall_knn = recall_score(y_test, y_pred_knn, average='weighted')
f1_knn = f1_score(y_test, y_pred_knn, average='weighted')

# If it's a binary classification problem, calculate ROC-AUC
if len(set(y_test)) == 2:
    roc_auc_knn = roc_auc_score(y_test, y_pred_knn)
else:
    roc_auc_knn = "Not applicable for multiclass classification"

# Step 7: Print results
print("Best parameters for KNN:", random_search_knn.best_params_)
print(f"Test Accuracy: {accuracy_knn:.4f}")
print(f"Precision: {precision_knn:.4f}")
print(f"Recall: {recall_knn:.4f}")
print(f"F1 Score: {f1_knn:.4f}")
print(f"ROC-AUC Score: {roc_auc_knn:.4f}")
```

```
Best parameters for KNN: {'weights': 'distance', 'n_neighbors': 7, 'metric': 'manhattan'}
Test Accuracy: 0.9976
Precision: 0.9976
Recall: 0.9976
F1 Score: 0.9976
ROC-AUC Score: 0.9974
```

For Decision Tree

Decision Tree: Reduce Overfitting Since Decision Trees tend to overfit, try the following:

- ◆ Reduce Max Depth: Limit the tree depth using max_depth.
- ◆ Prune the Tree: Set min_samples_split (minimum samples to split a node) and min_samples_leaf (minimum samples per leaf).
- ◆ Use Cost Complexity Pruning (CCP): In sklearn.tree.DecisionTreeClassifier, adjust ccp_alpha (higher values prune more).
- ◆ Reduce Features Used: Try max_features to control feature selection.

In [115...]

```
# Optimized Decision Tree
dt = DecisionTreeClassifier(max_depth=8, min_samples_split=30, min_samples_leaf=15,
dt.fit(X_train_selected, y_train)

# Predictions
y_pred_dt = dt.predict(X_test_selected)

# Evaluation Metrics
dt_accuracy = accuracy_score(y_test, y_pred_dt)
dt_precision = precision_score(y_test, y_pred_dt)
dt_recall = recall_score(y_test, y_pred_dt)
dt_f1 = f1_score(y_test, y_pred_dt)
dt_roc_auc = roc_auc_score(y_test, dt.predict_proba(X_test_selected)[:, 1])

print(" Decision Tree Performance:")
print(f" Accuracy: {dt_accuracy:.4f}\n Precision: {dt_precision:.4f}\n Recall: {dt_
```

Decision Tree Performance:

```
Accuracy: 0.9757
Precision: 0.9725
Recall: 0.9850
F1 Score: 0.9788
ROC-AUC: 0.9755
```

For Random Forest:

Even though Random Forest is an ensemble of trees, it can still overfit. Try:

- ◆ Reduce Number of Trees: Lower n_estimators (e.g., 100 to 50).
- ◆ Limit Tree Depth: Use max_depth (e.g., 10 instead of None).
- ◆ Reduce Features per Split: Use max_features (e.g., "sqrt" or "log2").
- ◆ Increase Min Samples per Leaf: Set min_samples_leaf to at least 5-10.
- ◆ Use Out-of-Bag (OOB) Score: Enable oob_score=True to check generalization.

In [117...]

```
# Optimized Random Forest
rf = RandomForestClassifier(
    n_estimators=100, max_depth=10, min_samples_split=10, min_samples_leaf=5,
    max_features="sqrt", oob_score=True, random_state=42
)
rf.fit(X_train_selected, y_train)

# Predictions
y_pred_rf = rf.predict(X_test_selected)

# Evaluation Metrics
rf_accuracy = accuracy_score(y_test, y_pred_rf)
rf_precision = precision_score(y_test, y_pred_rf)
rf_recall = recall_score(y_test, y_pred_rf)
rf_f1 = f1_score(y_test, y_pred_rf)
rf_roc_auc = roc_auc_score(y_test, rf.predict_proba(X_test_selected)[:, 1])

print("\n随机森林性能:")
print(f" Accuracy: {rf_accuracy:.4f}\n Precision: {rf_precision:.4f}\n Recall: {rf_
```

随机森林性能:
Accuracy: 0.9987
Precision: 0.9984
Recall: 0.9993
F1 Score: 0.9989
ROC-AUC: 1.0000

XGBoost: Reduce Overfitting

XGBoost can be prone to overfitting, so apply these:

- ◆ Increase Regularization: Use lambda (L2 regularization) and alpha (L1 regularization).
- ◆ Use Lower Learning Rate: Try reducing learning_rate (e.g., 0.01-0.1).
- ◆ Reduce Tree Depth: Use max_depth=4-6 (default is 6).
- ◆ Add Early Stopping: Monitor validation loss and stop early.
- ◆ Reduce Boosting Rounds: Lower n_estimators (e.g., 100 instead of 500).
- ◆ Use Subsampling: Apply subsample=0.8 and colsample_bytree=0.8.

In [119...]

```
# Step 1: Train-Test Split BEFORE Feature Selection
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

```

# Step 2: Apply Feature Selection AFTER Splitting
selector = SelectKBest(score_func=f_classif, k=15) # Select best 15 features
X_train_selected = selector.fit_transform(X_train, y_train)
X_val_selected = selector.transform(X_val)
X_test_selected = selector.transform(X_test)

# Step 3: Train XGBoost Model
xgb_model = xgb.XGBClassifier(
    n_estimators=100, max_depth=6, learning_rate=0.05, reg_lambda=10, reg_alpha=5,
    subsample=0.8, colsample_bytree=0.8, random_state=42, eval_metric="logloss"
)

# Fit model using selected features
xgb_model.fit(X_train_selected, y_train, eval_set=[(X_val_selected, y_val)], verbose=1)

# Step 4: Predictions
y_pred_xgb = xgb_model.predict(X_test_selected)

# Step 5: Evaluation Metrics
xgb_accuracy = accuracy_score(y_test, y_pred_xgb)
xgb_precision = precision_score(y_test, y_pred_xgb)
xgb_recall = recall_score(y_test, y_pred_xgb)
xgb_f1 = f1_score(y_test, y_pred_xgb)
xgb_roc_auc = roc_auc_score(y_test, xgb_model.predict_proba(X_test_selected)[:, 1])

# Step 6: Print Performance Metrics
print("\n👉 XGBoost Performance:")
print(f" Accuracy: {xgb_accuracy:.4f}\n Precision: {xgb_precision:.4f}\n Recall: {xgb_recall:.4f}\n F1 Score: {xgb_f1:.4f}\n ROC-AUC: {xgb_roc_auc:.4f}")

```

👉 XGBoost Performance:
Accuracy: 0.9989
Precision: 0.9987
Recall: 0.9993
F1 Score: 0.9990
ROC-AUC: 1.0000

Now Building Pipelines

```

In [121...]: from sklearn.pipeline import Pipeline
# Define the pipeline steps without scaling
pipeline = Pipeline(steps=[
    ('feature_selection', SelectKBest(score_func=f_classif, k=15)), # Feature selection
    ('classifier', DecisionTreeClassifier(max_depth=8, min_samples_split=30, min_samples_leaf=5))
])

# Fit the pipeline on training data
pipeline.fit(X_train_selected, y_train)

# Make predictions on test data
y_pred_dt = pipeline.predict(X_test_selected)

# Evaluation Metrics
dt_accuracy = accuracy_score(y_test, y_pred_dt)
dt_precision = precision_score(y_test, y_pred_dt)

```

```

dt_recall = recall_score(y_test, y_pred_dt)
dt_f1 = f1_score(y_test, y_pred_dt)
dt_roc_auc = roc_auc_score(y_test, pipeline.predict_proba(X_test_selected)[:, 1])

# Print the evaluation metrics
print(" Decision Tree Performance:")
print(f" Accuracy: {dt_accuracy:.4f}\n Precision: {dt_precision:.4f}\n Recall: {dt_

```

Decision Tree Performance:
 Accuracy: 0.9756
 Precision: 0.9732
 Recall: 0.9846
 F1 Score: 0.9789
 ROC-AUC: 0.9765

Checking Overfitting for Decision tree classifier

Compare training vs. test accuracy:

```
In [123...]: train_pred = pipeline.predict(X_train_selected)
test_pred = pipeline.predict(X_test_selected)

train_acc = accuracy_score(y_train, train_pred)
test_acc = accuracy_score(y_test, test_pred)

print(f"Training Accuracy: {train_acc:.4f}")
print(f"Test Accuracy: {test_acc:.4f}")
```

Training Accuracy: 0.9770
 Test Accuracy: 0.9756

Best model is considered as Decision Tree classifier.

The training and test accuracy are very close, which means the model is well-generalized and not overfitting.

Final Decision: stopping tuning the parameters because:

There's no significant gap between training and test accuracy (indicating minimal overfitting). Your recall improved, which is good if you care about correctly identifying positive cases. The model is performing consistently on unseen data.

Predictions VS Test values using sample instances

```
In [126...]: # Select 20 random indices
random_indices = np.random.choice(X.shape[0], 20, replace=False)

# Use NumPy indexing directly
X_random = X.iloc[random_indices]
y_random = y[random_indices]
```

```
In [127...]
# Ensure X_random has the same feature selection applied as training data
X_random_selected = selector.transform(X_random) # Apply the same SelectKBest transformation

# Make predictions using the transformed data
predicted = pipeline.predict(X_random_selected)

# Convert to DataFrame
results = pd.DataFrame({
    "URL": df.loc[X_random.index, 'URL'], # Corresponding URLs
    "Actual": y_random,
    "Predicted": predicted
})

# Convert class labels to meaningful text (Optional)
results["Actual"] = results["Actual"].map({0: "Legitimate", 1: "Phishy"})
results["Predicted"] = results["Predicted"].map({0: "Legitimate", 1: "Phishy"})

# Display first few results
results.head(10)
```

Out[127...]

	URL	Actual	Predicted
95055	http://www.berenbord.nl	Legitimate	Legitimate
15106	http://www.3d-game.com	Legitimate	Legitimate
103282	https://www.amnesty.it	Phishy	Phishy
221741	http://www.obostreet.com	Legitimate	Legitimate
222717	https://hdfccpageekyc09.web.app/	Legitimate	Legitimate
121638	https://www.skiddle.com	Phishy	Phishy
59500	https://www.countryareacode.net	Phishy	Phishy
231441	https://www.forwardstepsblog.com	Phishy	Phishy
112547	https://mather32.wixsite.com/my-site	Legitimate	Legitimate
204170	http://brdls-3qaaa-aaaah-qafaq-cai.raw.ic0.app/	Legitimate	Legitimate

Saving the Model:

In [129...]

```
import joblib

# Save the model to a file
joblib.dump(pipeline, 'decision_tree_model.pkl')
print("Model saved successfully!")
```

Model saved successfully!

Loading the Model:

```
In [131]:  
# Load the model from the file  
loaded_model = joblib.load('decision_tree_model.pkl')  
  
# Use the Loaded model to make predictions  
y_pred_loaded = loaded_model.predict(X_test_selected)  
  
# Evaluate the Loaded model  
loaded_accuracy = accuracy_score(y_test, y_pred_loaded)  
print(f"Loaded Model Accuracy: {loaded_accuracy:.4f}")
```

Loaded Model Accuracy: 0.9756

Limitations of the model:

- Generalization Issues – Might fail on new phishing techniques or domains not in training data.
- Class Imbalance – If legitimate URLs dominate, the model may favor non-phishing classifications.
- Overfitting – If overly tuned to training data, it might perform poorly on unseen URLs.
- Latency & Performance – Too slow for real-time use if complex, making deployment difficult.
- Lack of Adaptive Learning – Phishing tactics evolve, and a static model may become outdated.
- Lack of Real-Time Analysis – The model may not work effectively on live URLs without additional web-based features.
- Static Analysis Constraint – Only analyzing the URL structure may miss phishing pages that dynamically load content.

Future Scopes :

- Integrate NLP Techniques – Use TF-IDF, word embeddings, or transformers (BERT/LSTM) to analyze URL text patterns.
- Enhance Feature Engineering – Include domain similarity (Levenshtein distance), tokenization, and n-gram analysis.
- Train Deep Learning Models – Use CNNs, RNNs, or transformers to better understand URL structures.
- Handle Real-Time URL Analysis – Implement live detection using APIs (e.g., Google Safe Browsing, VirusTotal).
- Improve Model Generalization – Regularly update the dataset to detect new phishing tactics.
- Optimize for Deployment – Convert the model into a lightweight API or browser extension for real-time use.

```
In [ ]:
```