

**UCHECK**

CSC 447 – Programming Project

Haneen albayoud

202105444

Dr. Khalil Mershad

**Introduction**

Phishing has rapidly developed into a complex and alarmingly widespread Internet risk in recent times. It is carried out by a malicious or hoax website links to which unsuspecting victim clicks and may fall victim to disastrous effects, such as data stealing, or pay DDoS attacks.

To fight back this increasing problem, I have created Ucheck, a very successfully tool that helps to check for the authenticity of URLs. Utilizing sophisticated algorithms Ucheck monitors and reviews web links in real time thus giving users a solid safety shield while browsing the web to avoid the dangers of phishing. Through this tool, not only the access to potentially dangerous websites but also the educational practical safe internet could be provided for the best individual security and organizational security.

**State of Art**

In the field of cybersecurity, more specifically phishing URL detection, several research projects utilize machine learning to improve the accuracy of their detection systems. A survey of research and industry suggests that many projects tend to use a suite of methods including Random Forest RF, Support Vector Machine SVM, and Logistic Regression LR. It is well established in the scientific community that SVM works particularly well in solving binary classification tasks such as phishing detection. It is a method that is often described as highly precise and interpretable. And after evaluating URL-based features, we have discovered many instances of machine learning research that conclude that SVM is a superior model for this task. The model is beneficial in analyzing high dimensional data, and this is particularly useful as scammers create more complex and larger datasets. Several papers in the community also cite Logistic Regression to be a useful method in solving phishing tasks. When the goal is towards classification, the output of the model of the model is also in probability form, aiding domain experts in understanding the output of the model. The third method Random Forest is widely used in fields where overfitting is an issue. The algorithm is based on multiple decision trees, and the ensemble model is designed to alleviate overfitting in algorithms. Random Forest uniquely works well in balancing independent categorical and continuous variables, most likely due to the heterogeneous data from the internet. In summary, the literature that we researched is largely consistent in that they all draw conclusions that support the above models. By doing so, we find that the community generally has a widely accepted set of models to solve website based threats. However, our approach stands out as it introduces several new features to the existing literature.

How it is Different:

1. Our unified model: Unlike the projects listed above, which leverage only one of these models in their machine learning system, we integrate and use them in conjunction with each other: a comprehensive model that aims to synergize the strengths of all these models and cancel out their individual shortcomings.

2. The \_wordlist\_ function: Ours is the first system to integrate a \_wordlist\_ function in our features to build a comprehensive dataset.

3.Enhanced Preprocessing Techniques: The project adopts refined preprocessing strategies. For example, to avoid any catastrophic failure due to unforeseen categories from untrained data, custom encoders like the SafeLabelEncoder are implemented to handle these issues cleanly in the operational phase of the model. This will improve the adaptability to new data and minimize label mismatch errors.

4. k-Fold Cross-Validation: k-Fold Cross-Validation is not only used as a model validation technique but also an integral part of the training process. This ensures that the entire dataset is utilized in separate training and validation capacities; this method is known to reduce variance in performance estimates.

5. Post-Modeling Analysis and Error Correction: Post-model deployment, the project integrates more backend checks beyond just the predictive output that is exported. These checks are designed to correct some errors that are false positives and negatives. All these additional features adopted by the URL detection system implemented in this project ensure it not only matches but exceeds the results from the best performing technologies in both industrial and academic domains. The integration of multiple models, numerous preprocessing and validation strategies have placed our system in a better position than the currently deployed phishing detection technologies.

**Design, Implementation and Testing**

Several software solutions, algorithms and libraries have been employed in the development of this application. The main technologies used include:

1. Colab for running the codes
2. Python script to generate an accurate urls dataset
3. 3 machine learning models to implement the dataset on them: Random forest, Logistic regression and supervised machine learning algorithm along with stratified kfold in order to enhance the accuracy.
4. UI using Tkinter library in python
5. Pickle to load the models on the ui

URL dataset generation

1. URL generator script

To write and run the models, I wanted to find some datasets that contain a fair amount of both legitimate and phishing urls along with their features in order to help the model compare and based on that output results. However, I couldn’t find any accurate datasets. That’s why, I have decided to generate my own, 100% accurate dataset.

To generate this dataset, first I used a function that generates random real an phishing urls. I implemented it such that it generates the same amount of real and phishing urls. That is because, I didn’t want my model to be bias, so I made sure that the number of real urls injected in the model are going to be equal to the number of phishing urls.

def generate\_random\_urls\_optimized(count=15000):

    fake = Faker()

    real\_domains = [

        "google.com", "facebook.com", "amazon.com", "youtube.com", "wikipedia.org",

        "twitter.com", "linkedin.com", "instagram.com", "reddit.com", "pinterest.com"

    ]

    phishing\_domains = [

        "google-security.com", "facebook-login.com", "amaz0n.co", "y0utube.com", "wikipediia.org",

        "tw1tter.com", "linkedinn.biz", "instagrarn.com", "redd1t.net", "pint3rest.com"

    ]

    deceptive\_paths = ['/login/', '/verify/', '/secure/', '/update-account/', '/confirm/']

    queries = ['?username=admin&password=admin', '?user=guest&auth=1234', '?account=update&service=mail']

    real\_paths = set()

    phishing\_urls\_set = set()

    # Generate unique real paths

    while len(real\_paths) < count // 2:

        path = '/' + '/'.join(fake.words(nb=random.randint(1, 3)))

        real\_paths.add(path)

    # Generate unique phishing URLs

    while len(phishing\_urls\_set) < count // 2:

        protocol = 'https://'

        domain = random.choice(phishing\_domains)

        deceptive\_path = random.choice(deceptive\_paths)

        query = random.choice(queries)

        phishing\_url = protocol + domain + deceptive\_path + query

        phishing\_urls\_set.add((phishing\_url, 'phishing'))

    # Convert sets to lists

    real\_urls = [(f"https://{random.choice(real\_domains)}{path}", 'real') for path in real\_paths]

    phishing\_urls = list(phishing\_urls\_set)

    # Shuffle both lists

    random.shuffle(real\_urls)

    random.shuffle(phishing\_urls)

    # Combine the two lists

    urls = real\_urls + phishing\_urls

    # Shuffle the combined list

    random.shuffle(urls)

    # Ensure the list has the correct count

    urls = urls[:count]

    # Ensure all URLs are unique and balance is maintained

    if len(urls) != len(set(urls)):

        raise ValueError("Duplicate URLs found in the generated list.")

    return urls

1. Google index and Web traffic for each url:

I tried to implement functions to find the google index and web traffic, and I also used google api, for the google index and similar web api for theweb trafficking. They worked at first, but then stopped. It turned out that they were limited and needed subscription.

Google index function:

def check\_google\_indexing(url, user\_agent):

    headers = {'User-Agent': user\_agent}

    query = {'q': 'info:' + url}

    google\_url = "https://www.google.com/search?" + urlencode(query)

    try:

        response = requests.get(google\_url, headers=headers)

        response.raise\_for\_status()

        soup = BeautifulSoup(response.text, "html.parser")

        # Check if the URL is indexed

        check = soup.find(id="rso").find("div", recursive=False).find("div", recursive=False).find("h3").find("a")

        indexed = True if check else False

        print(f"{url} is indexed: {indexed}")

        return indexed

    except requests.exceptions.HTTPError as e:

        print(f"HTTP error occurred: {e}")

        return False

    except requests.exceptions.RequestException as e:

        print(f"Other request error occurred: {e}")

        return False

    return indexed

Web traffic function

def get\_web\_traffic(url, api\_key):

    endpoint = f"https://api.similarweb.com/v1/website/{url}/total-traffic-and-engagement/visits"

    params = {

        'api\_key': api\_key,

        'start\_date': '2023-01',

        'end\_date': '2023-02',

        'main\_domain\_only': False

    }

    response = requests.get(endpoint, params=params)

    if response.status\_code == 200:

        return response.json()

    else:

        return {"error": f"Failed to fetch data: {response.status\_code}"}

1. DNS records

I also implemented a function that gets the dns records of a url. This, then, in the machine learning models, takes the records and checks if atleast on of them is present and returns True, else false.

def get\_dns\_records(domain):

    records = {'A': [], 'MX': [], 'NS': []}

    try:

        a\_records = dns.resolver.resolve(domain, 'A')

        records['A'] = [rdata.address for rdata in a\_records]

    except (dns.resolver.NXDOMAIN, dns.resolver.NoAnswer, dns.resolver.Timeout, dns.exception.DNSException):

        records['A'] = 'Unavailable'

    try:

        mx\_records = dns.resolver.resolve(domain, 'MX')

        records['MX'] = [rdata.exchange.to\_text() for rdata in mx\_records]

    except (dns.resolver.NXDOMAIN, dns.resolver.NoAnswer, dns.resolver.Timeout, dns.exception.DNSException):

        records['MX'] = 'Unavailable'

    try:

        ns\_records = dns.resolver.resolve(domain, 'NS')

        records['NS'] = [rdata.to\_text() for rdata in ns\_records]

    except (dns.resolver.NXDOMAIN, dns.resolver.NoAnswer, dns.resolver.Timeout, dns.exception.DNSException):

        records['NS'] = 'Unavailable'

    return records

1. WHO-IS registry

The who registry was also checked for every url for more accuracy and insurance.

def check\_whois\_registered(domain):

    try:

        w = whois.whois(domain)

        # Instead of checking just domain\_name, check for other fields as well

        if any([w.domain\_name, w.registrar, w.creation\_date, w.expiration\_date]):

            return 1

    except whois.parser.PywhoisError as e:  # Adjusted for a more specific exception

        print(f"WHOIS lookup failed for {domain}: {e}")

    except Exception as e:  # Catching any other exception that might occur

        print(f"Unexpected error during WHOIS lookup for {domain}: {e}")

    return 0

1. URL features

Another important feature in the phishing vs. benign URL classification task is the examining of the URL properties, including the URL content, length, domain name position, domain name, and top-level domain. Although some of them are not visible directly to human users, phishing URLs often have slightly different structures from the actual benign URLs, and the difference just lies in its subtleness. These small structural difference actually matters and can contribute to identifying phishing threats in their early stages.

def extract\_features(url):

    suspicious\_tlds = ['xyz', 'info', 'top', 'gq', 'cf', 'tk', 'ml', 'ga', 'men', 'loan', 'date', 'win', 'faith', 'review', 'party', 'webcam', 'trade', 'accountant', 'download', 'racing', 'science', 'cricket', 'bid']

  features = {}

    parsed\_url = tldextract.extract(url)

    features['length'] = len(url)

    features['protocol'] = url.split('://')[0]

  features['domain'] = parsed\_url.domain

  features['subdomain'] = parsed\_url.subdomain

    features['suffix'] = parsed\_url.suffix

    features['path'] = url.split(parsed\_url.domain)[-1]

    features['number\_of\_subdomains'] = len(parsed\_url.subdomain.split('.')) if parsed\_url.subdomain else 0

  if contains\_ip(url):

      features['has\_ip\_address'] = True

    else:

      features['has\_ip\_address'] = False

    features['is\_https'] = features['protocol'] == 'https'

    features['special\_char\_count'] = sum(not c.isalnum() for c in url)

    suspicious\_words = ['login', 'verify', 'account', 'secure', 'update', 'banking']

    features['has\_suspicious\_word'] = any(word in url for word in suspicious\_words)

    features['is\_suspicious\_tld'] = features['suffix'] in suspicious\_tlds

    return features

1. Excel table generator

Lastly, all the data generated with these functions were combined and put in an excel sheet. And also, for each url generated, I made sure that the url generator function returns to me whether it’s legitimate or phishing.

def generate\_and\_collect\_data(count=15000, include\_suspicious=False, suspicious\_ratio=0.3, api\_key=None, cse\_id=None, similarweb\_key=None, safe\_browsing\_key=None, uuid=None):

    # Generate URLs with associated labels

    urls\_with\_labels = generate\_random\_urls\_optimized(count)

    data = []

    for url, label in urls\_with\_labels:

        # Initialize the data dictionary for the URL

        url\_data = {'URL': url, 'Label': label}

        # Extract URL features

        url\_data.update(extract\_features(url))

        # Parse the domain information

        parsed\_url = tldextract.extract(url)

        full\_domain = f"{parsed\_url.subdomain}.{parsed\_url.domain}.{parsed\_url.suffix}".strip('.')

        # Add DNS records if the domain is valid

        if full\_domain:

            url\_data.update(get\_dns\_records(full\_domain))

        # Check WHOIS registration status

        url\_data['WHOIS\_Registered'] = check\_whois\_registered(full\_domain)

        # Additional API-based features can be added here

        # Append the data dictionary to the list

        data.append(url\_data)

    # Convert the list of dictionaries into a DataFrame

    df = pd.DataFrame(data)

    return df

Machine Learning models

As I have mentioned before, I used three models to ensure better accuracy in this project.

For all of these models, the dataset loading and data preprocessing and kfolds methods were the same.

1. Data Loading:

data\_path = "/content/url\_data (4).csv"

data = pd.read\_csv(data\_path)

data.head()

1. Data preprocessing:

In order to obtain the best possible model, data exploration and preprocessing were crucial to prepare the dataset. A preprocessing function was created to transform the dataset into a form that can be easily aggregated to the machine learning model. This function first removes the unnecessary columns that will not be used in any analysis such as these: 'URL', 'path' and 'subdomain'. Next it creates some label encoders for the columns of categorical data: 'protocol', 'domain', 'suffix' and 'Label'. This is to transform those columns into numerical format, as is common practice before applying supervised machine learning methods. Apart from this, the function transforms the same columns into binary representation in case they contain 'Unavailable' string. This is to encode the absence of the particular DNS record (A, MX, NS) or their particular combination. A new feature 'DNS records' will keep the information if there are any DNS records in the host. The given pre-processing steps seem to be necessary in order to use the data in predictive modeling. These steps may have an impact on model accuracy and efficiency. And also, the data at the end was split in order to use some of the data for testing.

def preprocess\_data(df):

    # Define the function to encode 'Unavailable'

    def encode\_availability(column):

        return column.apply(lambda x: 0 if x == 'Unavailable' else 1)

    # Drop unnecessary columns

    df.drop(['URL', 'path', 'subdomain'], axis=1, inplace=True)

    # Initialize LabelEncoders

    le\_protocol = LabelEncoder()

    le\_domain = LabelEncoder()

    le\_suffix = LabelEncoder()

    le\_label = LabelEncoder()

    # Manually fit the LabelEncoder with both possible values for 'protocol'

    le\_protocol.fit(['http', 'https'])

    # Fit other encoders with data from the DataFrame

    le\_domain.fit(df['domain'].unique())

    le\_suffix.fit(df['suffix'].unique())

    le\_label.fit(df['Label'])

    # Apply LabelEncoder to the data

    df['protocol'] = le\_protocol.transform(df['protocol'])

    df['domain'] = le\_domain.transform(df['domain'])

    df['suffix'] = le\_suffix.transform(df['suffix'])

    df['Label'] = le\_label.transform(df['Label'])

    # Apply the 'encode\_availability' function

    df['A'] = encode\_availability(df['A'])

    df['MX'] = encode\_availability(df['MX'])

    df['NS'] = encode\_availability(df['NS'])

    # Create a new 'DNS records' column based on the availability of A, MX, or NS

    df['DNS records'] = (df['A'] | df['MX'] | df['NS']).astype(int)

    # Optionally drop the original columns if they are no longer needed

    df.drop(['A', 'MX', 'NS'], axis=1, inplace=True)

    return df, le\_protocol, le\_domain, le\_suffix, le\_label

data, le\_protocol, le\_domain, le\_suffix, le\_label = preprocess\_data(data.copy())

# Now, we select our features and target variable for the model

X = data.drop('Label', axis=1)  # Features

y = data['Label']               # Target variable

# Split data into training and testing sets

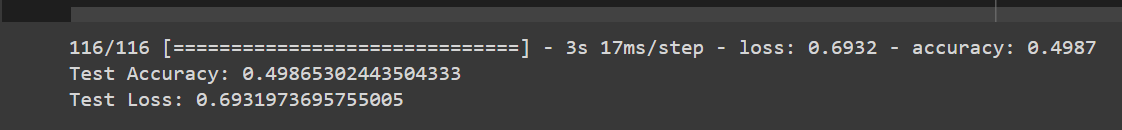
X\_train\_cv, X\_test, y\_train\_cv, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

1. Machine Learning Models

Long-short term Memory:

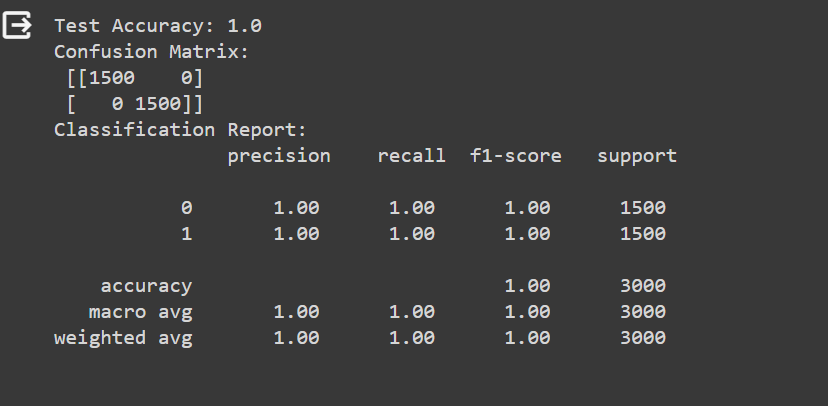
At first, I tried using the LTSM model, however, the accuracy was very low. It was approximately 0.5, which meant that the model was guessing. After doing some research, it turns out that ltsm actually wasn’t really the best type of model to use for excel datasets as its used for datasets where the order of the information matter, like time series, unlike the dataset we have where theres no specific sequence.

<https://colab.research.google.com/drive/1KlxcsZoq4KKbv9w4WC2AOzruWMTWeCSh#scrollTo=oCsKq-kTr3JZ>



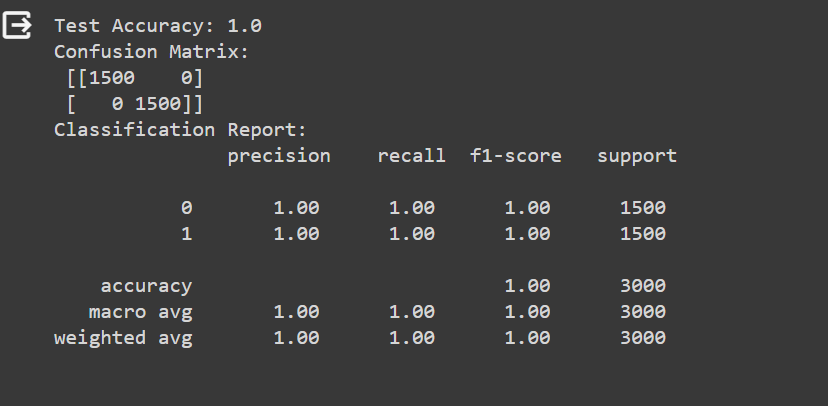
Random Forest:

For the URL dataset, I chose to use a Random Forest model. Random Forest can handle complex and nonlinear relationships between features quite well. URL data, such as URL length, special characters, and the structure of domains, can contain lots of this kind of relationship. This model is useful when working with large datasets having a large number of different types of features. Random Forest also supports feature importance, which indicates the most powerful predictors that can distinguish a malicious URL from a benign one. Finally, since Random Forest uses an ensemble of decision trees, it captures more variance and reduces overfitting with increased data volume and diversity. This makes it flexible enough to deal with the variations and noise present in URL datasets.



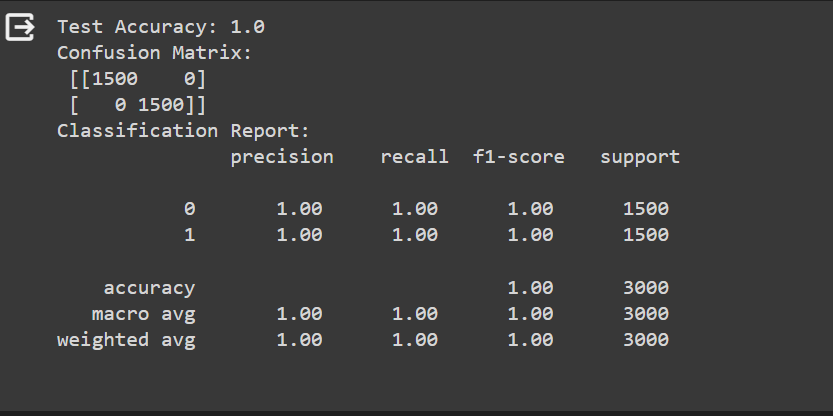
Support Vector Machine (SVM):

Support Vector Machine (SVM) is particularly useful in finding the best separation boundary between these classes. It can classify high-dimensional space very well. The URL dataset contains some of those features such as hostname length, IP addresses in URLs, and use of proper protocol. SVMs have the flexibility to use different kernel functions, mapping the feature space and generating a hyperplane that separates the classes of interest in the transformed space. This can be very important in identifying subtle features that are unique to certain phishing cases.



Logistic Regression:

Why use logistic regression in URL data? Simply because logistic regression is a mathematical equation that uses probabilities of a piece of data belonging to class 1 of the dependent variable. This algorithm is easy to implement and returns interpretable performance, hence quite suitable as a baseline model in our predictive tasks too. URL data typically have binary features, e.g. whether certain tokens or characters are present or not. Binary classification is the natural problem for Logistic Regression, and thus, serves as a good starting point.



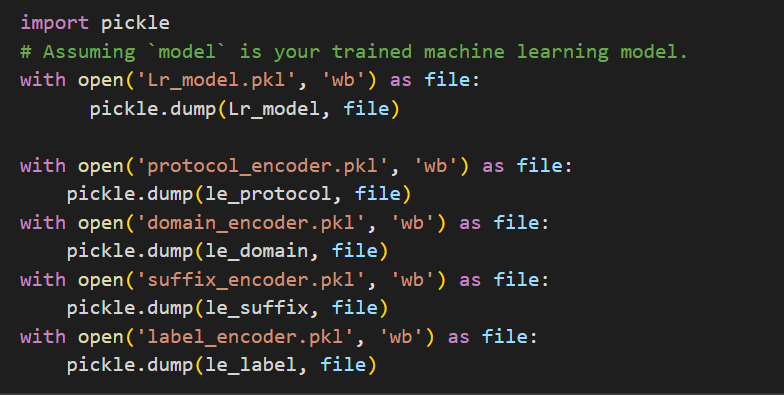
k-Fold Cross-Validation:

To guarantee that our model output is robust and generalisable, I had implemented k-Fold Cross-Validation in our modelling process. k-Fold Cross-Validation systematically splits the dataset into 'k' small ones, and trains on 'k - 1' small splits, validates on the remaining split. Using over k times with the models are trained/validated again, in each iteration with a different split as the validation set and rest as the training set. k-Fold helps not only to maximise the amount of data used in training and validation, but also gives good idea of how the model performs across different sections of the data. Moreover, k-Fold helps to avoid the risk of overfitting too, as the quality of the model can be evaluated multiple times. Thus, the performance metrics of k-Fold Cross-Validation would be more reliable and definitely give us a better understanding of real-world performance than a traditional validation set split.

After the Kfold, the accuracies remained the same. I met with the machine learning dr for this problem, and after analyzing the codes together, we concluded that even though its 1, based on my dataset and the codes, my accuracy is pretty good. However, youll see in the backend implementation part, that I had some false positives and I tried to fix them by adding more checking functions in the backend code, that check more after implementing the model to ensure that I get more accurate results.

Loading the models on the application

After finishing with running the models, training them and testing them, I used pickle library to download each model, along with the label encoders I used in the preprocessing part in order to use them in the “backend” code implemented in app.py.



1. Loading the machine learning models

# Load LabelEncoders

with open('protocol\_encoder.pkl', 'rb') as f:

    le\_protocol = pickle.load(f)

with open('domain\_encoder.pkl', 'rb') as f:

    le\_domain = pickle.load(f)

with open('suffix\_encoder.pkl', 'rb') as f:

    le\_suffix = pickle.load(f)

# Load models

models = {

    'SVM': pickle.load(open('svm\_model (1).pkl', 'rb')),

    'RF': pickle.load(open('rf\_model (1).pkl', 'rb')),

    'LR': pickle.load(open('Lr\_model (1).pkl', 'rb'))

}

1. Reimplementation of functions for features extraction

Same as the ones in the url dataset generator script (IP, dns, features). Ive implemented the urltofeatures function, that takes the features in a url and puts them in a dictionary and then turns them into dataframe so that the ml models accept them.

While testing the models, ive detected an error, that some of the urls the model wont take them because it haven’t seen them before. So I defined a custom class SafeLabelEncoder extending scikit-learn’s LabelEncoder functionality by safely fitting to unseen labels that might arise when transforming, and safely encoding categorical features by fitting to any known categories and storing these categories. When transforming, if an unknown category is encountered during transformation (i.e., not seen during fitting), it returns the predefined unknown class identifier, otherwise raising an error (as would happen in scikit-learn’s LabelEncoder). Finally, in the accompanying script, I fit and transform the ‘protocol’, ‘domain’ and ‘suffix’ columns of a new DataFrame features\_df created from a list of features: All categorical data in these columns have been numerically encoded so that the data is ready to work with machine learning models that expect numerical features.

class SafeLabelEncoder(BaseEstimator, TransformerMixin):

    def \_\_init\_\_(self, unknown\_class='unknown'):

        self.encoder = LabelEncoder()

        self.unknown\_class\_ = unknown\_class

    def fit(self, X, y=None):

        self.encoder.fit(X)

        self.classes\_ = set(self.encoder.classes\_)

        return self

    def transform(self, X, y=None):

        if X is None or X not in self.classes\_:

            return self.unknown\_class\_

        else:

            return self.encoder.transform([X])[0]

def url\_to\_features(url, protocol\_encoder, domain\_encoder, suffix\_encoder):

    url\_extract = tldextract.extract(url)

    features = {

        'length': len(url),

       'protocol': url.split('://')[0] if '://' in url else 'http',

        'domain': url\_extract.domain if url\_extract.domain else 'unknown',

        'suffix': url\_extract.suffix.split('.')[0] if url\_extract.suffix and url\_extract.suffix.split('.')[0] in ['com', 'edu', 'gov', 'org', 'net'] else 'unknown',

        'number\_of\_subdomains': len(url\_extract.subdomain.split('.')) if url\_extract.subdomain else 0,

        'has\_ip\_address': contains\_ip(url),

        'is\_https': 1 if url.startswith('https') else 0,

        'special\_char\_count': sum(not c.isalnum() for c in url),

        'has\_suspicious\_word': 1 if any(word in url for word in ['login', 'verify', 'account', 'secure', 'update', 'banking']) else 0,

        'is\_suspicious\_tld': 1 if url\_extract.suffix in ['xyz', 'info', 'top', 'gq', 'cf', 'tk', 'ml', 'ga', 'men', 'loan', 'date', 'win', 'faith', 'review', 'party', 'webcam', 'trade', 'accountant', 'download', 'racing', 'science', 'cricket', 'bid'] else 0,

        'DNS records': has\_dns\_server(url\_extract.domain)

    }

    features\_df = pd.DataFrame([features])

    features\_df['protocol'] = protocol\_encoder.transform(features\_df['protocol'])

    features\_df['domain'] = domain\_encoder.transform(features\_df['domain'])

    features\_df['suffix'] = suffix\_encoder.transform(features\_df['suffix'])

    return features\_df

And lastly, Ive implemented the predict url code, that uses the model chosen from the ui as well as checks with some extra if conditions for more accuracy.

def predict\_url(url, model, protocol\_encoder, domain\_encoder, suffix\_encoder):

    try:

        features\_df = url\_to\_features(url, protocol\_encoder, domain\_encoder, suffix\_encoder)

    except ValueError:

        url\_extract = tldextract.extract(url)  # Catch the error when the encoder encounters an unseen label

        if has\_dns\_server(url\_extract.domain):

            return "Real"

        else:

            return "Phishing"  # Treat unseen labels as 'phishing'

    # Additional model prediction logic if needed

    predicted\_label = model.predict(features\_df)

    if predicted\_label == [1]:

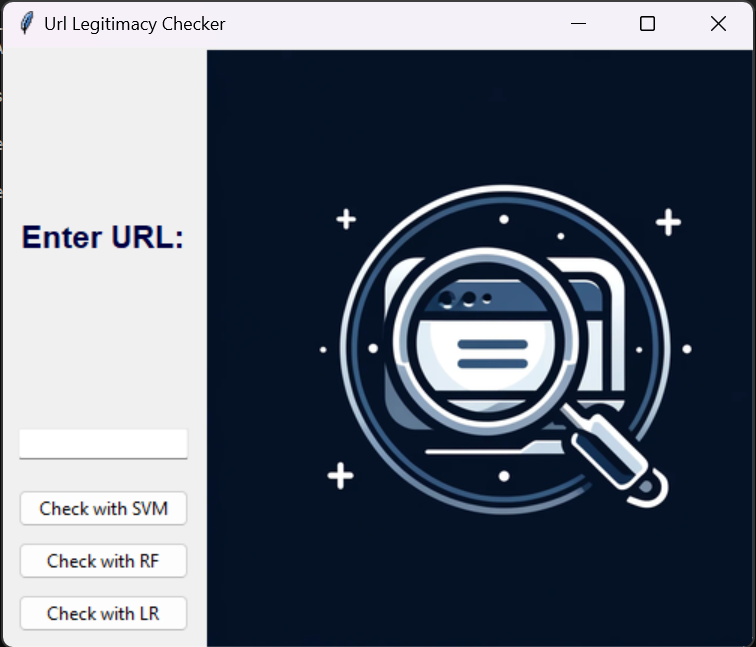
        return "Real"

    else:

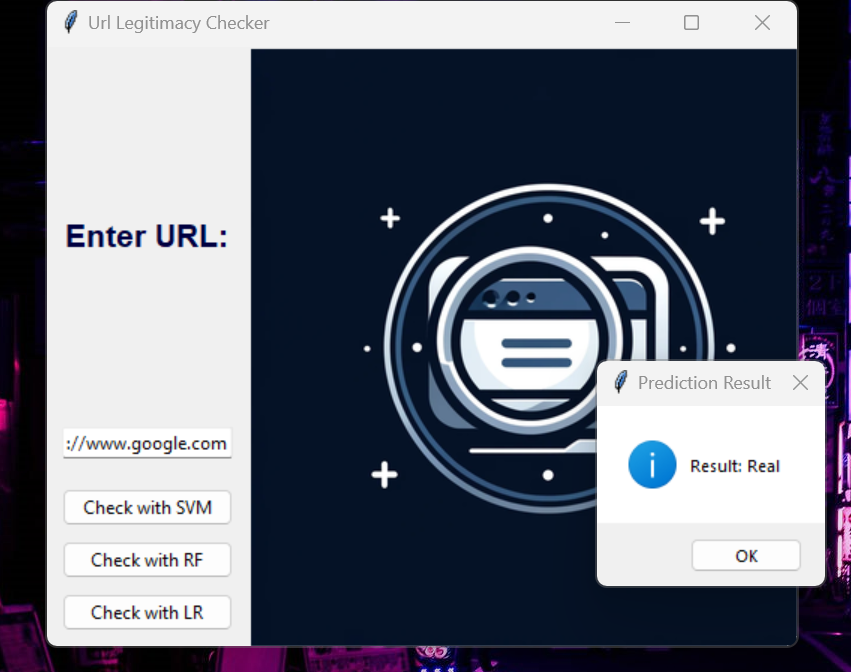
        return "Phishing"

1. GUI

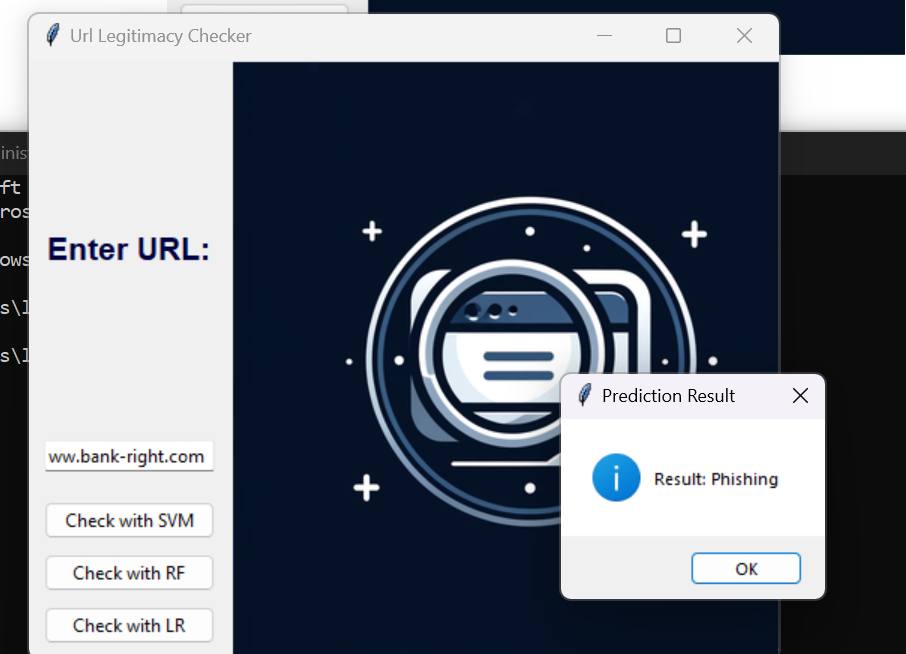
For the user interface, I used a basic tkinter interface with buttons for each model and a url input field.



For legitimate urls:

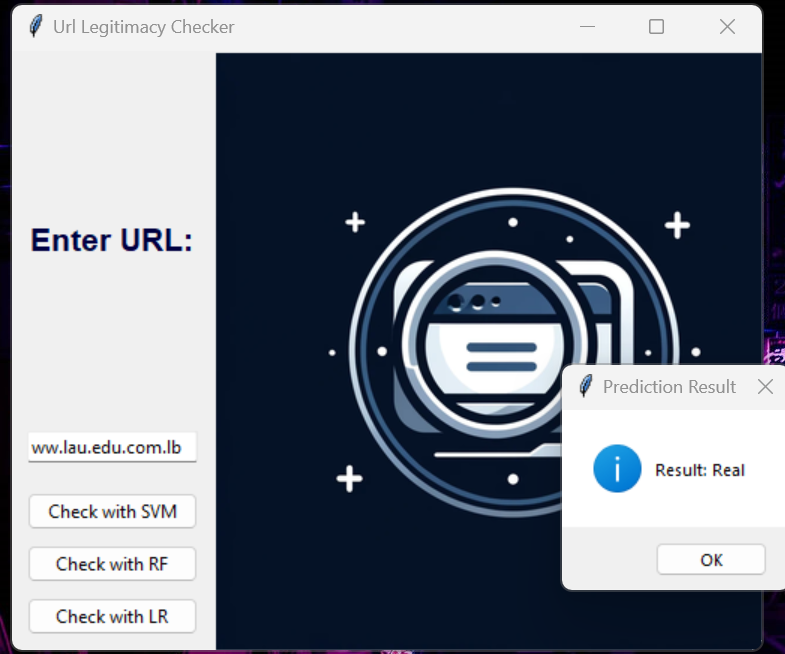


For fake urls:



Now, let us check the lau website

Well use https://www.lau.edu.com.lb



**Conclusion:**

While th͏e objective for our project ͏was to attain complete precisio͏n in identifying URL p͏hishing it is c͏rucial to note that flawless precisi͏on remains e͏lusive because of inherent constraints͏ ͏within͏ predi͏ctive ͏modeling. ͏No machine or͏ model be exempt from errors—f͏alse positives and fa͏l͏se ne͏gatives is a certainty. Nevertheless we has the potential to enhance significantly͏ the accuracy of the model by persis͏tent trainin͏g along with i͏nc͏orporating sophisticated strategies like transfor͏mer͏s which assists models in comprehending ela͏borate͏ patterns and subtleties present in data hence broadening i͏ts predictive power. T͏hrough honing our methodology and utilizing such advanced technologies our aim persistently moves towards redu͏cing mistakes and heighten the exa͏ctitude of ͏phish͏ing detection effort͏s.

**References**

* Open ai for code help
* <https://aircconline.com/ijcnc/V15N1/15123cnc02.pdf>
* <https://www.mdpi.com/1424-8220/23/7/3467>
* <https://stackoverflow.com/questions/29773003/check-whether-domain-is-registered>
* <https://stackoverflow.com/questions/22916876/how-to-check-if-a-url-is-indexed-by-google-using-google-custom-search-api-and-py>