Project Work

Group: SIS-2208

Student: Bekzadayeva Kymbat

Topic: Email Phishing URL Detection

The goal: To classify URLs as safe or malicious using ML features.

Introduction

Phishing is a common type of cyberattack in which attackers attempt to deceive users into revealing sensitive information by impersonating legitimate online services. This is typically achieved through fraudulent websites and deceptive URLs that closely resemble trusted resources. Due to the rapid growth of online services, phishing has become one of the most widespread threats in modern cybersecurity.

Traditional phishing detection methods are usually based on blacklists and manually defined rules. While such approaches can block known threats, they are ineffective against newly generated phishing URLs, which are constantly changing. As a result, there is a growing need for adaptive and automated detection mechanisms.

Machine learning provides an effective solution to this problem by enabling systems to learn patterns from data and generalize to previously unseen URLs. By analyzing structural characteristics of URLs, machine learning models can identify phishing attempts without relying on webpage content or external reputation services.

In this project, a machine learning–based approach is proposed for detecting phishing websites using URL-level information. The work covers the complete pipeline, including data preprocessing, feature extraction, model training, evaluation, and deployment in a practical web-based application.

Project Objectives

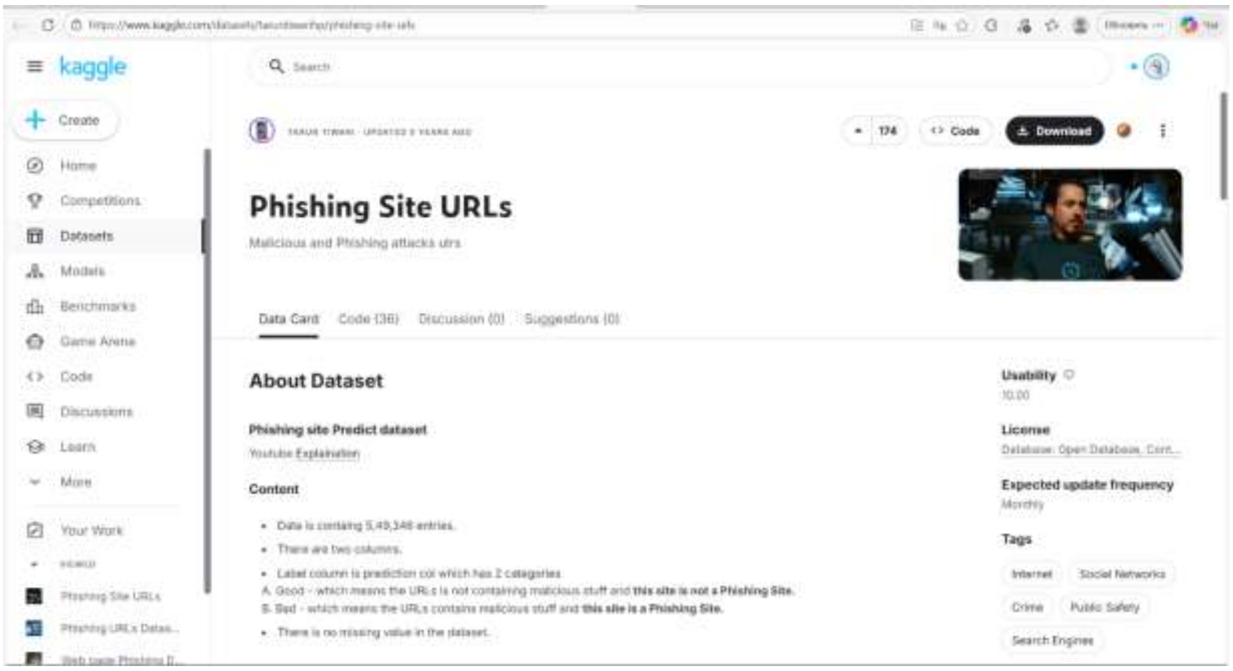
The main objectives of this project are:

- to study the phishing problem and its importance in modern cybersecurity;
- to analyze URL-based data for phishing detection;
- to apply machine learning techniques for classifying URLs as phishing or legitimate;

- to evaluate the performance of the proposed model using standard metrics;
- to demonstrate the practical application of the trained model through a simple web-based interface.

Dataset Description

The dataset used in this project was obtained from the public Kaggle repository **“Phishing Site URLs”**, published by Tarun Tiwari. It is designed for phishing URL classification and contains a large collection of malicious and legitimate website links.



Phishing Site URLs

Malicious and Phishing attacks urls

About Dataset

Phishing site Predict dataset

Youtube Explaination

Content

- Data is containg 5,49,346 entries.
- There are two columns.
- Label column is prediction col which has 2 categories
A. Good - which means the URLs is not containing malicious stuff and **this site is not a Phishing Site.**
B. Bad - which means the URLs contains malicious stuff and **this site is a Phishing Site.**
- There is no missing value in the dataset.

Usability 10.00

License Database: Open Database, Cont...

Expected update frequency Monthly

Tags Internet, Social Networks, Crime, Public Safety, Search Engines

The original dataset consists of **549,346 URL records** with two columns: **URL**, representing the web address, and **Label**, indicating whether the URL is legitimate (*Good*) or phishing (*Bad*). After removing duplicate entries during preprocessing, the final dataset used for training contained **507,195 unique URLs**.

The dataset does not contain missing values, which simplifies preprocessing and improves data consistency. All URLs are publicly available and anonymized, and no personal or sensitive information is included, making the dataset suitable for academic research.

Due to its large scale and diversity of URL structures, the dataset provides a realistic foundation for phishing detection. The variety of legitimate and malicious URLs enables machine learning models to learn meaningful patterns and generalize effectively to previously unseen data.

Label Encoding

Since machine learning models require numerical target values, the original categorical labels were converted into binary format during preprocessing. In this project, phishing URLs (*Bad*) were encoded as **1**, while legitimate URLs (*Good*) were encoded as **0**, enabling supervised binary classification.

Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) was conducted to better understand the structure, distribution, and characteristics of the phishing URL dataset prior to model training. The main objectives of this stage were to analyze class distribution, examine basic statistical properties of the URLs, and identify structural differences between phishing and legitimate samples.

Class Distribution Analysis

The dataset consists of two classes: legitimate URLs and phishing URLs. After preprocessing and duplicate removal, legitimate URLs represent the majority of the dataset, while phishing URLs form a smaller but still substantial portion. This moderate class imbalance reflects real-world conditions, where legitimate websites significantly outnumber malicious ones. Such imbalance motivates the use of evaluation metrics beyond simple accuracy and justifies the application of class weighting during model training.

URL Length Characteristics

One of the most informative structural properties examined during EDA is the total length of URLs. Empirical observation shows that phishing URLs tend to be significantly longer than legitimate ones. This is commonly due to attackers embedding misleading paths, excessive subdomains, random strings, or tracking parameters in order to obscure the true destination of the link.

Legitimate URLs, in contrast, are typically shorter and more structured, often following consistent naming conventions. This difference in length distribution supports the hypothesis that URL length is a strong discriminative signal for phishing detection.

Structural Patterns in URLs

Further qualitative analysis reveals that phishing URLs more frequently contain:

- a higher number of digits,
- an increased presence of special characters such as “-”, “_”, “@”, and “=”,
- deeper path hierarchies and longer query strings.

These patterns are consistent with common obfuscation techniques used by attackers to imitate trusted domains or to bypass basic detection mechanisms. Legitimate URLs, on the other hand, usually contain fewer random characters and follow cleaner structural formats.

Implications for Feature Engineering

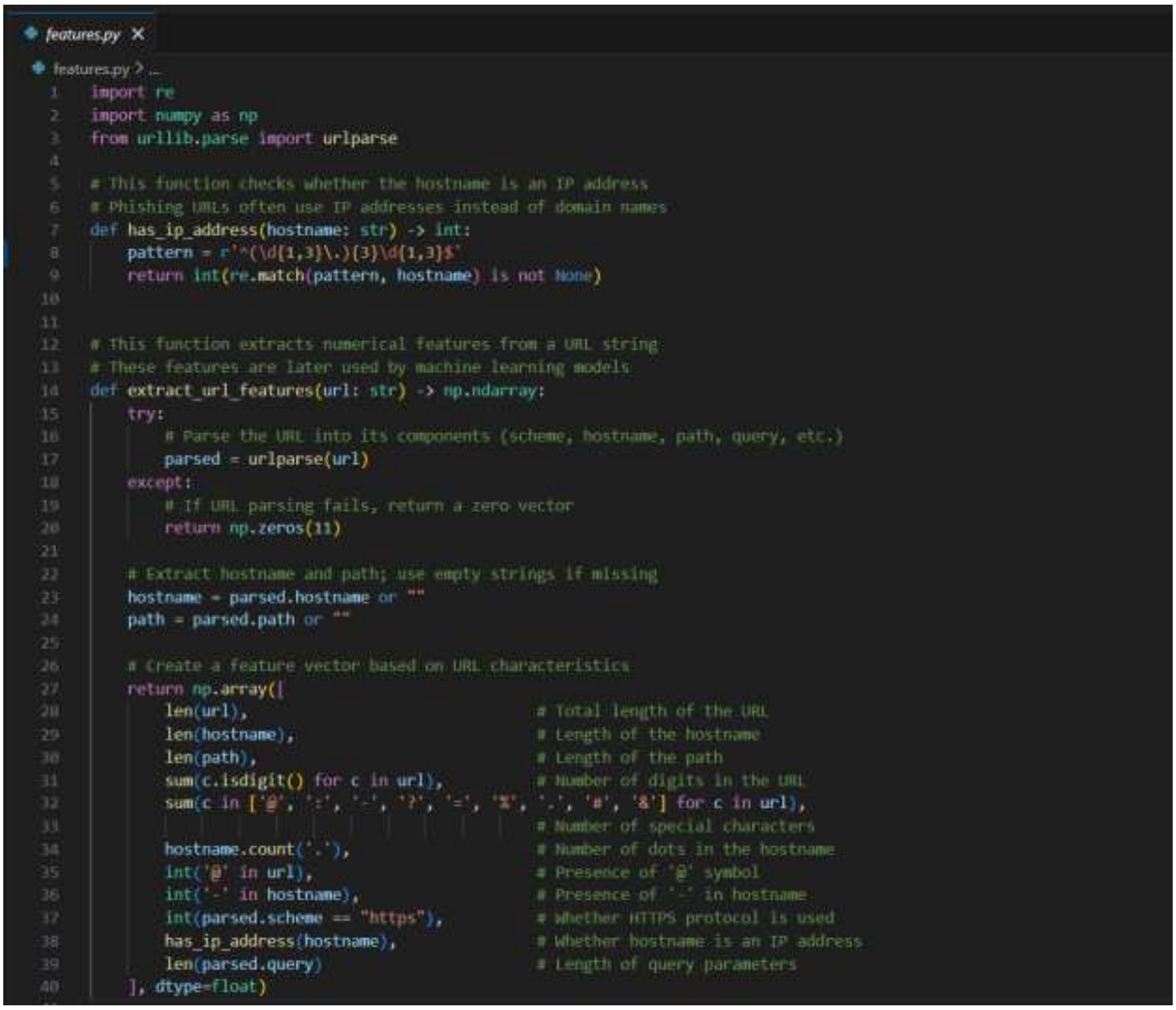
The observed structural differences between phishing and legitimate URLs confirm that URL-level lexical features provide meaningful information for classification. These findings justify the choice of character-level feature extraction methods rather than word-based tokenization, as URLs do not follow natural language grammar and often contain non-alphabetic patterns.

The EDA results directly support the use of character-level TF-IDF n-grams, as such representations are capable of capturing subtle structural variations, obfuscation patterns, and uncommon character sequences frequently associated with phishing attacks.

Practical Part

As part of this project, a lexical feature extraction module was implemented in the file features.py. The purpose of this module is to transform a raw URL string into a fixed-length numerical feature vector suitable for machine learning classification.

The URL is first parsed using the urlparse function from the Python standard library to extract its main components, including the protocol, hostname, path, and query parameters. If a URL cannot be parsed correctly, a zero feature vector is returned to ensure system robustness and prevent runtime errors.


features.py

features.py > ...

```python
import re
import numpy as np
from urllib.parse import urlparse

# This function checks whether the hostname is an IP address
# Phishing URLs often use IP addresses instead of domain names
def has_ip_address(hostname: str) -> int:
    pattern = r'^(\d{1,3}\.){3}\d{1,3}$'
    return int(re.match(pattern, hostname) is not None)


# This function extracts numerical features from a URL string
# These features are later used by machine learning models
def extract_url_features(url: str) -> np.ndarray:
    try:
        # Parse the URL into its components (scheme, hostname, path, query, etc.)
        parsed = urlparse(url)
    except:
        # If URL parsing fails, return a zero vector
        return np.zeros(11)

    # Extract hostname and path; use empty strings if missing
    hostname = parsed.hostname or ""
    path = parsed.path or ""

    # Create a feature vector based on URL characteristics
    return np.array([
        len(url),                                  # Total length of the URL
        len(hostname),                             # Length of the hostname
        len(path),                                 # Length of the path
        sum(c.isdigit() for c in url),             # Number of digits in the URL
        sum(c in ['@', ':', '-', '?', '=', '%', '.', '#', '&'] for c in url),
                                                   # Number of special characters
        hostname.count('.'),                       # Number of dots in the hostname
        int('@' in url),                           # Presence of '@' symbol
        int('-' in hostname),                      # Presence of '-' in hostname
        int(parsed.scheme == "https"),             # Whether HTTPS protocol is used
        has_ip_address(hostname),                  # Whether hostname is an IP address
        len(parsed.query)                          # Length of query parameters
    ], dtype=float)
```

Extracted URL Features

For each URL, a vector of 11 numerical lexical features is extracted:

- total URL length
- hostname length
- path length
- number of digits
- number of special characters
- number of dots in the hostname
- presence of the @ symbol
- presence of hyphens in the hostname
- use of HTTPS
- use of an IP address instead of a domain name

- length of the query string

These features reflect common phishing techniques such as URL obfuscation, excessive length, misleading subdomains, and suspicious character usage. All extracted features are converted into numerical format and combined into a single fixed-length feature vector.

Feature Selection Rationale

The selected lexical features are based on common phishing patterns reported in cybersecurity research. A key advantage of this approach is interpretability, as each feature has a clear structural meaning. Additionally, all features can be extracted directly from the URL without relying on external services or blacklists, enabling fast and lightweight detection.

Role of the Lexical Features Module

The features.py module demonstrates an alternative phishing detection approach based on explicit URL structure analysis. Although the final system focuses on TF-IDF–based representation, this module highlights the feasibility of using classical lexical features for phishing URL classification.

Feature Engineering: TF-IDF Representation

In this project, I applied the TF-IDF (Term Frequency – Inverse Document Frequency) technique combined with character-level n-grams to represent URLs in a numerical form suitable for machine learning models. Feature engineering plays a critical role in phishing detection, as the quality of extracted features directly affects the model’s ability to distinguish between legitimate and phishing URLs.

The main objective of this stage is to convert raw URL strings into high-dimensional numerical vectors that preserve structural patterns commonly associated with phishing behavior.

Feature Engineering: TF-IDF Representation

To represent URLs in numerical form, a TF-IDF (Term Frequency–Inverse Document Frequency) feature extraction technique was applied using character-level n-grams. Feature engineering is a critical stage in phishing detection, as the quality of extracted features directly affects classification performance.

TF-IDF assigns weights to character sequences based on their frequency within individual URLs and their distribution across the dataset. This allows the model to emphasize distinctive patterns commonly associated with phishing URLs while reducing the influence of common and non-informative sequences.

Character-level n-grams were chosen instead of word-level tokens because URLs do not follow natural language rules and often contain digits, symbols, and obfuscated strings. This representation enables the model to capture fine-grained structural patterns that are typical for phishing attacks.

In this project, character n-grams with lengths ranging from 3 to 6 characters were used. Shorter n-grams provide limited information, while longer n-grams significantly increase dimensionality and computational cost. The selected range offers a balance between expressiveness and efficiency.

Overall, TF-IDF with character-level n-grams provides a robust and scalable representation for URL-based phishing detection and generalizes well to previously unseen URLs.

❖ Model Training and Evaluation (train_model.py)

In this project, I implemented the complete model training and evaluation pipeline in the train_model.py module. This stage is responsible for preparing the dataset, extracting features, training the classification model, evaluating its performance, and saving the trained components for later deployment in the web application.

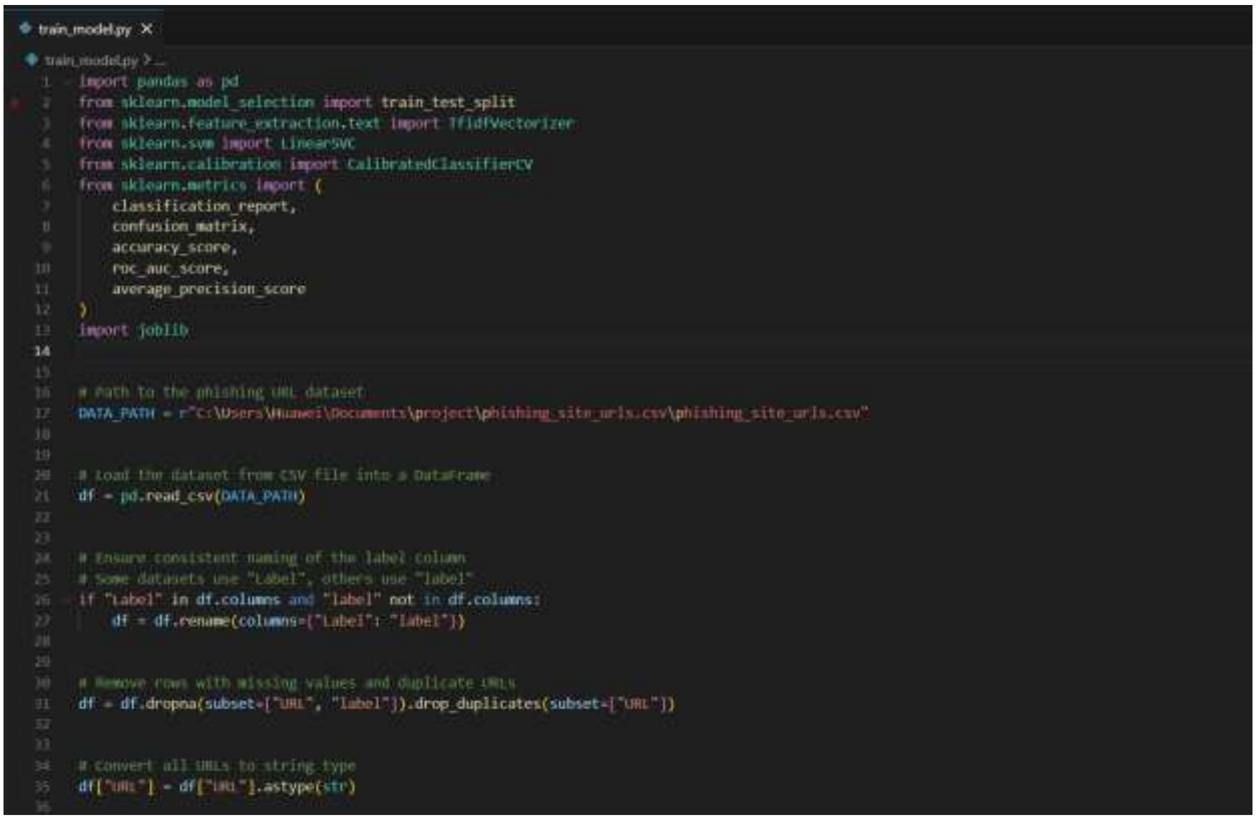
Dataset Loading and Preprocessing

First, I loaded the phishing URL dataset from a CSV file using the pandas library. The dataset contains URLs along with their corresponding labels indicating whether a URL is phishing or legitimate.

To ensure data consistency, I performed the following preprocessing steps:

- standardized the label column name;
- removed rows with missing values in critical fields;
- removed duplicate URLs to prevent biased evaluation;
- converted all URLs to string format.

These steps help improve data quality and ensure that the model is trained on clean and consistent input data.



```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
from sklearn.calibration import CalibratedClassifierCV
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    accuracy_score,
    roc_auc_score,
    average_precision_score
)
import joblib


# Path to the phishing URL dataset
DATA_PATH = r"C:\Users\Huawei\Documents\project\phishing_site_urls.csv\phishing_site_urls.csv"


# Load the dataset from CSV file into a DataFrame
df = pd.read_csv(DATA_PATH)


# Ensure consistent naming of the label column
# Some datasets use "Label", others use "label"
if "Label" in df.columns and "label" not in df.columns:
    df = df.rename(columns={"Label": "label"})


# Remove rows with missing values and duplicate URLs
df = df.dropna(subset=["URL", "label"]).drop_duplicates(subset=["URL"])


# Convert all URLs to string type
df["URL"] = df["URL"].astype(str)
```

Label Encoding

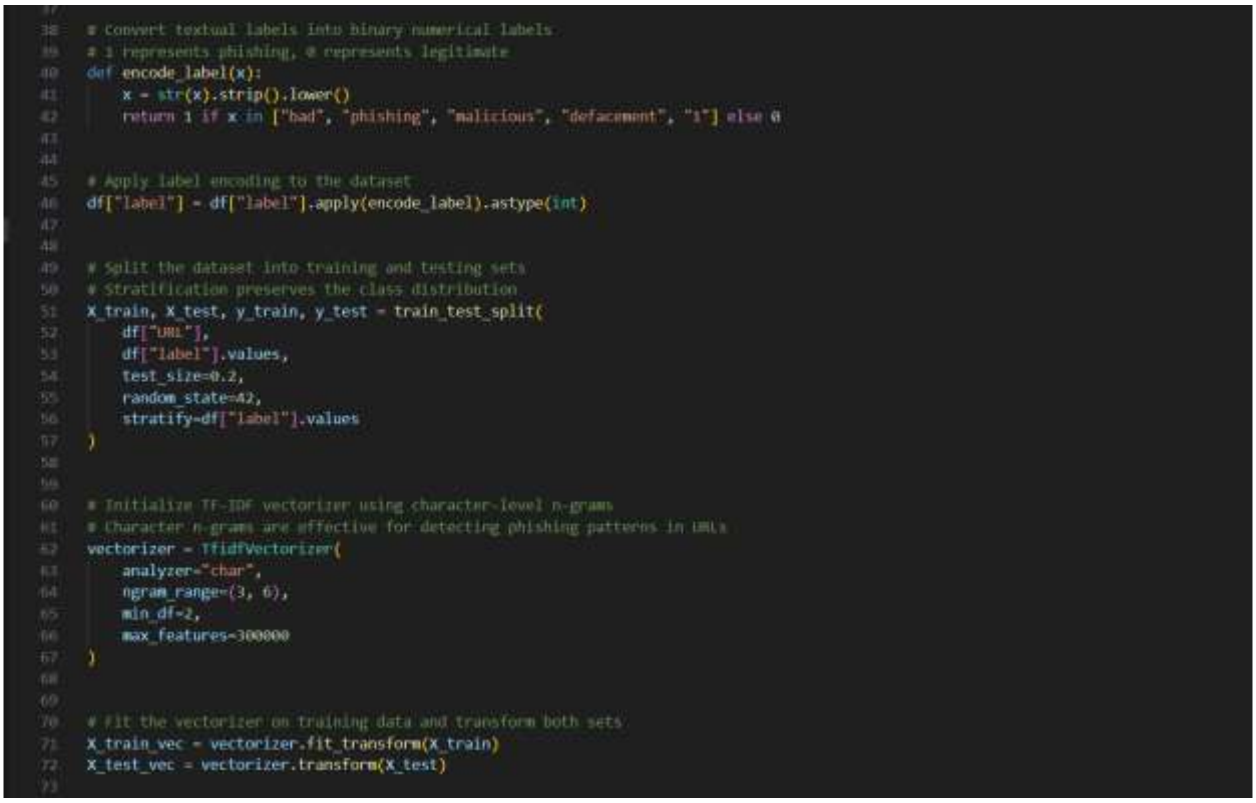
Since machine learning models require numerical labels, I converted the original textual labels into binary values, where 1 represents a phishing URL and 0 represents a legitimate URL. This encoding enables binary classification and ensures compatibility with standard evaluation metrics.

Train–Test Split

To evaluate the model's generalization ability, I split the dataset into training and testing sets using an 80/20 ratio with stratification to preserve the original class distribution. A fixed random seed was used to ensure reproducibility and to enable fair evaluation without introducing data leakage.

TF-IDF Feature Extraction

After splitting the dataset, I transformed the URLs into numerical feature vectors using TF-IDF with character-level n-grams (3–6). The TF-IDF vectorizer was fitted only on the training data and then applied to both training and test sets, ensuring that no information from the test set leaked into the training process.



```python
# Convert textual labels into binary numerical labels
# 1 represents phishing, 0 represents legitimate
def encode_label(x):
    x = str(x).strip().lower()
    return 1 if x in ["bad", "phishing", "malicious", "defacement", "1"] else 0


# Apply label encoding to the dataset
df["label"] = df["label"].apply(encode_label).astype(int)


# Split the dataset into training and testing sets
# Stratification preserves the class distribution
X_train, X_test, y_train, y_test = train_test_split(
    df["URL"],
    df["label"].values,
    test_size=0.2,
    random_state=42,
    stratify=df["label"].values
)


# Initialize TF-IDF vectorizer using character-level n-grams
# Character n-grams are effective for detecting phishing patterns in URLs
vectorizer = TfidfVectorizer(
    analyzer="char",
    ngram_range=(3, 6),
    min_df=2,
    max_features=300000
)


# Fit the vectorizer on training data and transform both sets
X_train_vec = vectorizer.fit_transform(X_train)
X_test_vec = vectorizer.transform(X_test)
```

Model Selection and Training

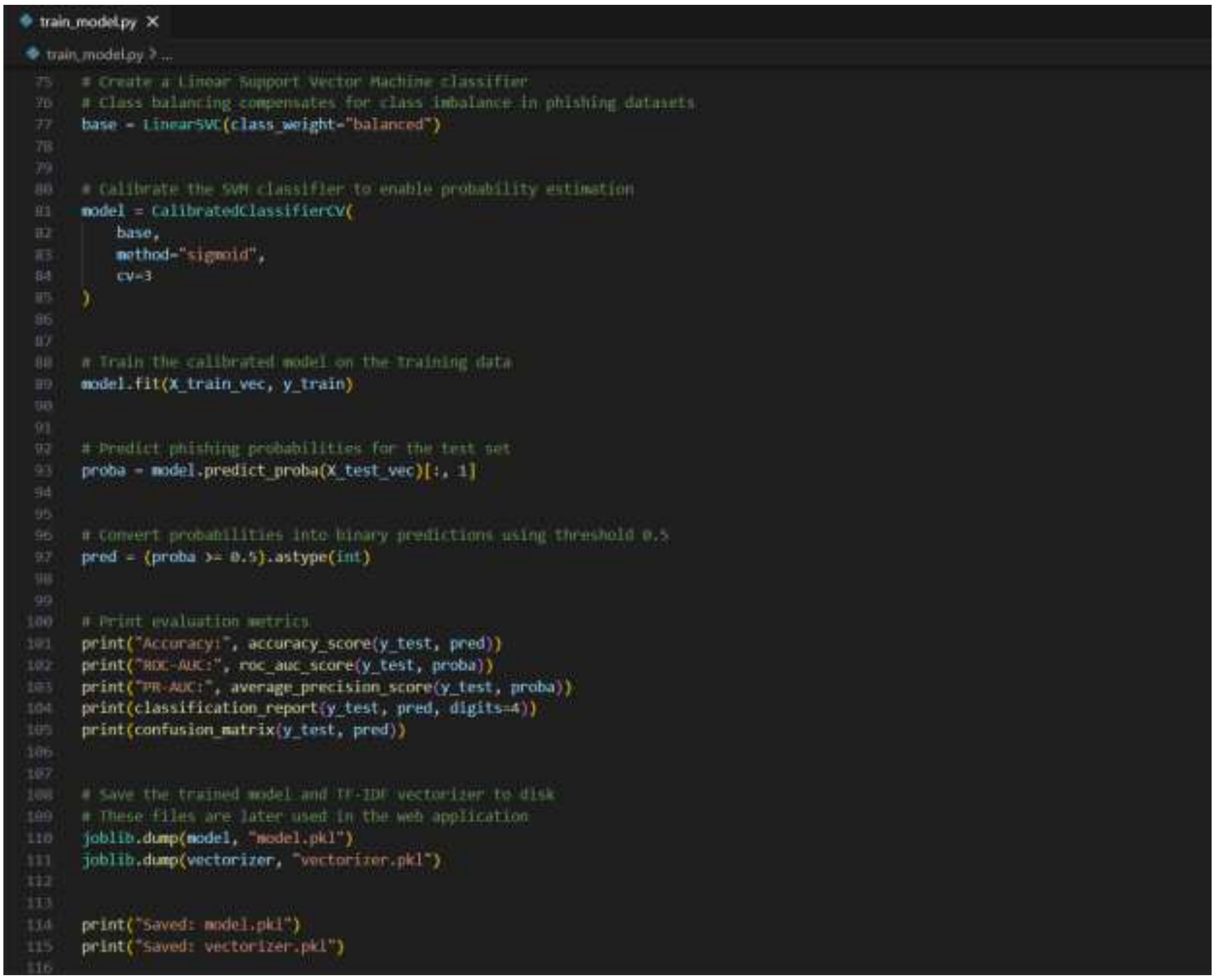
For classification, I selected a Linear Support Vector Machine (LinearSVC) as the base model, as it is well suited for high-dimensional sparse data such as TF-IDF feature vectors. To address class imbalance commonly present in phishing datasets, I applied the parameter class_weight="balanced", which adjusts the loss function to give equal importance to both classes.

Since LinearSVC does not natively provide probability estimates, I wrapped the classifier using CalibratedClassifierCV with sigmoid calibration. This enables the model to output well-calibrated probability scores, which are later used for threshold-based decision making in the web application. The calibrated model was trained exclusively on the training feature vectors.

Prediction and Evaluation Metrics

After training, I evaluated the model on the test set. The model outputs phishing probabilities, which were converted into binary predictions using a decision threshold of 0.5. To assess performance, I computed standard evaluation metrics, including accuracy, ROC-AUC, PR-AUC, precision, recall, F1-score, and the confusion matrix.

The results indicate high classification effectiveness while maintaining proper evaluation methodology and avoiding data leakage.


train_model.py
train_model.py > ...

```python
# Create a Linear Support Vector Machine classifier
# Class balancing compensates for class imbalance in phishing datasets
base = LinearSVC(class_weight="balanced")


# Calibrate the SVM classifier to enable probability estimation
model = CalibratedClassifierCV(
    base,
    method="sigmoid",
    cv=3
)


# Train the calibrated model on the training data
model.fit(X_train_vec, y_train)


# Predict phishing probabilities for the test set
proba = model.predict_proba(X_test_vec)[:, 1]


# Convert probabilities into binary predictions using threshold 0.5
pred = (proba >= 0.5).astype(int)


# Print evaluation metrics
print("Accuracy:", accuracy_score(y_test, pred))
print("ROC-AUC:", roc_auc_score(y_test, proba))
print("PR-AUC:", average_precision_score(y_test, proba))
print(classification_report(y_test, pred, digits=4))
print(confusion_matrix(y_test, pred))


# Save the trained model and TF-IDF vectorizer to disk
# These files are later used in the web application
joblib.dump(model, "model.pkl")
joblib.dump(vectorizer, "vectorizer.pkl")


print("Saved: model.pkl")
print("Saved: vectorizer.pkl")
```

Model Persistence

Finally, I saved the trained model and the TF-IDF vectorizer to disk using the joblib library. These files are later loaded by the Flask web application for real-time phishing detection.

This separation between training and deployment ensures modularity and allows the system to scale or retrain models independently.

Summary of the Training Process

In this module, I implemented a complete and robust training pipeline that includes data preprocessing, feature extraction, model training, evaluation, and persistence. The chosen approach balances performance, interpretability, and practical deployment considerations, making it suitable for real-world phishing detection tasks.

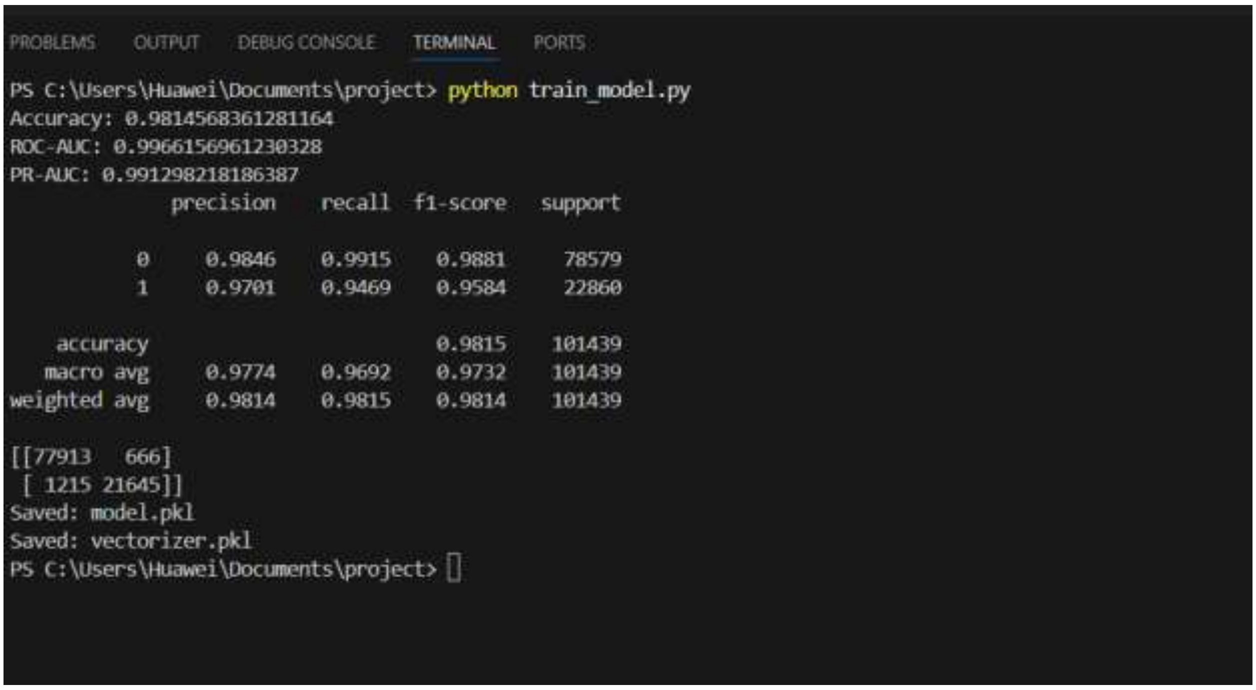
Model Evaluation Results

After training the model, I evaluated its performance on the held-out test set using multiple evaluation metrics to obtain a comprehensive assessment of classification quality.

The model achieved an **accuracy of 98.15%**, indicating a high overall correctness of predictions. In addition, the **ROC-AUC score of 0.9966** demonstrates excellent discriminative ability between phishing and legitimate URLs. The **PR-AUC score of 0.9913** further confirms strong performance under class imbalance, which is common in phishing detection tasks.

The classification report shows balanced precision and recall for both classes. The confusion matrix indicates a low number of misclassifications, with most phishing and legitimate URLs correctly identified.

Importantly, these results were obtained without data leakage, as feature extraction and model training were performed exclusively on the training set, while evaluation was conducted on unseen test data. Therefore, the reported metrics reflect the model's true generalization capability rather than overfitting.

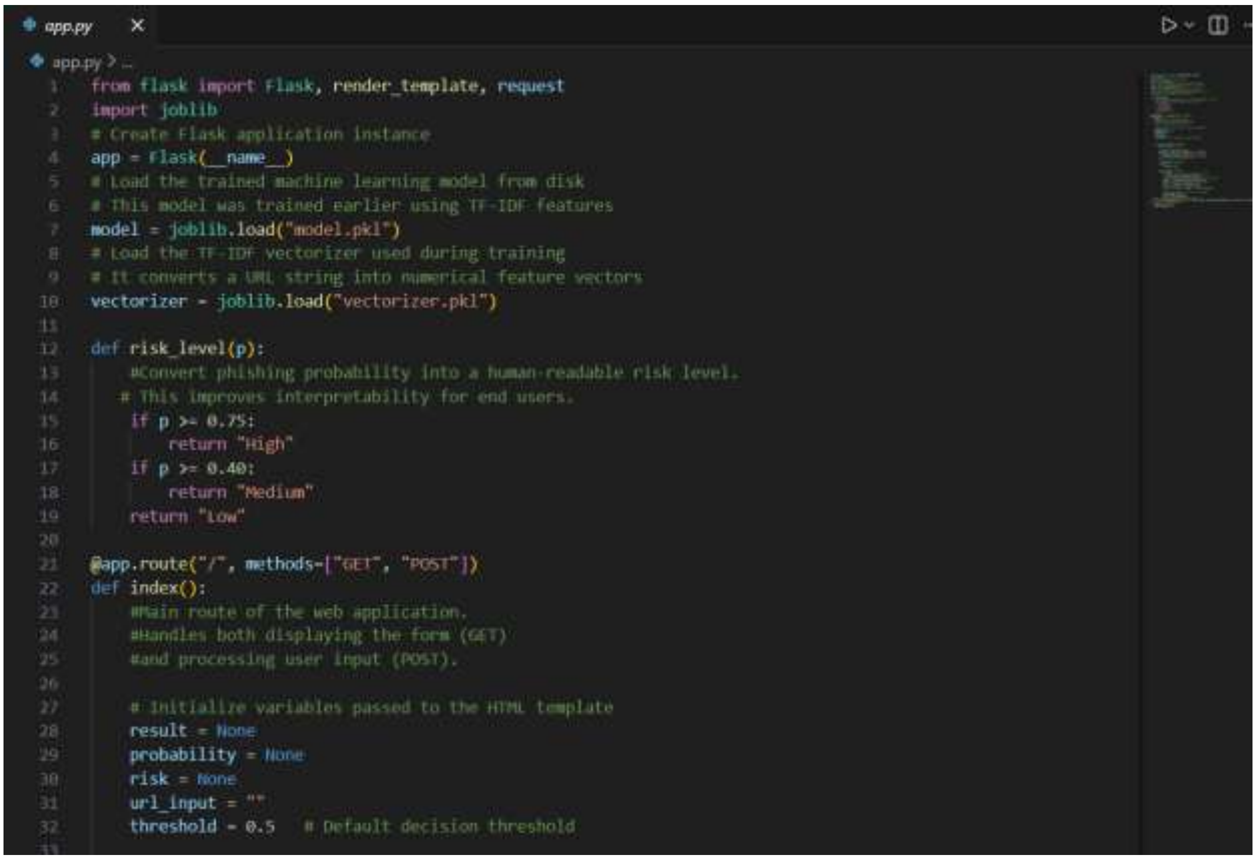


```
PS C:\Users\Huawei\Documents\project> python train_model.py
Accuracy: 0.9814568361281164
ROC-AUC: 0.9966156961230328
PR-AUC: 0.991298218186387
              precision    recall  f1-score   support

           0     0.9846    0.9915    0.9881     78579
           1     0.9701    0.9469    0.9584     22860

    accuracy                         0.9815    101439
   macro avg     0.9774    0.9692    0.9732    101439
weighted avg     0.9814    0.9815    0.9814    101439

[[77913   666]
 [ 1215 21645]]
Saved: model.pkl
Saved: vectorizer.pkl
PS C:\Users\Huawei\Documents\project> 
```

Web Application and Model Deployment (app.py)

After training and evaluating the machine learning model, I implemented a web-based phishing URL detection interface using the Flask framework. The application logic is contained in the app.py module and enables real-time interaction with the trained model. The main purpose of this component is to allow users to submit a URL, receive a phishing prediction, and obtain an interpretable risk assessment based on model output probabilities.

At application startup, I initialized the Flask backend and loaded the trained classification model along with the TF-IDF vectorizer from disk using the joblib library. Loading these components once at initialization ensures efficient inference and avoids retraining the model for each user request, enabling fast and scalable real-time detection.


app.py
app.py > ...

```python
from flask import Flask, render_template, request
import joblib
# Create Flask application instance
app = Flask(__name__)
# Load the trained machine learning model from disk
# This model was trained earlier using TF-IDF features
model = joblib.load("model.pkl")
# Load the TF-IDF vectorizer used during training
# It converts a URL string into numerical feature vectors
vectorizer = joblib.load("vectorizer.pkl")

def risk_level(p):
    #Convert phishing probability into a human-readable risk level.
   # This improves interpretability for end users.
    if p >= 0.75:
        return "High"
    if p >= 0.40:
        return "Medium"
    return "Low"

@app.route("/", methods=["GET", "POST"])
def index():
    #Main route of the web application.
    #Handles both displaying the form (GET)
    #and processing user input (POST).

    # Initialize variables passed to the HTML template
    result = None
    probability = None
    risk = None
    url_input = ""
    threshold = 0.5   # Default decision threshold
```

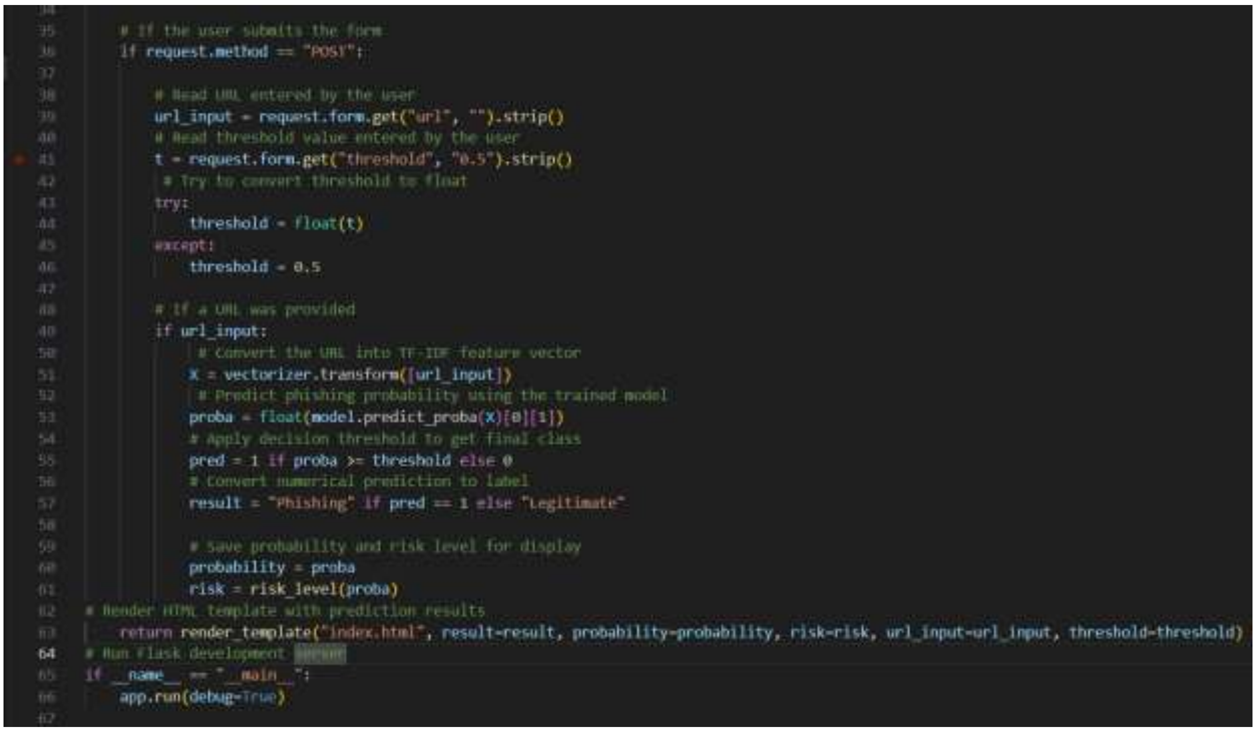
Risk Level Interpretation, Application Logic, and Inference

To improve result interpretability for end users, I implemented a helper function that converts the predicted phishing probability into a human-readable risk level. Based on predefined probability thresholds, the system classifies URLs as low, medium, or high risk, allowing users to understand the model's confidence without relying solely on numerical values.

The core application logic is implemented in the root route of the Flask application, which supports both GET and POST HTTP methods. GET requests are used to display the input form, while POST requests handle user-submitted data and trigger model inference. At initialization, default values are assigned to all variables passed to the HTML template, including the decision threshold, predicted class, probability score, and risk level.

When a user submits a URL, the application reads the input along with an optional custom decision threshold. The threshold value is validated and converted to a floating-point number, and if invalid input is provided, a default threshold of 0.5 is applied to ensure robustness and prevent runtime errors.

If a valid URL is received, the application transforms it into a TF-IDF feature vector using the loaded vectorizer and applies the trained model to obtain a phishing probability. This probability is then compared against the selected threshold to generate a binary classification result (phishing or legitimate), after which the corresponding risk level is determined. This design enables flexible decision making by allowing users to adjust the classification sensitivity based on their risk tolerance.



```python
    # If the user submits the form
    if request.method == "POST":

        # Read URL entered by the user
        url_input = request.form.get("url", "").strip()
        # Read threshold value entered by the user
        t = request.form.get("threshold", "0.5").strip()
        # Try to convert threshold to float
        try:
            threshold = float(t)
        except:
            threshold = 0.5

        # If a URL was provided
        if url_input:
            # Convert the URL into TF-IDF feature vector
            X = vectorizer.transform([url_input])
            # Predict phishing probability using the trained model
            proba = float(model.predict_proba(X)[0][1])
            # Apply decision threshold to get final class
            pred = 1 if proba >= threshold else 0
            # Convert numerical prediction to label
            result = "Phishing" if pred == 1 else "Legitimate"

            # Save probability and risk level for display
            probability = proba
            risk = risk_level(proba)
# Render HTML template with prediction results
    return render_template("index.html", result=result, probability=probability, risk=risk, url_input=url_input, threshold=threshold)
# Run Flask development server
if __name__ == "__main__":
    app.run(debug=True)
```

Rendering Results and Application Execution

After model inference, the application renders the index.html template and passes the predicted class label, phishing probability, risk level, submitted URL, and selected decision threshold to the frontend. This clear separation between backend logic and frontend presentation ensures clean application architecture and simplifies future modifications and maintenance.

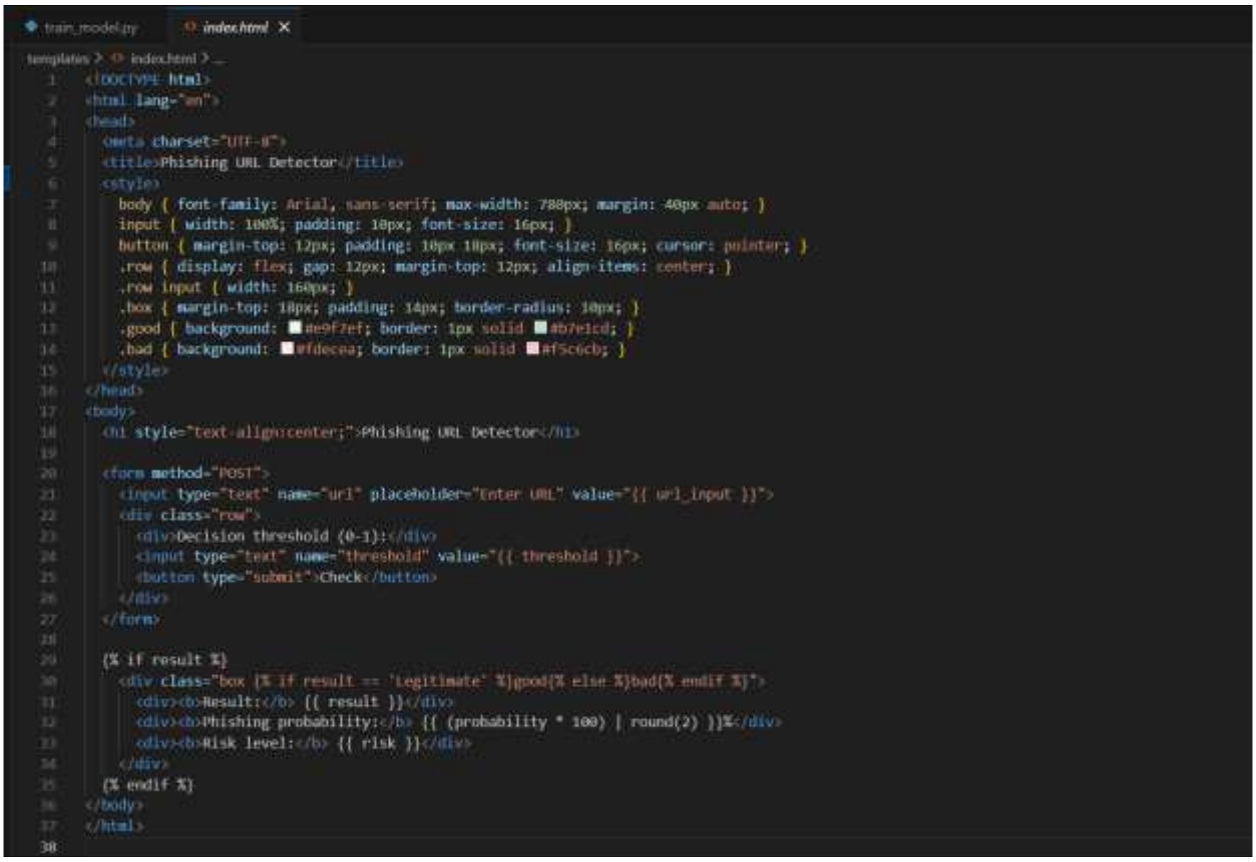
For development and testing purposes, the Flask application is executed in debug mode, which allows easier troubleshooting during implementation and demonstration of the system.

Overall, this deployment stage integrates the trained machine learning model into a user-friendly web application that supports real-time phishing detection, probability-based risk assessment, and adjustable decision thresholds. This design makes the system both practical and interpretable for end users.

User Interface Design (index.html)

To provide a user-friendly interface for the phishing URL detection system, I implemented a simple and intuitive web interface using HTML with embedded CSS. The index.html file serves as the frontend component of the system and is dynamically rendered using the Flask templating engine.

The interface follows a minimalistic design approach to ensure clarity and ease of use. It includes a clearly defined title, an input form for URL submission and threshold adjustment, and a results section that appears only after a prediction is made. Basic embedded styling is used to improve readability and visual structure without relying on external libraries.



```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Phishing URL Detector</title>
  <style>
    body { font-family: Arial, sans-serif; max-width: 780px; margin: 40px auto; }
    input { width: 100%; padding: 10px; font-size: 16px; }
    button { margin-top: 12px; padding: 10px 18px; font-size: 16px; cursor: pointer; }
    .row { display: flex; gap: 12px; margin-top: 12px; align-items: center; }
    .row input { width: 160px; }
    .box { margin-top: 18px; padding: 14px; border-radius: 10px; }
    .good { background: #e9f7ef; border: 1px solid #b7e1cd; }
    .bad { background: #fdecea; border: 1px solid #f5c6cb; }
  </style>
</head>
<body>
  <h1 style="text-align:center;">Phishing URL Detector</h1>

  <form method="POST">
    <input type="text" name="url" placeholder="Enter URL" value="{{ url_input }}">
    <div class="row">
      <div>Decision threshold (0-1):</div>
      <input type="text" name="threshold" value="{{ threshold }}">
      <button type="submit">Check</button>
    </div>
  </form>

  {% if result %}
    <div class="box {% if result == 'Legitimate' %}good{% else %}bad{% endif %}">
      <div><b>Result:</b> {{ result }}</div>
      <div><b>Phishing probability:</b> {{ (probability * 100) | round(2) }}%</div>
      <div><b>Risk level:</b> {{ risk }}</div>
    </div>
  {% endif %}
</body>
</html>
```

Input Form, Dynamic Rendering, and User Interface Integration

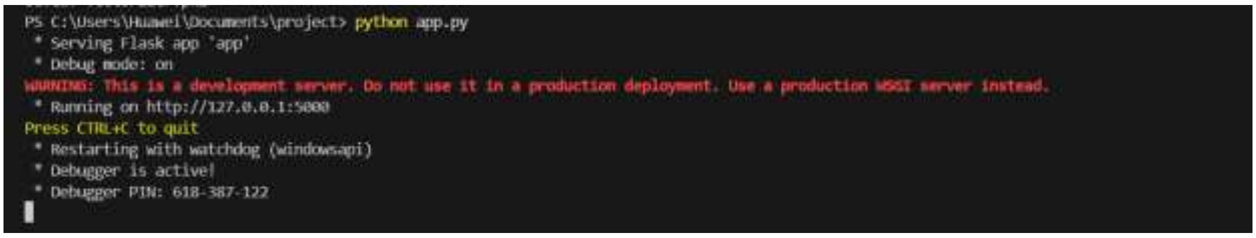
The user interface provides a simple input form that allows users to enter a URL for analysis and optionally specify a custom decision threshold between 0 and 1. The URL input field occupies the full width of the page, making it convenient for pasting long links, while the threshold parameter enables users to adjust the sensitivity of phishing detection according to their risk tolerance. All input values are preserved after submission using Flask template variables, which improves usability and prevents the need to re-enter data.

Prediction results are rendered dynamically using Jinja2 template logic and are displayed only after the model produces an output. The result section presents the predicted class label, phishing probability, and corresponding risk level. To enhance interpretability, visual cues are applied, with a green background indicating legitimate URLs and a red background highlighting phishing URLs. This visual distinction allows users to immediately understand the outcome of the analysis.

The frontend is tightly integrated with the Flask backend through template variables passed from app.py, which dynamically populate the submitted URL, selected threshold, prediction result, probability score, and risk level. This clear separation between presentation logic and backend processing ensures clean architecture and simplifies future UI modifications or extensions.

The interface was intentionally designed to be minimalistic and accessible for non-technical users. Clear parameter labeling, immediate feedback after submission, and interpretable outputs contribute to ease of use, making the system suitable for demonstration and educational purposes.

Overall, the index.html file completes the phishing detection system by providing a clean and interactive interface for real-time model inference. After implementing the web application logic, I launched the Flask server locally in development mode. Once started, the trained model and TF-IDF vectorizer are loaded into memory, allowing the system to process user input and perform real-time predictions without retraining. This confirms that the machine learning model is fully integrated into a functional and interactive web application.



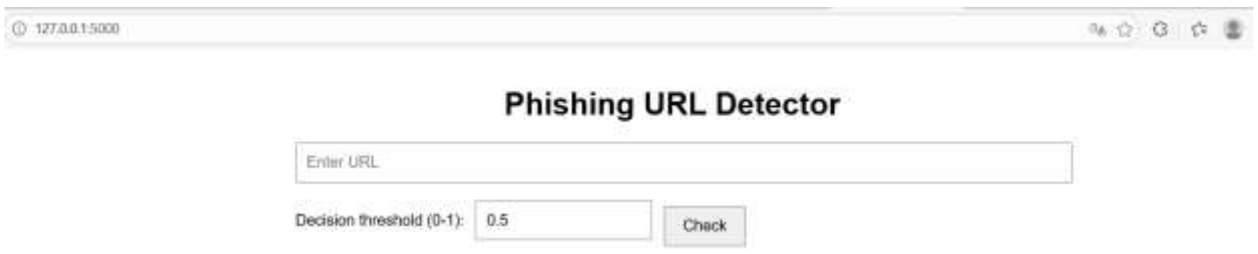
```
PS C:\Users\Huawei\Documents\project> python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with watchdog (windowsapi)
 * Debugger is active!
 * Debugger PIN: 618-387-122
```

User Interface Overview

The main page of the web application provides a simple and intuitive interface for phishing URL detection. Users can enter a URL for analysis and optionally adjust the decision threshold that controls the sensitivity of the classification.

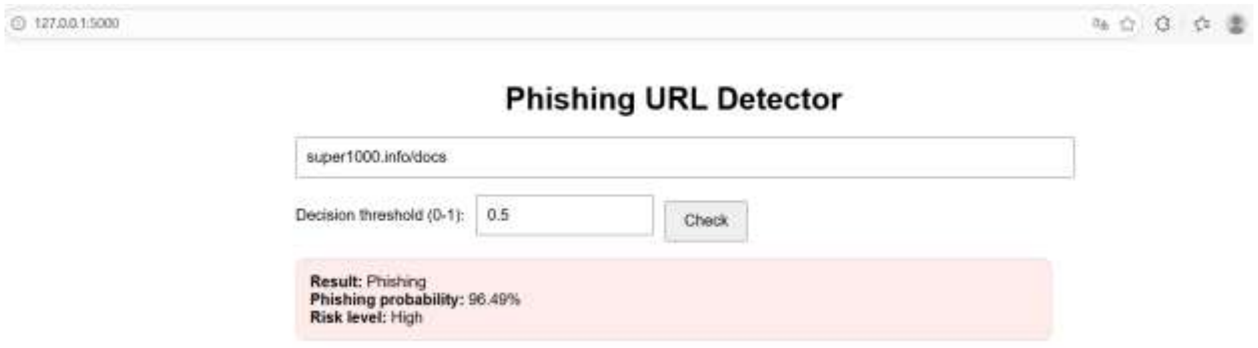
The interface follows a minimalistic design to ensure ease of use for non-technical users. By allowing threshold adjustment, the system enables users to define the probability level at which a URL is classified as phishing, making the detection process more flexible and interpretable.

This page serves as the entry point of the application and connects user input directly with the machine learning model deployed on the backend.



Decision threshold (0-1): 0.5

Figure demonstrates an example of the system's output when analyzing a suspicious URL. The model classified the URL as **Phishing** with a predicted probability of **96.49%**, corresponding to a **High risk** level. The result is clearly presented through the predicted class, probability score, and risk interpretation, with a red background visually emphasizing the potential threat. This example confirms that the trained machine learning model is successfully integrated into the web application and provides interpretable phishing detection results in real time.

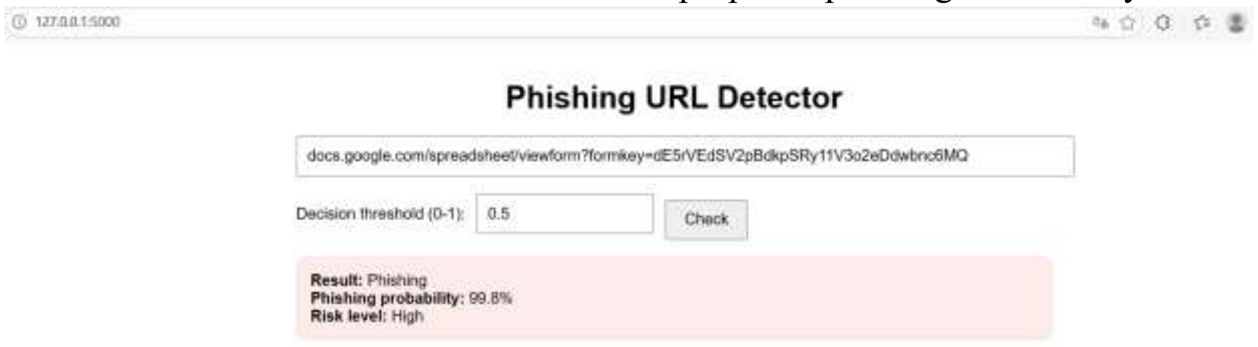


super1000.info/docs

Decision threshold (0-1): 0.5

Result: Phishing
Phishing probability: 96.49%
Risk level: High

Next figure illustrates another example of the system's output when analyzing a suspicious URL. The model classified the URL as **Phishing** with a predicted probability of **99.8%**, corresponding to a **High risk** level. The result demonstrates the model's ability to confidently identify highly suspicious URLs, with the predicted class, probability score, and risk interpretation clearly presented. Visual highlighting further emphasizes the severity of the detected threat, confirming the robustness and real-time effectiveness of the proposed phishing detection system.



Phishing URL Detector

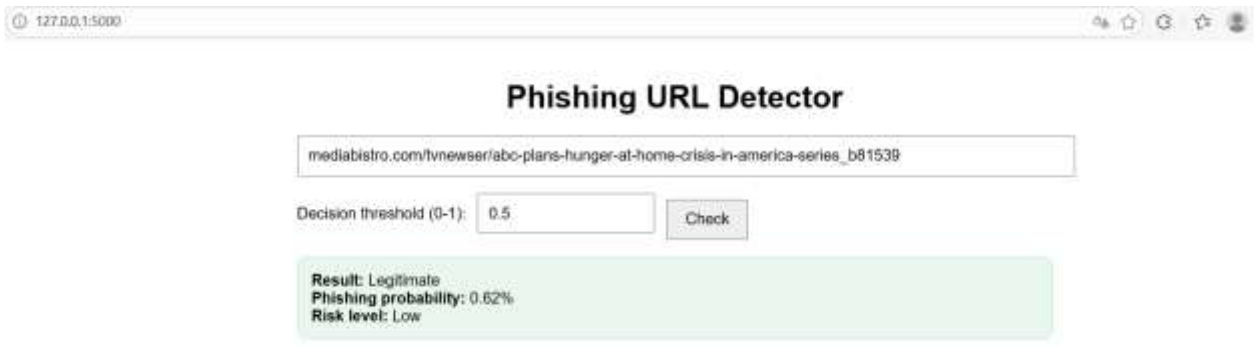
docs.google.com/spreadsheet/viewform?formkey=dE5rVEdSV2pBdkpSRy11V3o2eDdwbnc6MQ

Decision threshold (0-1): 0.5

Result: Phishing
Phishing probability: 99.8%
Risk level: High

Legitimate URL Classification Example

Figure presents an example of the system's output when analyzing a legitimate URL. The model classified the URL as **Legitimate** with a phishing probability of **0.62%**, corresponding to a **Low risk** level. The result demonstrates the model's ability to correctly identify benign URLs and avoid false positives, with a green background visually indicating a safe outcome. Together with the phishing detection examples, this confirms that the system provides balanced and reliable real-time predictions for both classes.



Phishing URL Detector

mediabistro.com/tvnewser/abc-plans-hunger-at-home-crisis-in-america-series_b81539

Decision threshold (0-1): 0.5

Result: Legitimate
Phishing probability: 0.62%
Risk level: Low

Conclusion

In this project, I developed and implemented a machine learning–based system for phishing URL detection using character-level TF-IDF feature extraction and a calibrated Linear Support Vector Machine classifier. The proposed approach effectively distinguishes between phishing and legitimate URLs based solely on URL-level information.

A complete machine learning pipeline was implemented, including data preprocessing, feature engineering, model training, evaluation, and deployment. The use of character-level TF-IDF n-grams proved to be well suited for URL analysis, enabling the model to capture structural patterns commonly associated with phishing attacks while achieving strong performance across multiple evaluation metrics without data leakage.

To demonstrate practical applicability, the trained model was successfully deployed as a Flask web application that provides real-time phishing detection with probability-based risk assessment and adjustable decision thresholds. The results confirm that machine learning techniques can be effectively applied to phishing detection using only URL-based features. Future work may involve incorporating additional features, exploring deep learning methods, or extending the system to browser-level protection.