# Phishing Detection and Prevention System

This system consists of modular Python scripts for detecting and preventing phishing URLs using machine learning and safe browsing checks. It includes feature extraction, model training (Random Forest and SVM), prediction, Google Safe Browsing API integration, and a Tkinter GUI for user interaction. The system is organized as follows:

* **feature\_extractor.py**: Parses a URL and computes lexical and host-based features (e.g. length, digit counts, special characters, IP usage)[[1]](https://medium.com/pythons-gurus/phishing-no-more-creating-a-url-classifier-with-machine-learning-a8a1b3e90a0f#:~:text=%27length_url%27%2C%20%27length_hostname%27%2C%20%27ip%27%2C%20%27nb_dots%27%2C%20%27nb_hyphens%27%2C,domain_in_brand%27%2C%20%27brand_in_subdomain%27%2C%20%27brand_in_path%27%2C%20%27suspecious_tld%27)[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no).
* **train\_model.py**: Loads a phishing URL dataset (e.g. UCI Phishing Websites[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no)), extracts features, trains a Random Forest and an SVM, evaluates their performance, and saves the models.
* **predict.py**: Loads a saved model and uses it to predict whether a new URL is phishing or legitimate.
* **api\_check.py**: Integrates Google Safe Browsing Lookup API (v4) to check if a URL is known to be malicious[[3]](https://developers.google.com/safe-browsing/v4/lookup-api#:~:text=).
* **gui.py**: A Tkinter GUI where users can enter URLs; it uses the ML model and Safe Browsing API to display a visual result (Safe vs. Phishing).
* **utils.py**: Utility functions (URL validation, logging setup, etc.).
* **requirements.txt**: Lists required Python packages.
* **Sample Dataset & Instructions**: Instructions to obtain the dataset (e.g. from UCI Phishing Websites dataset[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no)) if not included.

Below we describe each component with code. All scripts include comments and proper error handling.

## Feature Extraction Module (feature\_extractor.py)

This module defines functions to extract features from a URL. We compute **lexical features** (length, digit counts, special-character counts, etc.) and **host-based features** (e.g. whether the hostname is an IP address)[[1]](https://medium.com/pythons-gurus/phishing-no-more-creating-a-url-classifier-with-machine-learning-a8a1b3e90a0f#:~:text=%27length_url%27%2C%20%27length_hostname%27%2C%20%27ip%27%2C%20%27nb_dots%27%2C%20%27nb_hyphens%27%2C,domain_in_brand%27%2C%20%27brand_in_subdomain%27%2C%20%27brand_in_path%27%2C%20%27suspecious_tld%27)[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no). For example, features include: - url\_length, hostname\_length (lengths of the full URL and hostname)  
- Counts of digits, dots, hyphens, ?, =, &, etc. in the URL  
- Flags like has\_ip (1 if the host is an IP address) and shortening\_service (1 if a known URL shortener is used)  
- Presence of http:// or https:// tokens in the URL  
- Number of subdomains (count of . in hostname)

These features are inspired by prior work (e.g. Kaggle’s phishing dataset features[[1]](https://medium.com/pythons-gurus/phishing-no-more-creating-a-url-classifier-with-machine-learning-a8a1b3e90a0f#:~:text=%27length_url%27%2C%20%27length_hostname%27%2C%20%27ip%27%2C%20%27nb_dots%27%2C%20%27nb_hyphens%27%2C,domain_in_brand%27%2C%20%27brand_in_subdomain%27%2C%20%27brand_in_path%27%2C%20%27suspecious_tld%27) and UCI’s Phishing Websites dataset[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no)). The code below shows how to parse a URL and compute these features:

# feature\_extractor.py  
  
import re  
from urllib.parse import urlparse  
from utils import is\_valid\_url  
  
def extract\_features(url):  
 """  
 Extract lexical and host-based features from the given URL.  
 Returns a dictionary of numeric feature values.  
 """  
 # Validate URL format  
 if not is\_valid\_url(url):  
 raise ValueError(f"Invalid URL: {url}")  
  
 parsed = urlparse(url)  
 host = parsed.netloc  
 path = parsed.path  
  
 features = {}  
 # Lexical features  
 features["url\_length"] = len(url)  
 features["hostname\_length"] = len(host)  
 features["count\_digits\_url"] = sum(c.isdigit() for c in url)  
 features["count\_digits\_host"] = sum(c.isdigit() for c in host)  
 features["count\_digits\_path"] = sum(c.isdigit() for c in path)  
  
 # Special-character counts  
 specials = {  
 '?': "count\_question",  
 '&': "count\_ampersand",  
 '=': "count\_equal",  
 '-': "count\_hyphen",  
 '@': "count\_at",  
 '%': "count\_percent",  
 '.': "count\_dot",  
 '/': "count\_slash",  
 '\*': "count\_star",  
 '#': "count\_hash"  
 }  
 for char, fname in specials.items():  
 features[fname] = url.count(char)  
 features["count\_www"] = url.count("www.")  
 features["count\_dot\_com"] = url.count(".com")  
  
 # Token presence  
 features["has\_http\_token"] = 1 if "http://" in url else 0  
 features["has\_https\_token"] = 1 if "https://" in url else 0  
  
 # Host-based features  
 hostname = host.split(':')[0] # remove port if present  
 # Check if hostname is an IP address  
 features["has\_ip"] = 1 if re.match(r'^\d+\.\d+\.\d+\.\d+$', hostname) else 0  
 # Known URL shortening services (common examples)  
 shorteners = ['bit.ly','goo.gl','tinyurl.com','ow.ly','is.gd','buff.ly','adf.ly','t.co','bit.do','mcaf.ee']  
 features["shortening\_service"] = 1 if any(s in url for s in shorteners) else 0  
 # Number of subdomains (counts of '.')  
 features["count\_subdomain"] = hostname.count('.')  
  
 return features

**Logging and Validation:** We validate the URL using a helper (is\_valid\_url) and raise an error for invalid input. The utils.py module (below) provides is\_valid\_url and logging setup for consistency.

## ML Model Training (train\_model.py)

This script loads a labeled dataset of URLs (phishing vs. legitimate), extracts features using feature\_extractor.py, trains machine learning models, and saves them. We demonstrate using **Random Forest** and **Support Vector Machine (SVM)** classifiers from scikit-learn[[1]](https://medium.com/pythons-gurus/phishing-no-more-creating-a-url-classifier-with-machine-learning-a8a1b3e90a0f#:~:text=%27length_url%27%2C%20%27length_hostname%27%2C%20%27ip%27%2C%20%27nb_dots%27%2C%20%27nb_hyphens%27%2C,domain_in_brand%27%2C%20%27brand_in_subdomain%27%2C%20%27brand_in_path%27%2C%20%27suspecious_tld%27).

Typical steps: 1. **Load Dataset:** The code attempts to load a CSV (e.g. phishing\_data.csv) with columns like url and label (0 or 1). If not found, the script instructs the user to download the UCI Phishing Websites dataset[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no) or a similar dataset. 2. **Extract Features:** We apply extract\_features(url) to each URL to build a feature matrix X. 3. **Train-Test Split:** The data is split (e.g. 80/20) into training and testing sets. 4. **Train Models:** - *Random Forest:* Uses RandomForestClassifier for robust ensemble learning.  
- *SVM:* Uses an SVC (with a pipeline including StandardScaler) for a different perspective[[1]](https://medium.com/pythons-gurus/phishing-no-more-creating-a-url-classifier-with-machine-learning-a8a1b3e90a0f#:~:text=%27length_url%27%2C%20%27length_hostname%27%2C%20%27ip%27%2C%20%27nb_dots%27%2C%20%27nb_hyphens%27%2C,domain_in_brand%27%2C%20%27brand_in_subdomain%27%2C%20%27brand_in_path%27%2C%20%27suspecious_tld%27). 5. **Evaluate:** Print accuracy and classification report on test data. 6. **Save Models:** Use joblib.dump to save trained models (rf\_model.joblib, svm\_model.joblib).

# train\_model.py  
  
import sys  
import pandas as pd  
import numpy as np  
from sklearn.model\_selection import train\_test\_split  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.svm import SVC  
from sklearn.preprocessing import StandardScaler  
from sklearn.pipeline import make\_pipeline  
from sklearn.metrics import classification\_report, accuracy\_score  
import joblib  
from feature\_extractor import extract\_features  
from utils import logger  
  
def load\_data(csv\_file):  
 """  
 Load dataset CSV with columns ['url', 'label'].  
 """  
 try:  
 df = pd.read\_csv(csv\_file)  
 except FileNotFoundError:  
 logger.error(f"Dataset file '{csv\_file}' not found.")  
 sys.exit("Please download the dataset (e.g. from UCI) and save as 'phishing\_data.csv'.")  
 return df  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Load data  
 data\_file = "phishing\_data.csv"  
 df = load\_data(data\_file)  
  
 # Ensure columns 'url' and 'label' exist  
 if 'url' not in df.columns or 'label' not in df.columns:  
 sys.exit("CSV must contain 'url' and 'label' columns.")  
  
 # Extract features for all URLs  
 feature\_list = []  
 for url in df['url']:  
 try:  
 feats = extract\_features(url)  
 feature\_list.append(feats)  
 except Exception as e:  
 logger.warning(f"Skipping URL due to error: {e}")  
 feature\_list.append({}) # placeholder for error (won't happen if data is clean)  
  
 # Convert list of dicts to DataFrame  
 features\_df = pd.DataFrame(feature\_list).fillna(0)  
 X = features\_df.values  
 y = df['label'].values  
  
 # Train/test split  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(  
 X, y, test\_size=0.2, random\_state=42, stratify=y  
 )  
  
 # Train Random Forest  
 rf = RandomForestClassifier(n\_estimators=100, random\_state=42)  
 rf.fit(X\_train, y\_train)  
 rf\_pred = rf.predict(X\_test)  
 logger.info("Random Forest performance:")  
 print(classification\_report(y\_test, rf\_pred))  
 print(f"Accuracy: {accuracy\_score(y\_test, rf\_pred):.4f}")  
  
 # Train SVM (with scaling)  
 svm\_pipeline = make\_pipeline(StandardScaler(), SVC(kernel='rbf', probability=True, random\_state=42))  
 svm\_pipeline.fit(X\_train, y\_train)  
 svm\_pred = svm\_pipeline.predict(X\_test)  
 logger.info("SVM performance:")  
 print(classification\_report(y\_test, svm\_pred))  
 print(f"Accuracy: {accuracy\_score(y\_test, svm\_pred):.4f}")  
  
 # Save models  
 joblib.dump(rf, "rf\_model.joblib")  
 joblib.dump(svm\_pipeline, "svm\_model.joblib")  
 logger.info("Models saved: rf\_model.joblib, svm\_model.joblib")

**Notes:**  
- We log progress with logger.  
- The UCI dataset’s features (e.g. “domain\_registration\_length”) can be incorporated by extending feature\_extractor, but here we focus on URL-based features[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no)[[1]](https://medium.com/pythons-gurus/phishing-no-more-creating-a-url-classifier-with-machine-learning-a8a1b3e90a0f#:~:text=%27length_url%27%2C%20%27length_hostname%27%2C%20%27ip%27%2C%20%27nb_dots%27%2C%20%27nb_hyphens%27%2C,domain_in_brand%27%2C%20%27brand_in_subdomain%27%2C%20%27brand_in_path%27%2C%20%27suspecious_tld%27).  
- Users should provide or download a suitable dataset. For example, the [UCI Phishing Websites dataset](https://archive.ics.uci.edu/ml/datasets/Phishing+Websites)[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no) (in ARFF or CSV form) contains similar features.

## Prediction Module (predict.py)

This script loads a saved model and predicts on new input URLs. It accepts a URL (either via command-line argument or input prompt), extracts features, loads a model file, and outputs whether it is **Phishing** or **Legitimate**.

Usage example:

python predict.py --url "http://example.com/login" --model rf\_model.joblib

# predict.py  
  
import argparse  
import joblib  
from feature\_extractor import extract\_features  
from utils import logger  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 parser = argparse.ArgumentParser(description="Predict phishing (1) or legitimate (0) URL.")  
 parser.add\_argument("--url", type=str, required=True, help="URL to classify")  
 parser.add\_argument("--model", type=str, default="rf\_model.joblib", help="Path to saved model file")  
 args = parser.parse\_args()  
  
 url = args.url.strip()  
 model\_file = args.model  
  
 try:  
 features = extract\_features(url)  
 except Exception as e:  
 logger.error(f"Error extracting features: {e}")  
 exit(1)  
  
 try:  
 model = joblib.load(model\_file)  
 except FileNotFoundError:  
 logger.error(f"Model file '{model\_file}' not found.")  
 exit(1)  
  
 # Prepare feature vector for prediction (ensure correct order)  
 X\_new = [list(features.values())]  
 pred = model.predict(X\_new)[0]  
 label = "Phishing" if pred == 1 else "Legitimate"  
 print(f"Prediction for URL '{url}': {label}")

This module can be integrated into other scripts (like the GUI) to provide on-the-fly predictions.

## Google Safe Browsing API Integration (api\_check.py)

This component checks a URL against Google’s Safe Browsing lists. We use the Lookup API v4: send a POST request with our API key and parse the response. If the URL is flagged (e.g. as SOCIAL\_ENGINEERING or MALWARE), we report it as malicious[[3]](https://developers.google.com/safe-browsing/v4/lookup-api#:~:text=).

You must obtain a Google API key for the Safe Browsing API. In the code below, replace "YOUR\_API\_KEY" or configure it securely (e.g., via an environment variable).

# api\_check.py  
  
import os  
import requests  
from utils import logger  
  
# Endpoint for Safe Browsing lookup (v4)  
API\_ENDPOINT = "https://safebrowsing.googleapis.com/v4/threatMatches:find"  
  
def check\_url\_safebrowsing(url, api\_key=None):  
 """  
 Check a URL against Google Safe Browsing lists.  
 Returns True if URL is found malicious, False otherwise.  
 """  
 if api\_key is None:  
 api\_key = os.getenv("GOOGLE\_API\_KEY")  
 if not api\_key:  
 raise ValueError("Google API key not provided.")  
  
 body = {  
 "client": {"clientId": "phish-detector", "clientVersion": "1.0"},  
 "threatInfo": {  
 "threatTypes": ["MALWARE", "SOCIAL\_ENGINEERING", "UNWANTED\_SOFTWARE", "POTENTIALLY\_HARMFUL\_APPLICATION"],  
 "platformTypes": ["ANY\_PLATFORM"],  
 "threatEntryTypes": ["URL"],  
 "threatEntries": [{"url": url}]  
 }  
 }  
 params = {"key": api\_key}  
 try:  
 response = requests.post(API\_ENDPOINT, params=params, json=body)  
 response.raise\_for\_status()  
 result = response.json()  
 # If "matches" in result, the URL is on a list  
 return "matches" in result  
 except requests.RequestException as e:  
 logger.error(f"Safe Browsing API request failed: {e}")  
 return False  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 test\_url = input("Enter URL to check with Google Safe Browsing: ").strip()  
 api\_key = os.getenv("GOOGLE\_API\_KEY") or "YOUR\_API\_KEY"  
 try:  
 malicious = check\_url\_safebrowsing(test\_url, api\_key)  
 if malicious:  
 print("Safe Browsing result: MALICIOUS URL!")  
 else:  
 print("Safe Browsing result: URL not found in malicious lists (likely safe).")  
 except Exception as e:  
 logger.error(e)  
 print("An error occurred while checking Safe Browsing.")

[[3]](https://developers.google.com/safe-browsing/v4/lookup-api#:~:text=) shows the structure of the POST request used. The script prints whether the URL is identified as malicious by Google. In the GUI (below), we combine this check with the ML model's prediction for a final verdict.

## GUI Application (gui.py)

The gui.py script builds a simple desktop GUI using Tkinter. Users can enter a URL and click “Check”. The application will run the **prediction module** and **Safe Browsing check**, then display a message indicating whether the URL is safe or phishing. Color-coded output (green for safe, red for phishing) improves user feedback.

Key points: - Import the predict function or model to get ML prediction. - Call check\_url\_safebrowsing to get external validation. - Display combined result.

# gui.py  
  
import tkinter as tk  
from tkinter import messagebox  
from feature\_extractor import extract\_features  
from api\_check import check\_url\_safebrowsing  
import joblib  
from utils import is\_valid\_url, logger  
  
# Load the trained model (default: Random Forest)  
try:  
 model = joblib.load("rf\_model.joblib")  
except Exception as e:  
 messagebox.showerror("Error", f"Cannot load model: {e}")  
 exit(1)  
  
def on\_check():  
 url = entry.get().strip()  
 if not is\_valid\_url(url):  
 messagebox.showwarning("Invalid URL", "Please enter a valid URL (include http:// or https://).")  
 return  
  
 # Get ML prediction  
 try:  
 feats = extract\_features(url)  
 pred = model.predict([list(feats.values())])[0]  
 ml\_label = "Phishing" if pred == 1 else "Legitimate"  
 except Exception as e:  
 messagebox.showerror("Error", f"Failed to extract features: {e}")  
 return  
  
 # Check Google Safe Browsing  
 sb\_label = ""  
 try:  
 api\_key = None # or retrieve from config/environment  
 is\_mal = check\_url\_safebrowsing(url, api\_key)  
 sb\_label = " (Malicious!)" if is\_mal else ""  
 except Exception as e:  
 logger.warning(f"Safe Browsing check failed: {e}")  
 sb\_label = " (Safe Browsing API error)"  
  
 # Final decision: if either indicates phishing, warn the user  
 if pred == 1 or (sb\_label != "" and "(Malicious)" in sb\_label):  
 result\_text = "WARNING: Phishing URL!"  
 result\_color = "red"  
 else:  
 result\_text = "Safe URL"  
 result\_color = "green"  
  
 # Display result  
 output.config(text=result\_text + sb\_label, fg=result\_color)  
  
# Build GUI  
root = tk.Tk()  
root.title("Phishing URL Detector")  
  
tk.Label(root, text="Enter URL:").grid(row=0, column=0, padx=5, pady=5)  
entry = tk.Entry(root, width=50)  
entry.grid(row=0, column=1, padx=5, pady=5)  
check\_button = tk.Button(root, text="Check URL", command=on\_check)  
check\_button.grid(row=0, column=2, padx=5, pady=5)  
output = tk.Label(root, text="", font=("Arial", 14))  
output.grid(row=1, column=0, columnspan=3, pady=10)  
  
root.mainloop()

This GUI integrates the ML model and Safe Browsing check to give a comprehensive result. It assumes rf\_model.joblib is in the same directory and that the Safe Browsing function is available.

## Utility Functions (utils.py) and requirements.txt

We include a utility module for common tasks:

# utils.py  
  
import logging  
from urllib.parse import urlparse  
  
# Configure logging  
logging.basicConfig(level=logging.INFO, format='%(asctime)s %(levelname)s: %(message)s')  
logger = logging.getLogger("phish\_detector")  
  
def is\_valid\_url(url):  
 """  
 Simple URL validation: checks for scheme and netloc.  
 """  
 try:  
 parts = urlparse(url)  
 return all([parts.scheme, parts.netloc])  
 except Exception:  
 return False

The requirements.txt should list all external packages needed:

numpy  
pandas  
scikit-learn  
joblib  
requests

(We use standard libraries for tkinter, urllib, etc., so they do not need installation via pip.)

## Sample Dataset

For a dataset, you can use the **UCI Phishing Websites dataset**[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no) or any labeled collection of URLs. If using the UCI data, convert it to CSV with columns url,label (where label is 1=phishing, 0=legitimate). Alternatively, prepare a CSV with sample URLs and labels. Ensure this CSV is named phishing\_data.csv in the working directory or adjust train\_model.py accordingly.

If you include an example, a minimal CSV could be:

url,label  
http://example.com,0  
http://192.0.2.1/login,1  
...

Run train\_model.py to train and generate rf\_model.joblib and svm\_model.joblib. These model files can then be used by predict.py and gui.py.

## Final Notes

* **Modularity:** Each component is in its own script for clarity. All code is commented and exceptions are handled to ensure robustness.
* **Extensibility:** You can add more features (e.g. WHOIS domain age, favicon checks) by extending feature\_extractor.py.
* **Resources:** Feature ideas come from prior research and datasets[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no)[[1]](https://medium.com/pythons-gurus/phishing-no-more-creating-a-url-classifier-with-machine-learning-a8a1b3e90a0f#:~:text=%27length_url%27%2C%20%27length_hostname%27%2C%20%27ip%27%2C%20%27nb_dots%27%2C%20%27nb_hyphens%27%2C,domain_in_brand%27%2C%20%27brand_in_subdomain%27%2C%20%27brand_in_path%27%2C%20%27suspecious_tld%27). Google Safe Browsing API usage is based on Google’s documentation[[3]](https://developers.google.com/safe-browsing/v4/lookup-api#:~:text=).

With all components in place (see file listing above), this system can be packaged into a ZIP archive. After installing the required packages, users can train the models, and then run gui.py for a desktop phishing checker interface.

[[1]](https://medium.com/pythons-gurus/phishing-no-more-creating-a-url-classifier-with-machine-learning-a8a1b3e90a0f" \l ":~:text=%27length_url%27%2C%20%27length_hostname%27%2C%20%27ip%27%2C%20%27nb_dots%27%2C%20%27nb_hyphens%27%2C,domain_in_brand%27%2C%20%27brand_in_subdomain%27%2C%20%27brand_in_path%27%2C%20%27suspecious_tld%27) Phishing No More: Creating a URL Classifier with Machine Learning | by Ahmed Islam | Python’s Gurus | Medium

<https://medium.com/pythons-gurus/phishing-no-more-creating-a-url-classifier-with-machine-learning-a8a1b3e90a0f>

[[2]](https://archive.ics.uci.edu/dataset/327/phishing+websites#:~:text=Variable%20Name%20Role%20Type%20Description,no%20having_sub_domain%20Feature%20Integer%20no) UCI Machine Learning Repository

<https://archive.ics.uci.edu/dataset/327/phishing+websites>

[[3]](https://developers.google.com/safe-browsing/v4/lookup-api#:~:text=) Safe Browsing Lookup API (v4)  |  Safe Browsing APIs (v4)  |  Google for Developers

<https://developers.google.com/safe-browsing/v4/lookup-api>