URL Phishing Detection in Real Time Using Machine Learning and Browser Extension

AAS - Final Report

José Gameiro, 108840
Tomás Victal, 109018

Aprendizagem Aplicada a segurança
DETI
University of Aveiro
January, 2026

Abstract

Phishing attacks are one of the most significant risks in cybersecurity, as they use deceptive URLs to steal user information. This project proposes a real-time solution using a Firefox browser extension that checks whether a URL is safe before the page loads, based on an AI model implemented in Python.The model was trained using a dataset of 729,737 URLs, consisting of 585,511 legitimate URLs and 144,226 phishing URLs. A total of 67 features were extracted from each URL, covering structural, lexical, and semantic characteristics. Using this data, two supervised learning models were trained: Random Forest and K-Nearest Neighbors (KNN).Based on the obtained results, Random Forest was selected as the most suitable model for this security use case, achieving an accuracy of 97.87% and a recall of 94.15%. Even though the KNN model produced fewer false positives on legitimate websites, it generated more false negatives, making it less appropriate for phishing detection.

Table of Contents

1 Introduction

Phishing attacks are among the most persistent threats in cybersecurity today. With the widespread use of the internet by people of all ages, ordinary users have become increasingly susceptible to these attacks.

Phishing typically works by tricking users into believing that a URL they are using is safe and legitimate. In most cases, these URLs imitate well-known domains or websites in order to steal account credentials from users on the platforms being impersonated. To deceive users, these URLs often use unusual characters that closely resemble legitimate letters, for example, replacing "o" with "0" or using similar-looking Unicode characters, and overly long URLs with multiple subdomains.

There is already a good level of awareness about this type of trick. However, with the increasing value for people carrying out phishing attacks and the growing number of less-informed users exploring the internet and phishing techniques evolving rapidly, simply alerting users is no longer sufficient. For this reason, multiple tools have emerged to address this problem. These tools include public databases of known phishing URLs and websites that check whether a given URL is associated with phishing activity.[1]

The problem with these tools is that they require the user to go visit the website and manually paste the URL they want to check, the analysis process can take some time, as the tool needs to examine various aspects of the website before providing a response.

To solve this problem in this project we will build a browser extension that analyses the URL using machine learning, this extension works by blocking the URL and then asks a service if the URL is safe depending on the response the extension lets you access the page or blocks it, if you are certain of what you are doing you can also add domains to a whitelist.

In this document, we will go through the multiple phases of this process, starting with the related work, where existing solutions are explored. We then present the methodology and implementation, discussing the data used, the methods applied, and their implementation. Finally, we analyze the results and discuss possible future work.

All the code developed for this solution is available on GitHub: `https://github.com/zegameiro/AAS-FinalProj-PUD`

2 Related Work

During our research for this project, we identified two relevant works that apply machine learning techniques to phishing URL detection.

2.1 lexical based machine learning in a real-time url phishing detection

In this paper the author develop a real-time machine learning-based phishing detection using the URL. Instead of analyzing full web page content making the process of evaluation slow and requiring more data, this method uses features that can be extracted directly from the URL string. For the model they tested four different models using: Random Forest, K-Nearest Neighbors (KNN), Support Vector Machine (SVM), and Logistic Regression.

The paper concluded that the Random Forest classifier outperform the others models, achieving an accuracy of 99.57% and a low false-positive rate of 0.53%. This method also proved to be fast enough for real time use, with response time of 52ms for the Random Forest and 12ms for the SVM.[2]

2.2 Analysis of Traditional Machine,Learning, Deep Learning and Boosting Algorithms on Phishing URL Detection

In this study, the authors made a comparative analysis of Traditional Machine Learning, Deep Learning and Boosting Algorithms for phishing URL detection based on URL features. Multiple models were compared including Random Forest, CNN, LSTM, and XGBoost, to determine which handled the given data most effectively. The results showed that Boosting algorithms, specifically LightGBM, XGBoost, and CatBoost, outperformed the other models achieving accuracies between 92% and 98%[3].

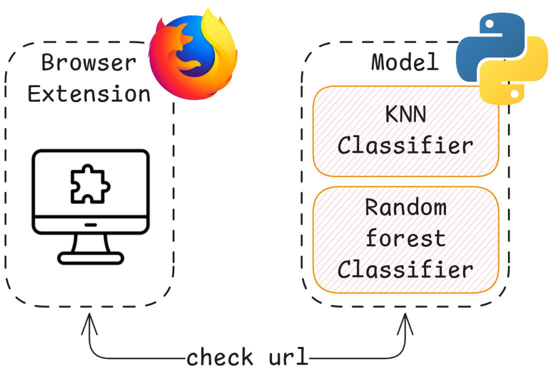


Figure 1: Architecture of the solution

3 Methodology

The methodology of the proposed solution is based on a browser extension that sends requests to a Python API, which uses two different classifiers: a K-Nearest Neighbors (KNN) classifier and a Random Forest classifier.1

3.1 Browser Extension

The browser extension will be developed for Firefox and will analyze every URL entered in the address bar before the page is loaded. To perform this analysis, the extension sends a request to the Python API. If the URL is considered unsafe, the user is redirected to a local warning page; if it is considered safe, the requested page is loaded normally. To avoid locking the user into the service's decision, the user can explicitly mark a domain as safe by adding it to a whitelist.

3.2 Models and Data

3.2.1 Data Acquisition

The data used in this work was obtained from multiple sources. One source was the Detecting-Malicious-URL-Machine-Learning repository on GitHub by rlilojr[4]. Phishing links were collected from the PhishTank database[5], beacause of rate limits for not having an account, we used this github[1] to acess the most recent version of the Phistank data. For benign URLs, we used domains from the Majestic Million dataset provided by Majestic[6] and developed a Python web crawler to scrape URLs from those domains.

To complement the phishing URL dataset, we also used one available on Kaggle created by Arvind Prasad[7], which was already preprocessed. This dataset contains **34,850 legitimate** and **100,945 phishing URLs**, making a total of **235,795 URLs** besides this it also contained some features related to each URL. However we decided to use only the Phishing URLs, since the developed web crawler obtained **959,239 legitimate URLs**, we also decided not to use the URL features present in this dataset since we developed our own feature extractor.

[1]https://github.com/ProKn1fe/phishtank-database

3.2.2 Pre-processing

The URLs obtained from the data sources described above were all labeled as either benign or phishing. URLs collected from the Majestic domains through the crawler were labeled as benign, while those obtained from PhishTank were labeled as phishing. The Kaggle dataset was already labeled, and no additional preprocessing was required.

The script filter_non_spam_by_majestic.py filters the URLs obtained by the crawler, removing any URL whose domain is not present in the Majestic Million[6]. It also filters out lines that do not contain a URL. During data analysis, we found multiple URLs duplicated within the benign set, and some URLs were present in both the benign and phishing sets. For the model, URLs present in both sets were considered phishing, as we chose to follow PhishTank's[5] classification rather than relying on the most popular URLs.

So in the end we obtained a total of **585,511 legitimate URLs**, and **144,226 phishing URLs**.

We divided the unique URLs into two separate groups. 85% of the data was allocated to the Training Dataset, used exclusively to train the models, while the remaining 15% was set aside as the Test Set. The datasets were split in a stratified manner, maintaining the same phishing-to-benign ratio in both the training and evaluation sets. We also added a verification step to ensure zero overlap between the groups. The models never encountered the Test Set data during training. This strict separation ensures that the evaluation results reflect the models' ability to generalize to new, unseen URLs.

For the KNN model, we used a standard feature scaling.

3.2.3 Feature Extraction

To train the models, we extracted a total of 50 features from each URL, plus 17, if the URL contained any of the special characters present in our `constants.py` file, in the **URL_PROPERTIES** file, making a total of 67 features. These capture multiple types of information, including structural, lexical, and semantic characteristics of the URL. The extracted features, detailed in Table 1, can be categorized into three groups:

1. **Lexical and Statistical Features:** Properties of the raw URL string. Phishing URLs often have high entropy (randomness), excessive length, or unusual character ratios (e.g., high digit density or mixed character sets).

2. **Domain and Structure Features:** These features focus on the composition of the protocol, host, path, and **query strings**. It checks for IP addresses in the hostname, path depth, protocol usage (HTTP/HTTPS), and the presence of suspicious Top-Level Domains (TLDs) or URL shorteners.

3. **Obfuscation and Brand Impersonation:** Phishing attacks often use techniques to visually trick the user or bypass filters. We extract features related to homoglyph attacks (non-ASCII characters confusingly similar to Latin letters), Punycode encoding, open redirects, and **social engineering keywords** (e.g., 'login'). Additionally, we measure brand impersonation using Levenshtein distance.

Table 1 provides a complete description of the feature set utilized in our classification model.

Feature Name	**Description**
`url_length`	Length of URL
`has_https` / `has_http`	Binary for the presence of HTTPS or HTTP protocol.
`has_ip`	Binary check if the domain is an IP address.
`has_port`	Binary check if a non-standard port is specified.
`has_fragment`	Binary check for the presence of a URL fragment (anchor).
`redirect_count`	Count of protocol indicators, used to detect open redirects.
`subdomain_count`	Number of subdomains (dots in domain minus one).
`domain_token_count`	Number of tokens in the domain delimited by dots.
`avg_domain_token_length`	Average length of domain tokens.
`max_domain_token_length`	Length of the longest token in the domain.
`tld_length`	Length of the Top-Level Domain (TLD).
`has_suspicious_tld`	Binary check against a list of typically malicious TLDs.
`has_trusted_tld`	Binary check against a list of reputable TLDs.
`is_url_shortener`	Binary indicator if the domain matches known shorteners.
`path_length`	Total length of the URL path string.
`path_depth`	Depth of the path (count of '/' characters).
`path_token_count`	Number of tokens in the path.
`double_slash_in_path`	Indicator for '//' inside the path (often used for obfuscation).
`query_length`	Total length of the query string.
`query_param_count`	Number of parameters in the query string.
`digit_count/ratio`	Count and ratio of numeric characters.
`letter_count/ratio`	Count and ratio of alphabetic characters.
`uppercase_count`	Count of uppercase characters.
`special_char_ratio`	Ratio of non-alphanumeric characters.
`char_diversity`	Ratio of unique characters to total length.
`entropy`	Shannon entropy score (randomness of characters).
`vowel_consonant_ratio`	Ratio of vowels to consonants (detects random strings).
`suspicious_word_count`	Count of security-sensitive keywords (e.g., 'login', 'secure').
`homoglyph_count`	Count of characters used in IDN homograph attacks.
`has_punycode`	Presence of 'xn–' indicating Punycode encoding.
`encoded_char_count`	Count of URL-encoded characters (%).
`consecutive_digits`	number of consecutive digits.

Continued on next page

Table 1 – continued from previous page

Feature Name	Description
`consecutive_consonants`	number of consecutive consonants.
`brand_in_subdomain`	is a known brand name in the subdomain.
`brand_in_path`	is a known brand name in the path.
`levenshtein_to_brand`	Minimum edit distance to closest known brand (typosquatting).
`path_to_url_ratio`	Ratio of path length to total URL length.
`query_to_url_ratio`	Ratio of query string length to total URL length.
`domain_to_url_ratio`	Ratio of domain length to total URL length.
`unicode_ratio`	Ratio of non-ASCII characters to total URL length.
`encoding_ratio`	Ratio of URL-encoded characters (%) to total URL length.
`longest_word_length`	Length of the longest alphabetic word found in the URL.
`avg_word_length`	Average length of all alphabetic words found in the URL.
`avg_path_token_length`	Average length of tokens in the path (delimited by '/').
`has_mixed_charset`	Binary check for mixing scripts (Latin, Cyrillic, Greek).
`has_homoglyphs`	Binary indicator if suspicious homoglyphs are present.
`non_ascii_count`	Count of characters with ASCII value ¿ 127.

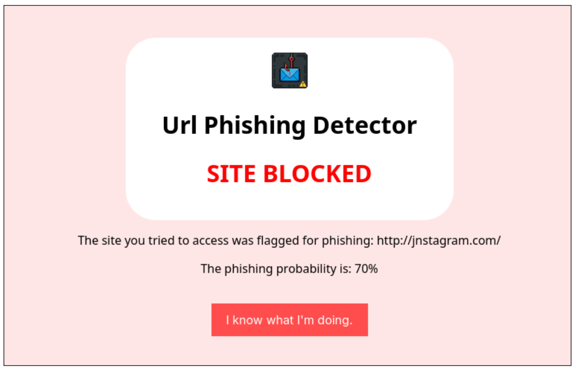
Table 1: Features extracted

3.2.4 Model Selection

Based on the work reviewed in Section 2, we decided to experiment with two different supervised models, K-Nearest Neighbors (KNN) and Random Forest, since our dataset contains labeled data.

KNN is used to calculate the distance between a new data point and the existing dataset. In this work, it is applied by calculating the distance between a new URL and all saved URLs, and then predicting whether the URL is benign or phishing based on the nearest neighbors.

Random Forest is another very used model for this use case. It works by constructing multiple decision trees, where each tree outputs a classification based on their decision. The final output is determined through a voting system that combines the predictions of all trees, using defined weights.



The site you tried to access was flagged for phishing: http://jnstagram.com/

The phishing probability is: 70%

Figure 2: Blocked Page

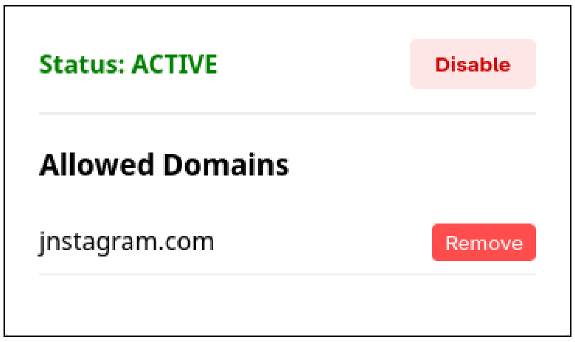


Figure 3: Extension Popup

4 Implementation

4.1 Libraries used

For this project, we used uv to manage the project and its dependencies. The machine learning models were trained using the scikit-learn package. To collect benign data, we implemented a crawler using the Python library crawlee. Since this library requires operating system–level dependencies, we used Docker to run the crawler in a controlled environment. We also used the requests library to retrieve the databases described in Section 3.2.1.

For data preprocessing, we used NumPy and Polars. To organize and validate data structures with enriched typing, we used Pydantic. Finally, to provide a service interface for the Firefox extension, we implemented a REST API using Flask.

4.2 Code Structure

The code is organized into reusable modules so that it can be shared between the training process and the prediction logic used by the Flask application. The main modules are:

- DatasetLoader: Loads local datasets and, if they are not available, downloads the data from the online databases.
- URLFeatureExtractor: Responsible for extracting features from URLs.

- ai_models: Contains a parent class and two derived classes for the KNN and Random Forest models. This design makes it easy to add new models and avoids code duplication across the models classes.

In addition, there are three separate main scripts: one for training the models (main.py), one for running the crawler (crawl.py), and one for the Flask application (app.py). Each script is executed independently to perform its specific task. As mentioned earlier, the crawler requires a Docker container to run; for this purpose, a shell script is provided to build the required image and execute it. The other two main scripts are executed using uv.

The main.py script starts by loading the urls and their labels from the dataset loader component. Then the loaded dataset is split into training and test sets. Next, the user selects which model to train: Random Forest, KNN, or both. For each selected model, training is performed only on the training set, with an internal validation split used during training. During the training the URLFeatureExtractor is used to extract the features of the URLs. After training, the model is saved to disk. Once trained the model is tested with the test set. Finally the results of the training are showed.

The Flask application loads the trained model and then waits for incoming requests containing a URL. It uses the loaded model to infer whether the URL is phishing or benign and returns the result.

For the browser component, we implemented an extension that verifies every URL entered in the address bar. The extension blocks the browser from loading the URL until it receives a response from the Flask application. Depending on the response, the extension may block the URL and display a warning page, as shown in Figure 2. If the user knows that the website they are accessing is safe, they can add the domain to an allowed domain list, which is available in the extension's popup interface, as shown in Figure 3. The extension also allows the user to temporarily disable protection and to edit the allowed domain list by removing entries.

4.3 Challenges

4.3.1 Crawler

Due to compatibility issues between the dependencies of the Crawlee library and the operating system, the installation and use of the tool were difficult. To work around this and simplify the setup, we decided to use a Docker image from Playwright[2].

4.3.2 Getting the Data

During the process,as was mentioned in 3.2.2, we encountered some data conflicts. While analyzing the data, we found multiple URLs duplicated within the benign URLs, and some URLs were present in both the benign and phishing sets. We removed the duplicates, and the URLs present in both sets were considered phishing, as they were well-known URLs from the Majestic Million[6] dataset but were classified as phishing by PhishTank[5].

[2]https://playwright.dev/python/docs/docker

5 Evaluation

To properly evaluate the results of the two approaches used, Random Forest and K-Nearest Neighbors, we choose the following metrics to understand exactly how the models were failing or succeeding:

- **Accuracy**: The overall percentage of correct predictions, for both Phishing and Legitimate URLs
- **Precision**: Measures the quality of a positive prediction or it tries to answer the following question *When the model says ”This is Phishing”, how often is it right?”*
- **Recall**: Measures the ability to find all positive instances, meaning, out of all actual phishing sites how many did the model catch
- **F1-Score**: The harmonic mean of Precision and Recall. This provides a single score that balances the trade-off between catching phishing attacks and avoiding false alarms.
- **ROC-AUC**: The area under the Receiver Operating Characteristic Curve. This metric evaluates how well the model can distinguish between classes across different probability thresholds.

Besides these metrics we also evaluated the structured of our dataset by identifying the distribution of features across the dataset used for the training process.

The data used to evaluate the metrics mentioned above was the one explained in 3.2.1, where each model trained with an equal training dataset, and then was evaluated with also the same evaluation dataset. The results are described in the following section

6 Results

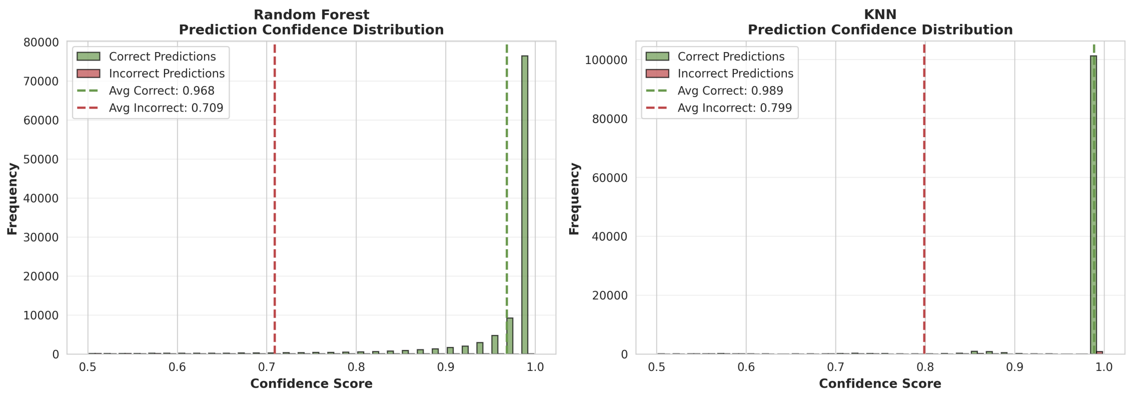


Figure 4: Prediction Confidence distribution for both models

The first visualization we present in Figure 4 displays the distribution of the confidence that the models have when they make a prediction, this means how certain they are of their prediction. A good model should be very sure when it is making a mistake

The two graphs show that the KNN model is often too sure of itself. In the right-hand chart, almost all the predictions are stacked at 100% confidence. While this leads to a high average score when the model is correct, 98.9%, it creates a problem when the model is incorrect. When this model makes a mistake, its average confidence is still quite high at roughly 80%. This means the model rarely signals that it is confused, it just makes a wrong guess with high certainty.

The Random Forest model behaves differently, as shown in the left-hand chart, its predictions are more spread out. When it is right, it is still confident, about 97%, but when it's wrong its confidence drops to about 71%, meaning that this model is better at lowering its confidence score when it encounters a difficult or confusing URL.

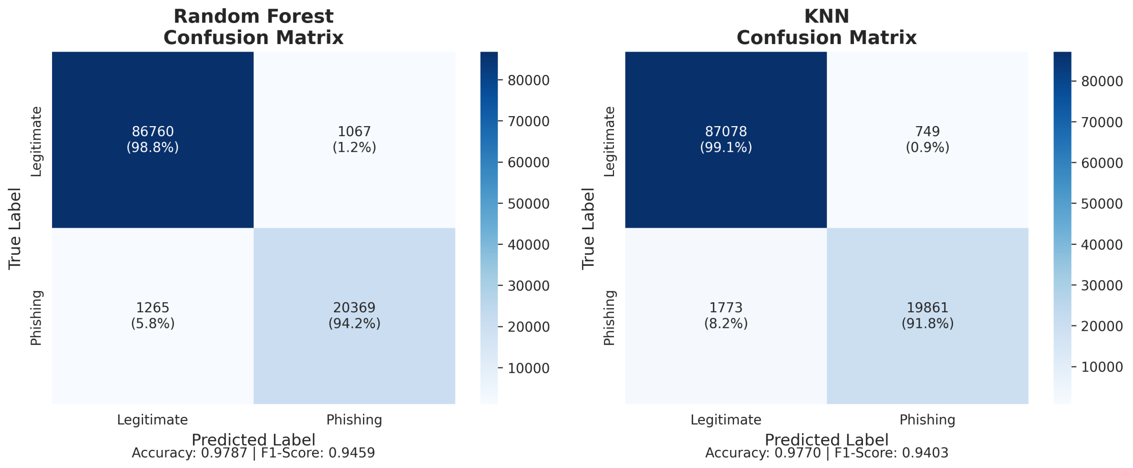


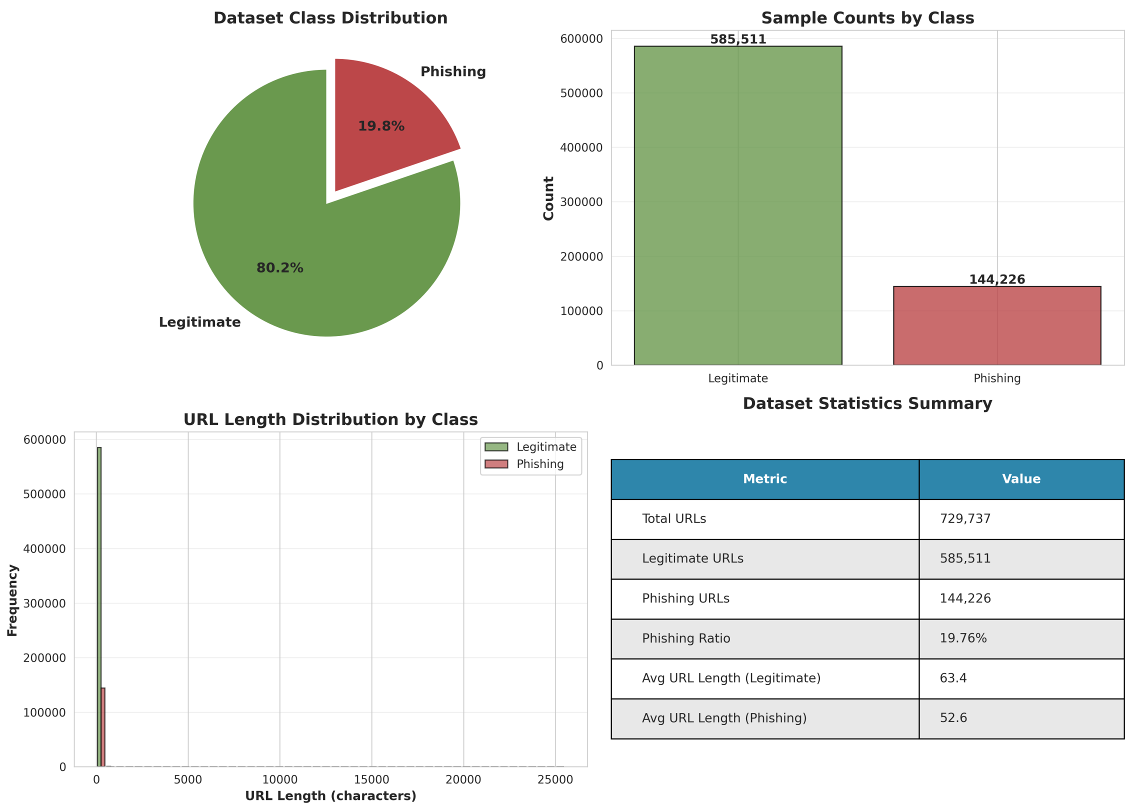
Figure 5: Confusion Matrices for both models

Next we used confusion matrices to look deeper into exactly how the models made mistakes, has shown in Figure 5.

The Random Forest model proved to be the stronger choice for detecting phishing URLs. Looking at the bottom of both matrices, which represents the actual phishing sites, Random Forest correctly identified 94.2% of them, however it missed **1,265** URLs. In comparison, the KNN model missed **1,773** URLs. This means that Random Forest is significantly better at stopping threats, catching nearly 500 more phishing URLs than KNN

However, the KNN model performed slightly better at recognizing safe websites. Looking at the top row of bot matrices, KNN identified correctly 99,1% of legitimate sites, whereas Random Forest was correct 98.8% of the time. This means KNN raised fewer false alarms and it only wrongly blocked 749 safe sites, while Random Forest wrongly block 1,067.

Despite having more false alarms, we considered that Random Forest is a superior approach, when considering a security context, because missing a phishing attack is much more dangerous than accidentally blocking a safe page. However in terms of user experience it may perform worse than KNN, since it may block websites that a user knows its safe.



Metric	Value
Total URLs	729,737
Legitimate URLs	585,511
Phishing URLs	144,226
Phishing Ratio	19.76%
Avg URL Length (Legitimate)	63.4
Avg URL Length (Phishing)	52.6

Figure 6: Dataset Statistics

We decided to also analyze the full dataset to understand the raw material our models were learning, the results can be seen in Figure 6

The most critical finding is the heavy imbalance between safe and dangerous sites. As shown in the pie chart, **80.2%** of the dataset is Legitimate, with 585,511 URLs, while only 19.8% is Phishing, with 144.226 URLs. The total dataset size is **729,737 URLs**, we considered to be an healthy amount of data for training machine learning.

We also looked at whether the length of a link predicts if it is either dangerous or not. The results were surprising, where the **Data Statistics Summary** Table shows that Legitimate URLs are actually longer on average than phishing URLs. This contradicts the common belief that phishing links are always long and more complex, it suggests that attackers often keep their links short, while legitimate sites often have deep folder structures or long article titles.

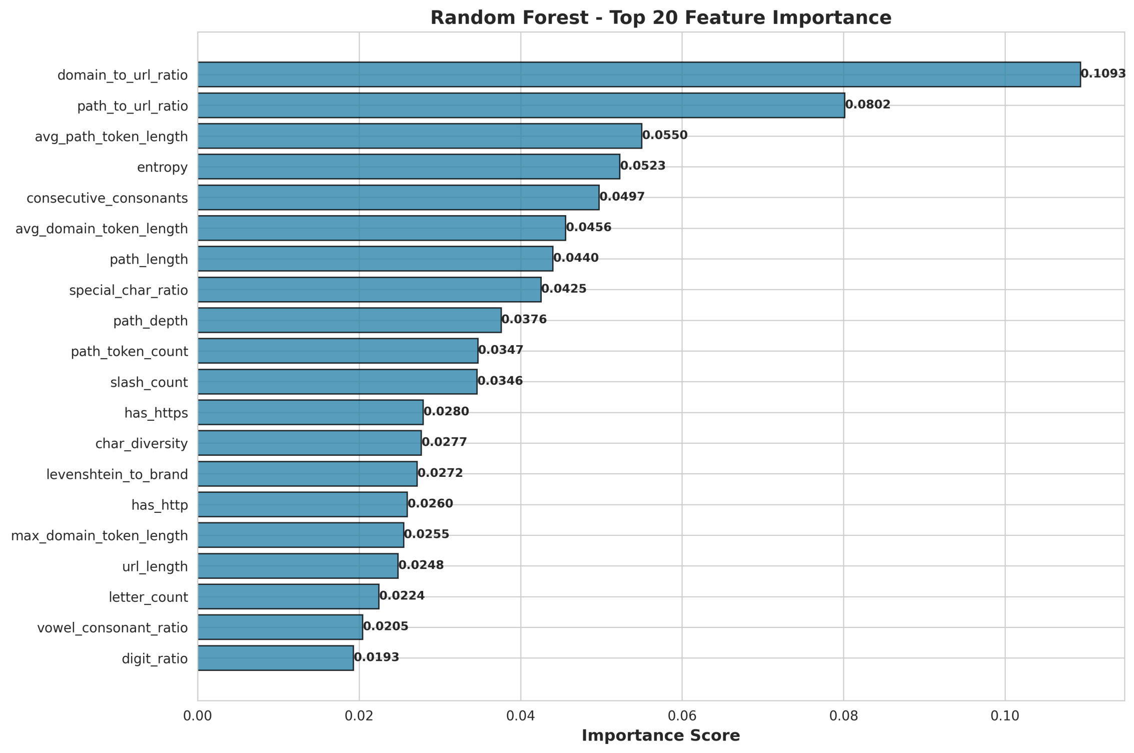


Figure 7: Feature Importance for Random Forest

We only manage to present the top 20 features most used to determine if a URL is considered to be Phishing or Legitimate for the Random Forest due, has it is available in Figure 7. This graph ranks the specific characteristics of a URL based on how useful they were in distinguishing between safe sties and phishing ones. The results show that the model looks at the structure of a link much more than simple key words.

The most important feature remains the Domain-to-URL Ratio (0.1093), followed by the Path-to-URL Ratio (0.0802). This means the model focuses heavily on the balance between the website name and the rest of the link. Phishing attacks often try to hide the real destination by using very long, complicated paths after the domain name. Legitimate sites usually have a cleaner, more balanced structure. The model has learned that if the ”path” part of the URL is disproportionately large compared to the domain, it is likely a trap.

The third most important feature is **Average Path Token Length** (0.0550). This suggests the model is looking at the size of the ”words” found in the URL path. Legitimate sites often use clear, readable words in their links (like /products/ or /category/), whereas phishing links often use random strings or very long, unbroken codes. **Entropy** (randomness) follows closely as the fourth feature (0.0523), confirming that the model is actively hunting for the gibberish or random characters typical of automated phishing generators.

Interestingly, features like **has_https** or **has_http** are much lower on the list (ranking 14th and 17th). This is a significant finding, since it suggests that simply checking for a ”secure” lock icon (HTTPS) is no longer a reliable way to spot a phishing site, because many attackers may use security certificates too. The model has correctly learned to instead of giving more importance to these features, to focus on the deeper shape and complexity of the URL to make its decision.

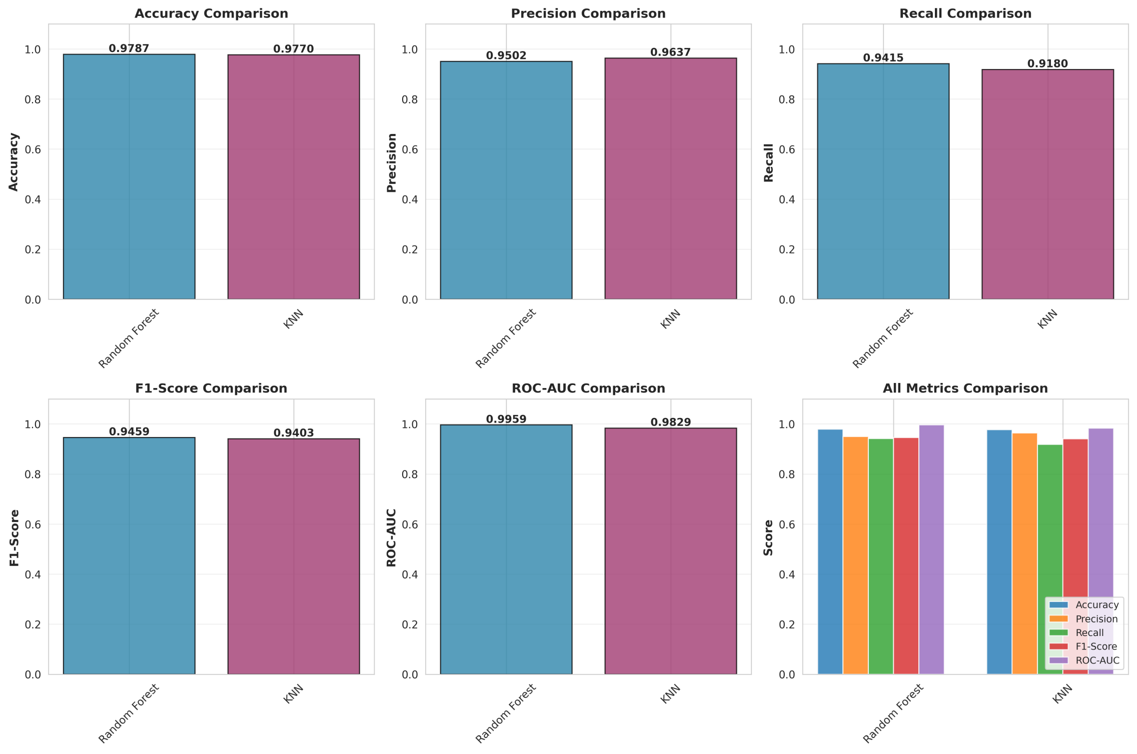


Figure 8: Random Forest and KNN models Comparison

We compared the Random Forest and KNN models side-by-side across the five key performance metrics mentioned in 5. The results can be seen in Figure 8.

The most important difference lies in Recall versus Precision. As shown in the **Recall Comparison** chart, the Random Forest model achieved a score of 94.15%, which is significantly higher than KNN's 91.80%. In simple terms, this means Random Forest is much better at finding phishing attacks, however it misses fewer dangerous links than KNN.

On the other hand, KNN achieved a higher Precision score (96.37% vs 95.02%). This means that when KNN flags a site as dangerous, it is almost always right. It is a more conservative model that rarely raises false alarms. However, this caution comes at a cost, it lets more real attacks slip through undetected.

Missing a phishing attack (a false negative) can lead to a security breach, whereas flagging a safe site (a false positive) is just a minor inconvenience for the user. Because Random Forest catches significantly more attacks, and also achieves higher scores in F1-Score (0.9459) and ROC-AUC (0.9959), it is the superior choice for our final system. Supporting even further the conclusions obtained in the analysis of the Confusion Matrices, Figure 5.

Finally, Figure 6 present the Precision Recall and ROC curves for both models.

The Precision-Recall Curve is critical for our specific problem, since our objective is to try and identify every single phishing site, we look at how the model behaves as "Recall" approaches 1.0. You can see that the KNN line begins to drop downwards sooner than the Random Forest line. This means that as we push the system to find those last few tricky phishing sites, KNN starts making significantly more mistakes

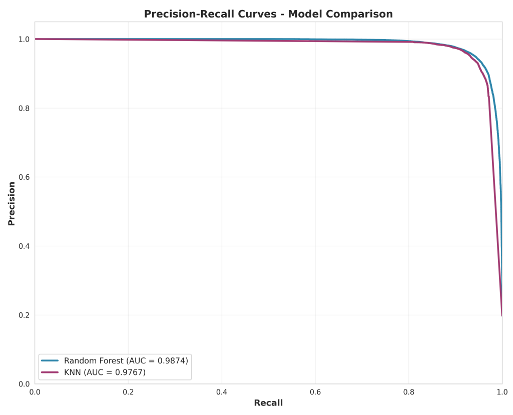


Figure 9: Precision recall curves for both models

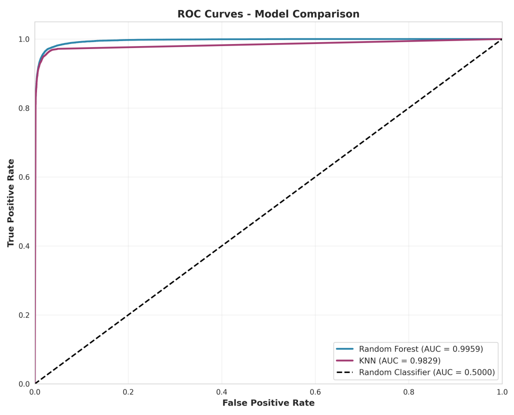


Figure 10: ROC Curves obtained for both models

(false alarms). Random Forest maintains its high precision for longer, proving it is the more robust choice for a high-security environment.

The ROC Curve measures the trade-off between catching attacks and raising false alarms. The ideal curve should hug the top-left corner as closely as possible, which represents catching 100% of attacks with 0% false alarms. The Random Forest model (blue line) is clearly superior here. Its curve is tighter to the corner, achieving an almost perfect Area Under the Curve (AUC) score of 0.9959. The KNN model (purple line) is slightly lower with an AUC of 0.9829. This gap indicates that Random Forest is generally better at distinguishing between safe and dangerous URLs without getting confused.

7 Conclusion and Future Work

This project allowed us to develop a solution to help minimise phishing attacks by creating a browser extension that detects phishing based on the URL. The extension uses an API developed by us, which relies on a machine learning model to make decisions.

To build the model, we extracted 67 features from each URL and used two different classifiers: Random Forest and K-Nearest Neighbors (KNN). The results showed that the Random Forest model performed better overall, achieving an accuracy of 97.87% and successfully identifying 94.2% of phishing attacks.

We selected Random Forest as the final solution because it blocked more phishing URLs than KNN. Although KNN produced fewer false alarms on legitimate websites, Random Forest was a better choice for security, since missing a phishing attack is worse than incorrectly flagging a legitimate page.

For future work, the models would likely benefit from more data and a dataset that is more representative of real-world usage, as the URLs used were those present in robots.txt files, which may differ significantly from the URLs that users typically visit. The model particularly needs improvement in reducing false positives and false negatives. As shown in the confusion matrices (Figure 5) both models contains multiples false positives that can be bad for the image of the websites and false negatives that can lead users to use websites containing phishing, so improving for a reduce classification errors is priority. Other

For the KNN model, we implemented parameter tuning but did not have time to evaluate it. Testing this tuning should therefore be the first step when improving the models.

Bibliography

[1] EasyDMARC Phishing Link Checker. `https://easydmarc.com/tools/phishing-url`, Accessed: 2025-12-31.

[2] Gupta, B. B.; Yadav, K.; Razzak, I.; Psannis, K.; Castiglione, A.; Chang, X. A novel approach for phishing URLs detection using lexical based machine learning in a real-time environment. *Computer Communications* **2021**, *175*, 47–57.

[3] Ashok, A.; Rathis, D.; Raghavendra, R.; Umadevi, V. A Comparative Analysis of Traditional Machine Learning, Deep Learning and Boosting Algorithms on Phishing URL Detection. 2024 IEEE International Conference on Computer Vision and Machine Intelligence (CVMI). 2024; pp 1–6.

[4] rlilojr Detecting-Malicious-URL-Machine-Learning. `https://github.com/rlilojr/Detecting-Malicious-URL-Machine-Learning`, Accessed: 2026-01-02.

[5] PhishTank PhishTank. `https://www.phishtank.com/`, Accessed: 2026-01-02.

[6] Majestic Majestic Million. `https://majestic.com/reports/majestic-million`, Accessed: 2026-01-02.

[7] Prasad, A. PhiUSIIL Phishing URL Dataset. `https://www.kaggle.com/datasets/ndarvind/phiusiil-phishing-url-dataset/data`, Accessed: 2026-01-02.