Developer Manual

Aegis Shield

Phishing Detector

CSG3101.2| Applied Project
Lead Coordinator| Ms. Ann APPUHAMY
Project Supervisor| Mr. Jude Myuran KIRUPARETNAM

Group Members

NAME	SID
Sudam PULLAPERUMA	- 10660248
Dulaj KANKANAMGE	- 10659489
Tharuka GUNASEKARA	- 10659483
Tishantha ELVITIGALA	- 10663914

Contents

Table of Figures

Table of Tables

Overview

Aegis Shield is an extension for the Chrome browser. It is aimed at identifying phishing threats and preventing them at real-time. By utilizing APIs and machine learning models, it performs URL and email content analysis as well as providing easy to use features like whitelist or blacklist management and notifications.

1.0 Features

1.1 URL Security Analysis

- Real-time URL scanning using VirusTotal API integration
- Machine learning-based URL classification
- Intelligent caching system for optimized performance
- Custom whitelist and blacklist management
- Visual security indicators and notifications

1.2 Email Content Analysis

- Machine learning-powered email content evaluation
- Real-time phishing probability assessment
- Support for multiple email formats and content types

1.3 Advanced Machine Learning Implementation

1.3.1 URL Classification Model:-

- Feature extraction from URL components
- Multiple classifier comparison (Random Forest, XGBoost, Decision Tree, Logistic Regression)
- Comprehensive feature importance analysis
- Model persistence for continuous improvement

1.3.2 Email Classification Model:-

- Text preprocessing and cleaning
- TF-IDF vectorization
- Multiple classifier implementation including Naive Bayes, Logistic Regression, SGD Classifier, XGBoost, Decision Tree, Random Forest, MLP Classifier
- Model evaluation with detailed performance metrics

1.4 Security Management:-

- Custom whitelist/blacklist configuration
- Real-time threat notifications
- Comprehensive security reporting
- Export functionality for security logs

2.0 Datasets

2.1 Phishing Email Dataset

The email classification model was trained on the "Phishing Email Dataset" from Kaggle, authored by Subha Jyoti Das. It contains the largest collection of legitimate plus phishing emails: 43,836 emails. The balanced and diversity of the data in the dataset makes it very useful to train robust phishing detection models.

2.1.1 Key characteristics of the email dataset:

- **Total samples**: 43,836 emails
- **Features**: Email content and type classification
- **Source**: Real-world email communications
- **Format**: CSV file with text preprocessing capabilities
- **Publication**: Available on Kaggle under open-source terms
- **Citation**: Phishing email Detection. (2023, July 6). Kaggle. https://www.kaggle.com/datasets/subhajournal/phishingemails

2.1.2 Dataset Structure

The Phishing_Email.csv dataset contains **18,650 rows and 3 columns**.

Columns:

- **Unnamed**: 0: Index column (can be ignored or removed during preprocessing).
- **Email Text**: The content of the email (subject, body).
- **Email Type**: The classification label, indicating whether the email is a "Safe Email" or "Phishing Email".

First Few Rows of the Dataset

	Unnamed: 0	Email Text
0	0	re : 6 . 1100 , disc : uniformitarianism , re ...
1	1	the other side of * galicismos * * galicismo *...
2	2	re : equistar deal tickets are you still avail...
3	3	\nHello I am your hot lil horny toy.\n I am...
4	4	software at incredibly low prices (86 % lower...

Figure 1 - First Few Rows of the Dataset (Phishing Email Dataset)

Distribution of Emails per Type

```
Distribution of Emails per Type:
Email Type
Safe Email        11322
Phishing Email     7328
Name: count, dtype: int64
```

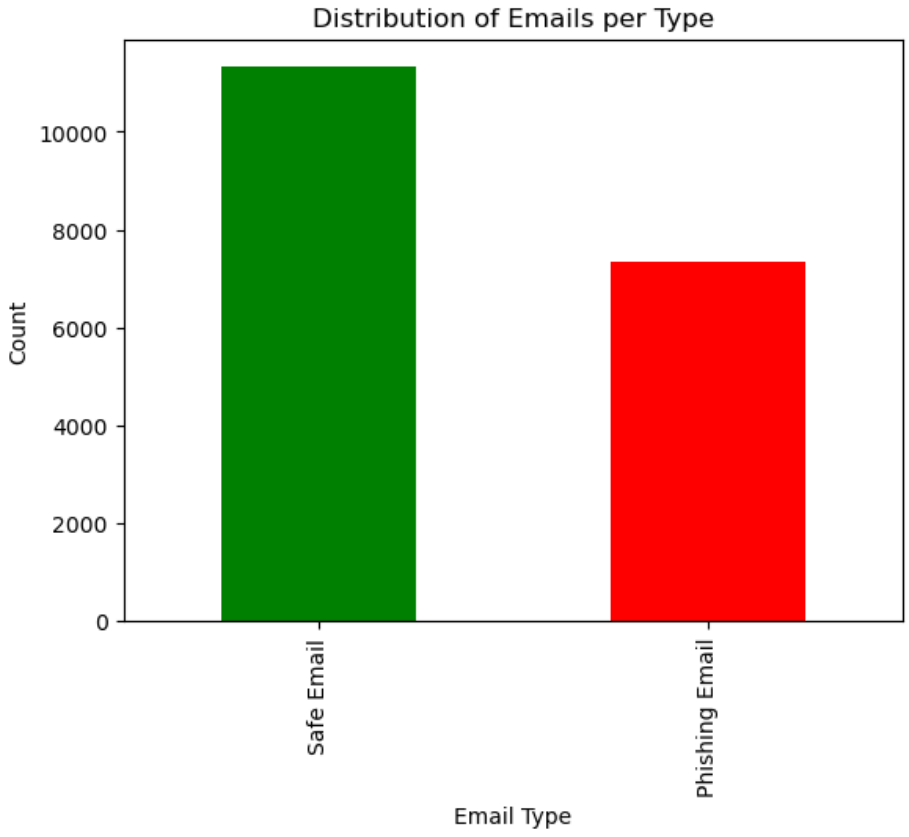

Figure 2 - Distribution of Emails per Type

2.1.3 Steps to Use the Dataset

1. Place the dataset in the data/ directory of your project repository.
2. Reference the file path correctly in your code or notebook.

```
# Load the dataset
import pandas as pd
data = pd.read_csv('dataset_path')
```

Figure 3 - Load the Phishing Email Dataset

2.1.4 Exploration and Preprocessing

1. Exploratory Analysis:
   - Inspect the distribution of email types using bar charts.
   - Visualize the text data using a **WordCloud.**
2. Preprocessing Tasks:
   - Convert text to lowercase.
   - Remove URLs, special characters and numbers.
   - Tokenize and vectorize the text using **TfidfVectorizer. :-**
     - Stop Words: Common words (e.g., “the”, “is”) are removed using English stop words.
     - Max Features: The vectorizer uses a maximum of 10,000 features.
     - Purpose: Converts email text into a numerical representation based on the importance of words in the document and across the dataset.

```
('tfidf', TfidfVectorizer(stop_words='english', max_features=10000)),
('classifier', MultinomialNB())
```

Figure 4 - TfidfVectorizer

```
# Preprocessing function
def preprocess_text(text):
    text = text.lower()
    text = re.sub(r'https?://\S+|www\.\S+', '', text)  # Remove URLs
    text = re.sub(r'[^a-z\s]', '', text)  # Remove special characters and numbers
    return text
```

Figure 5 - Pre-processing Phishing Email Dataset

2.1.5 Data Preprocessing and Feature Engineering

Data Preprocessing

1. Convert Text to Lowercase:- Ensures uniformity in the text.
2. Remove URLs:- Strips out links that do not contribute to the textual semantics.
3. Remove Special Characters and Numbers:- Retains only alphabetical characters.
4. Handle Missing Values:- Removes rows with missing data.

```
# Check for missing values
print("\nChecking for Missing Values:")
print(data.isnull().sum())
```

Figure 6 - Check for Missing Values in Phishing Email Dataset

Feature Engineering

1. Text Vectorization:- Converts textual data into numerical format using TF-IDF (Term Frequency-Inverse Document Frequency).
2. Label Encoding:- Converts categorical labels into numerical values.:-
   - The target column, Email Type is encoded using **LabelEncoder:**
   - 0: Phishing
   - 1: Safe
   - These labels serve as the dependent variable for the classification models.

```
# Convert categorical labels into numerical
lbl = LabelEncoder()
df['Email Type'] = lbl.fit_transform(df['Email Type'])  # 0 denotes phishing and 1 denotes safe
```

Figure 7 - Label Encoding Phishing Email Dataset

2.1.6 Machine Learning Workflow

Notebook Workflow

Data Exploration:

- Visualize the distribution of email types.
- Check for missing values or inconsistencies.

Model Pipelines:

Pipelines for classifiers:

- Naive Bayes

- Logistic Regression
- SGD Classifier
- XGBoost
- Decision Tree
- Random Forest
- MLP Classifier

Model Training and Evaluation:

- Train pipelines using the training data.
- Evaluate models with:
    1. Accuracy Score
    2. F1 Score
    3. Confusion Matrices

Table 1- Model Training and Evaluation (Phishing Email Dataset)

Model	Accuracy (%)	Recall (Phishing/Safe)	F1-Score (Phishing/Safe)
SGD Classifier	97.05	0.98 / 0.96	0.96 / 0.98
MLP Classifier	97.05	0.98 / 0.96	0.96 / 0.98
Naive Bayes	95.12	0.92 / 0.97	0.94 / 0.96
Random Forest	95.95	0.97 / 0.95	0.95 / 0.97
XGBoost	95.66	0.97 / 0.95	0.95 / 0.96
Decision Tree	91.34	0.92 / 0.91	0.89 / 0.93
Logistic Regression	95.60	0.92 / 0.98	0.94 / 0.96

Best Model: SGD Classifier or MLP Classifier

- Both SGD Classifier and MLP Classifier perform identically with the highest accuracy (97.05%), strong recall and excellent F1-scores.

1. Naive Bayes

```
naive_bayes - Accuracy: 95.12%, F1 Score: 96.01%
Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.92      0.94      1477
           1       0.95      0.97      0.96      2253

    accuracy                           0.95      3730
   macro avg       0.95      0.95      0.95      3730
weighted avg       0.95      0.95      0.95      3730
```

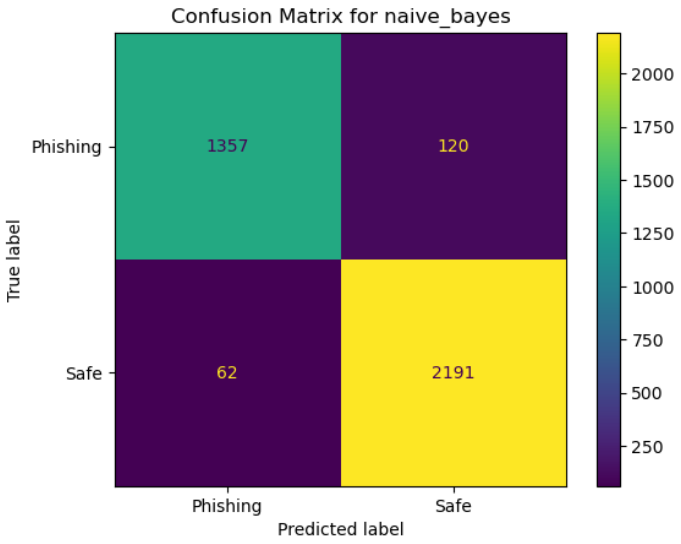

Figure 8 - Naive Bayes Confusion Matrix

2. Logistic Regression

```
logistic_regression - Accuracy: 95.60%, F1 Score: 96.43%
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.92      0.94      1477
           1       0.95      0.98      0.96      2253

    accuracy                           0.96      3730
   macro avg       0.96      0.95      0.95      3730
weighted avg       0.96      0.96      0.96      3730
```

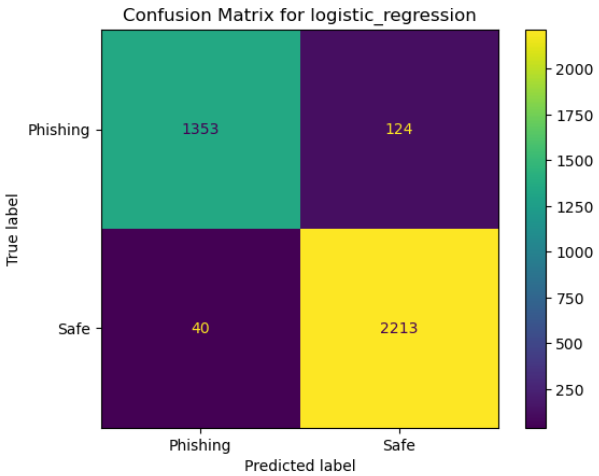

Figure 9 - Logistic Regression Confusion Matrix

3. SGD Classifier

```
sgd_classifier - Accuracy: 97.05%, F1 Score: 97.53%
Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.98      0.96      1477
           1       0.99      0.96      0.98      2253

    accuracy                           0.97      3730
   macro avg       0.97      0.97      0.97      3730
weighted avg       0.97      0.97      0.97      3730
```

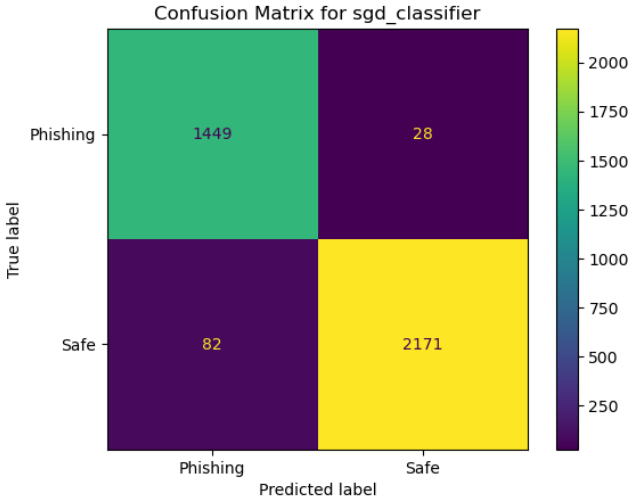

Figure 10 - SGD Confusion Matrix

4. XGBoost

```
xgboost - Accuracy: 95.66%, F1 Score: 96.34%
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.97      0.95      1477
           1       0.98      0.95      0.96      2253

    accuracy                           0.96      3730
   macro avg       0.95      0.96      0.96      3730
weighted avg       0.96      0.96      0.96      3730
```

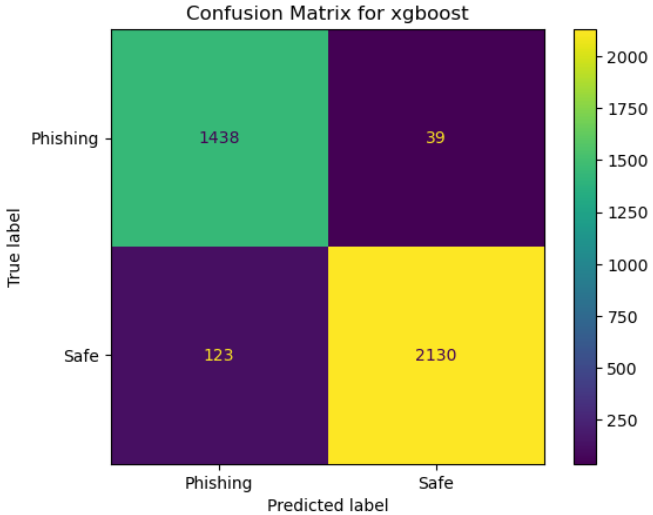

Figure 11 - XGBoost Confusion Matrix

5. Decision Tree

```
decision_tree - Accuracy: 91.34%, F1 Score: 92.67%
Classification Report:
              precision    recall  f1-score   support

           0       0.87      0.92      0.89      1477
           1       0.95      0.91      0.93      2253

    accuracy                           0.91      3730
   macro avg       0.91      0.92      0.91      3730
weighted avg       0.92      0.91      0.91      3730
```

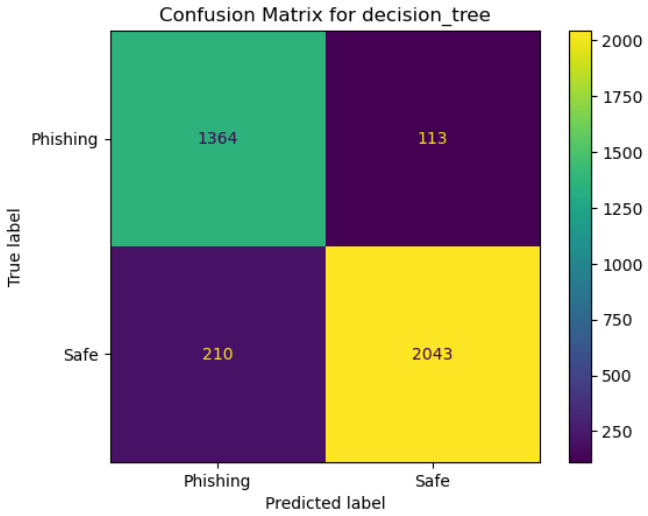

Figure 12 - Decision Tree Confusion Matrix

6. Random Forest

```
random_forest - Accuracy: 95.95%, F1 Score: 96.61%
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.97      0.95      1477
           1       0.98      0.95      0.97      2253

    accuracy                           0.96      3730
   macro avg       0.96      0.96      0.96      3730
weighted avg       0.96      0.96      0.96      3730
```

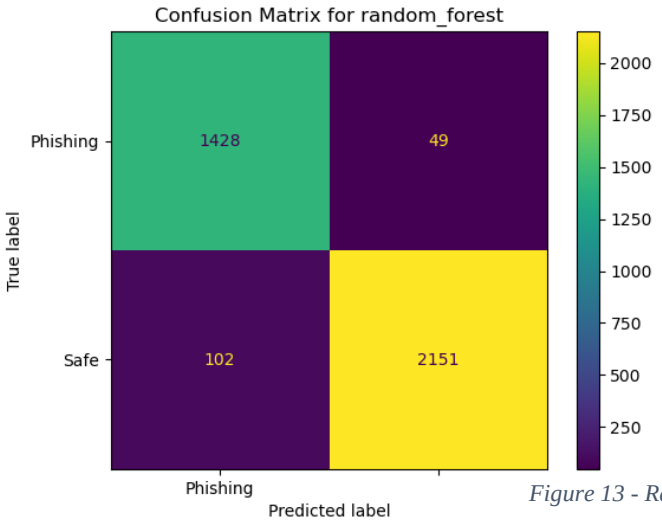

Figure 13 - Random Forest Confusion Matrix

7. MLP Classifier

```
mlp - Accuracy: 97.05%, F1 Score: 97.53%
Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.98      0.96      1477
           1       0.99      0.96      0.98      2253

    accuracy                           0.97      3730
   macro avg       0.97      0.97      0.97      3730
weighted avg       0.97      0.97      0.97      3730
```

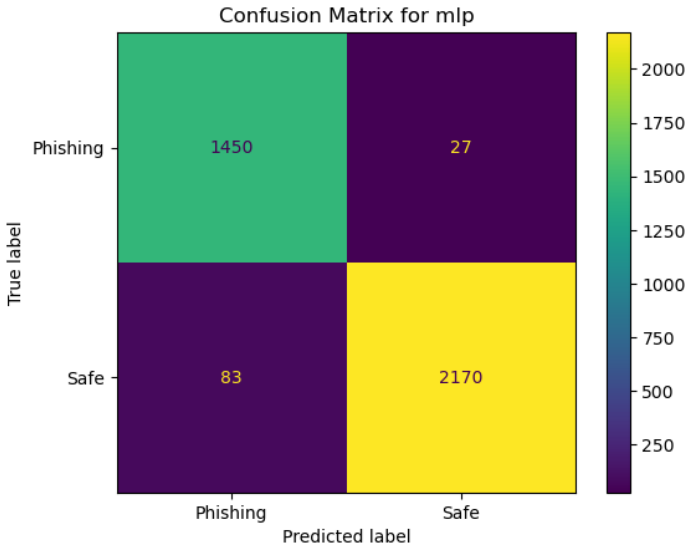

Figure 14 - MLP Confusion Matrix

Model Persistence:

- Save trained pipelines as **.pkl** files using **joblib**.

```
# Save the model
joblib.dump(pipeline, f'{name}_pipeline.pkl')
```

Figure 15 - Save trained pipelines as .pkl files using joblib

Comparison of Model Performance:

- Visualize and compare model accuracy using bar charts.

2.1.7 Deployment
Integration Steps

1. **Load Saved Model:**
2. **Predict New Emails:**
3. **Web Service Integration:**
    - Use **Flask** to build an API for real-time email classification.
    - Example endpoint:

```python
from flask import Flask, request, jsonify
from flask_cors import CORS
import joblib
import pandas as pd
import re
import tldextract
app = Flask(__name__)
CORS(app, resources={r"/*": {"origins": "*"}})
email_model = joblib.load("./models/sgd_classifier_pipeline.pkl")
@app.route("/predict/email", methods=["POST"])
def predict_email():
    try:

        email_text = None
        if request.is_json:
            email_text = request.json.get("email_text")
        else:
            email_text = request.form.get("email_text")


        if email_text is None:
            return jsonify({"error": "No email text provided"}), 400


        is_valid, error_message = is_valid_email_content(email_text)
        if not is_valid:
            return jsonify({"error": error_message}), 400


        prediction = email_model.predict([email_text])[0]
        label = "Safe Email" if prediction == 1 else "Phishing Email"


        return jsonify({
            "prediction": label,
            "status": "success"
        })

    except Exception as e:

        return jsonify({"error": f"Processing error: {str(e)}"}), 500
if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=5000)
```

Figure 16 - Flask to build an API for real-time email classification

2.1.8 Important

- Preprocessing ensures text data is clean, consistent and ready for analysis.
- Feature engineering transforms text into numerical data that models can process.
- Comparing multiple models identifies the best-performing classifier.
- Saved pipelines enable seamless integration into production environments.

2.1.9 Key Libraries Used

- pandas:- For data manipulation.
- scikit-learn:- For preprocessing, modeling and evaluation.
- xgboost:- For gradient-boosted decision trees.
- joblib:- For saving and loading models.
- matplotlib, seaborn:- For data visualization.
- wordcloud:- For visualizing text data.

2.2 URL Dataset

The URL classification model also makes use of the "Malicious and Benign URLs" dataset from Mendeley Data. A rich dataset for developing machine-learning-based approaches for phishing detection.

Key characteristics of the URL dataset:

- **Source**: Mendeley Data repository
- **Composition**: Collection of verified malicious and benign URLs
- **Features**: Nine engineered features including URL length, domain properties and special character analysis
- **Format**: Structured CSV format
- **Total URLs**: Comprehensive collection with balanced class distribution
- **Citation**: Kaitholikkal, J. K. S., & B, A. (2024). Phishing URL dataset. Mendeley Data. https://doi.org/10.17632/vfszbj9b36.1

2.2.1 Dataset Structure

The URL dataset.csv file contains **450,176 rows and 2 columns**.

Columns:

- **url**: The URL string to be analyzed.
- **type**: The classification label, which can be either "legitimate" or "phishing."

First Few Rows of the Dataset

	url	type
0	https://www.google.com	legitimate
1	https://www.youtube.com	legitimate
2	https://www.facebook.com	legitimate
3	https://www.baidu.com	legitimate
4	https://www.wikipedia.org	legitimate

Figure 17 - First Few Rows of the URL Dataset

Distribution of URLs per Type

```
Distribution of URLs per Type:
type
legitimate    345738
phishing      104438
Name: count, dtype: int64
```

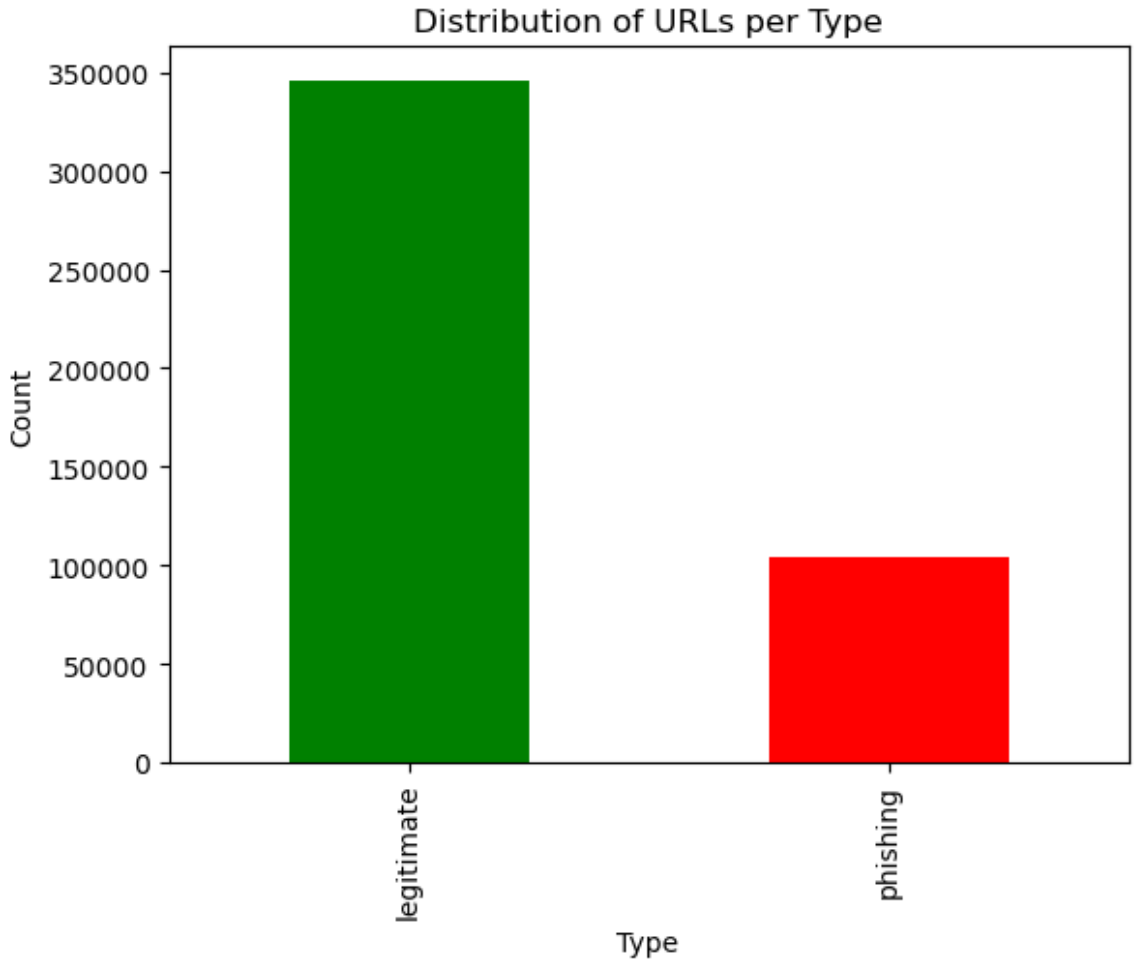

Figure 18 - Distribution of URLs per Type (URL Dataset)

2.2.2 Steps to Use the Dataset

1. Place the dataset in the data/ directory of your project repository.
2. Reference the file path correctly in your code or notebook.

```python
# Load the dataset
import pandas as pd
data = pd.read_csv('dataset_path')
```

Figure 19 - Load URL Dataset

2.2.3 Exploration

- Visualize the distribution of URL types:

```
# Distribution of URLs per Type
print("\nDistribution of URLs per Type:")
print(data['type'].value_counts())

# Visualize the distribution
data['type'].value_counts().plot(kind='bar', color=['green', 'red'])
plt.title('Distribution of URLs per Type')
plt.xlabel('Type')
plt.ylabel('Count')
plt.show()
```

Figure 20 - Visualize the distribution of URL types

- Check for missing values:

```
# Check for missing values
print("\nChecking for Missing Values:")
print(data.isnull().sum())
```

Figure 21 - Check for missing values (URL Dataset)

2.2.4 Data Preprocessing and Feature Engineering

Data Preprocessing

1. Encode Labels
   - Convert the type column into numerical labels:

     0 for phishing.

     1 for legitimate.

```
# Encode the 'type' column
label_encoder = LabelEncoder()
data['label'] = label_encoder.fit_transform(data['type'])
data = data.drop(columns=['type'])
```

Figure 22 - Encode Labels (URL Dataset)

2. Handle Missing Values
   - Remove rows with missing data, if any.

Feature Engineering

1. Extracted Features
   - **url_length**: The total length of the URL.

- **num_dots**: Number of dots (.) in the URL.
- **num_hyphens**: Number of hyphens (-) in the URL.
- **num_underscores**: Number of underscores (_) in the URL.
- **num_digits**: Count of numeric digits in the URL.
- **num_special_chars**: Count of non-alphanumeric characters (symbols) in the URL.
- **domain_length**: Length of the domain name.
- **subdomain_length**: Length of the subdomain.
- **path_length**: Length of the URL path after the domain.

```
# Function to extract features from a URL
def extract_features(url):
    features = {}
    features['url_length'] = len(url)
    features['num_dots'] = url.count('.')
    features['num_hyphens'] = url.count('-')
    features['num_underscores'] = url.count('_')
    features['num_digits'] = sum(c.isdigit() for c in url)
    features['num_special_chars'] = len(re.findall(r'[^A-Za-z0-9]', url))

    ext = tldextract.extract(url)
    features['domain_length'] = len(ext.domain)
    features['subdomain_length'] = len(ext.subdomain)
    features['path_length'] = len(url.split('/', 3)[-1]) if '/' in url else 0

    return features
```

Figure 23 - Feature Engineering URL Dataset

2. Apply Feature Extraction
    - Transform the dataset by extracting features from each URL
3. Analyze Features
    - Summarize and visualize feature distributions
4. Prepare Training Data
    - Separate features (X) and labels (y):

```
# Define features (X) and target (y)
X = final_data.drop(columns=['label'])
y = final_data['label']
```

Figure 24 - Separate features (X) and labels (y)

2.2.5 Machine Learning Workflow

Notebook Workflow

Data Exploration:

- Visualize the distribution of url types.
- Check for missing values or inconsistencies.

2.2.6 Data Splitting

- Split the dataset into training and testing subsets:

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 25 - Data Splitting

2.2.7 Model Training

Define Models

- Use multiple classifiers for comparison

Train Models

- Train each model on the training dataset

```
for idx, (model_name, model) in enumerate(models.items(), 1):
    # Train the model
    print(f"\nTraining {model_name}...")
    model.fit(X_train, y_train)
```

Figure 26 - Train each model on the training dataset

2.2.8 Evaluation

Accuracy and Confusion Matrix

- Calculated accuracy and visualized confusion matrices.

Table 2 - Model Training and Evaluation (URL Dataset)

Model	Accuracy	Recall (Sensitivity)	F1 Score
Logistic Regression	0.8914	0.5878	0.7173
Decision Tree	0.9657	0.9198	0.9263
Random Forest	0.9666	0.8743	0.9246
XGBoost	0.9672	0.8815	0.9266

Metrics Comparison (Random Forest vs. XGBoost):

1.Accuracy:-

- Random Forest:- 0.9666
- XGBoost:- 0.9672
- Difference:-Very small, only 0.0006, which is likely negligible in practical terms.

2.Recall (Sensitivity):-

- Random Forest:- 0.8743
- XGBoost:- 0.8815
- Difference:- Also very small (0.0072), which might not have a significant impact depending on the application.

3. F1 Score:-

- Random Forest:- 0.9246
- XGBoost:- 0.9266
- Difference:- Just 0.002, which indicates nearly identical performance in balancing precision and recall.

Why Random Forest Could Be Chosen:-

1. **Model Simplicity:-** Random Forest is often easier to interpret and tune than XGBoost, which can require more expertise for optimization.
2. **Stability:-** Random Forest may produce more stable results across different datasets or variations, making it a more robust choice.
3. **Computation:-** Random Forest is generally faster to train and deploy, while XGBoost can be computationally intensive.

When Random Forest is Preferable:

- If the performance differences (accuracy, recall, F1) are marginal and value simplicity, stability, or computational efficiency, **Random Forest** is an excellent choice.
- Ultimately, the “best” model isn’t always about the absolute highest metrics but what aligns best with your priorities and constraints.

2.2.9 Detailed Metrics

- Precision, recall and F1 Score:

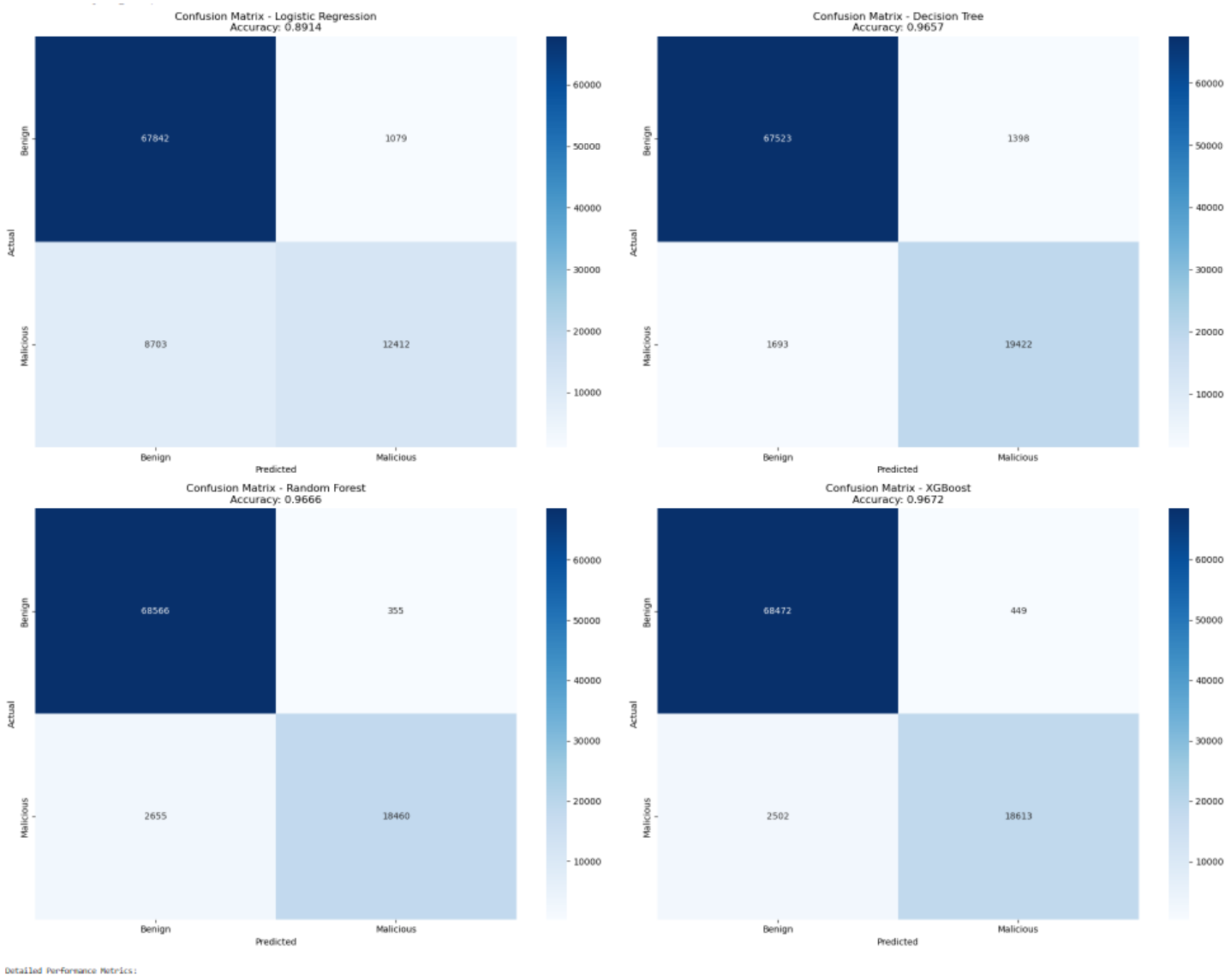

Figure 27 - Confusion Matrices of URL Dataset Train

```
Logistic Regression Metrics:
Accuracy: 0.8914
True Negatives (Benign correctly identified): 67842
False Positives (Benign misclassified as Malicious): 1079
False Negatives (Malicious misclassified as Benign): 8703
True Positives (Malicious correctly identified): 12412
Precision: 0.9200
Recall (Sensitivity): 0.5878
Specificity: 0.9843
F1 Score: 0.7173
--------------------------------------------------

Decision Tree Metrics:
Accuracy: 0.9657
True Negatives (Benign correctly identified): 67523
False Positives (Benign misclassified as Malicious): 1398
False Negatives (Malicious misclassified as Benign): 1693
True Positives (Malicious correctly identified): 19422
Precision: 0.9329
Recall (Sensitivity): 0.9198
Specificity: 0.9797
F1 Score: 0.9263
--------------------------------------------------

Random Forest Metrics:
Accuracy: 0.9666
True Negatives (Benign correctly identified): 68566
False Positives (Benign misclassified as Malicious): 355
False Negatives (Malicious misclassified as Benign): 2655
True Positives (Malicious correctly identified): 18460
Precision: 0.9811
Recall (Sensitivity): 0.8743
Specificity: 0.9948
F1 Score: 0.9246
--------------------------------------------------

XGBoost Metrics:
Accuracy: 0.9672
True Negatives (Benign correctly identified): 68472
False Positives (Benign misclassified as Malicious): 449
False Negatives (Malicious misclassified as Benign): 2502
True Positives (Malicious correctly identified): 18613
Precision: 0.9764
Recall (Sensitivity): 0.8815
Specificity: 0.9935
F1 Score: 0.9266
```

Figure 28 - Detailed Performance Metrics

2.2.10 Deployment

Deploy the trained model for real-time URL classification.

Model Saving

1. Save the trained model using joblib.

```
# Save the model
model_file = f"{model_name.replace(' ', '_').lower()}_model.pkl"
joblib.dump(model, model_file)
print(f"{model_name} saved as '{model_file}'")
```

Figure 29 - Save the trained model using joblib

2. Load the saved model for predictions.

API Integration

1. Build API using Flask:- Create an endpoint for real-time URL classification:

```
from flask import Flask, request, jsonify
from flask_cors import CORS
import joblib
import re
app = Flask(__name__)
CORS(app, resources={r"/*": {"origins": "*"}})
url_model = joblib.load("./models/random_forest_model.pkl")
def extract_url_features(url):
    features = {}
    features['url_length'] = len(url)  # Length of the URL
    features['num_dots'] = url.count('.')  # Number of dots in the URL
    features['num_hyphens'] = url.count('-')  # Number of hyphens in the URL
    features['num_underscores'] = url.count('_')  # Number of underscores in the URL
    features['num_digits'] = sum(c.isdigit() for c in url)  # Number of digits in the URL
    features['num_special_chars'] = len(re.findall(r'[^A-Za-z0-9]', url))  # Number of sp

    ext = tldextract.extract(url)
    features['domain_length'] = len(ext.domain)  # Length of the domain part of the URL
    features['subdomain_length'] = len(ext.subdomain)  # Length of the subdomain part of
    features['path_length'] = len(url.split('/', 3)[-1]) if '/' in url else 0  # Length o

    return features
@app.route("/predict/url", methods=['POST'])
def predict_url():
    try:
        try:
            data = request.get_json(force=True)
        except Exception:
            return jsonify({"error": "Invalid JSON format"}), 400
        url = data.get("url")
        if not url:
            return jsonify({"error": "No URL provided"}), 400
        is_valid, error_message = is_valid_url(url)
        if not is_valid:
            return jsonify({"error": error_message}), 400
        features = extract_url_features(url)
        features_df = pd.DataFrame([features])
        prediction = url_model.predict(features_df)
        result = "Phishing" if prediction[0] == 1 else "Safe"
        return jsonify({
            "url": url,
            "prediction": result,
            "status": "success"
        })
    except Exception as e:
        return jsonify({"error": f"Processing error: {str(e)}"}), 500

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=5000)
```

Figure 30 - Build API using Flask Create an endpoint for real-time URL classification

2. Run the Flask Server:- python app.py

2.2.11 Important

1. Feature Extraction:- Focused on structural aspects of URLs.
2. Model Performance:- Multiple models were trained and compared.
3. Deployment:- The trained model was deployed as a REST API for real-time predictions.

2.2.12 Libraries Used

- pandas: Data manipulation.
- scikit-learn: Preprocessing, modeling and evaluation.
- xgboost: Advanced gradient boosting classifier.
- tldextract: URL parsing and domain extraction.
- joblib: Model persistence.
- Flask: API development.

Both datasets underwent rigorous preprocessing and feature engineering to ensure optimal model training and performance. The selection of these datasets was based on their comprehensiveness, real-world relevance and the quality of their labeling.

3.0 Wireframe Layout

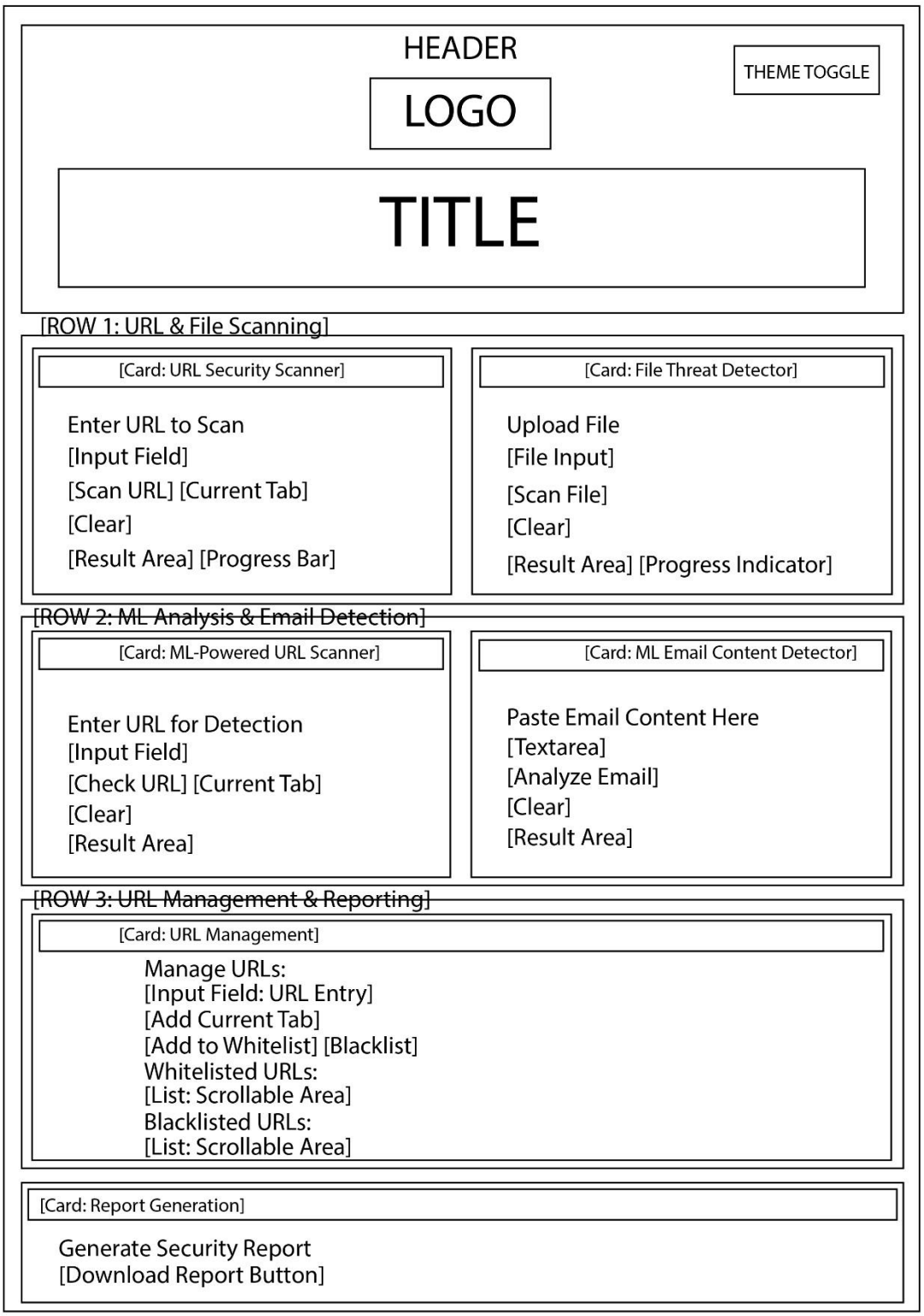

Figure 31 - Wireframe Layout

4.0 UI Mockups

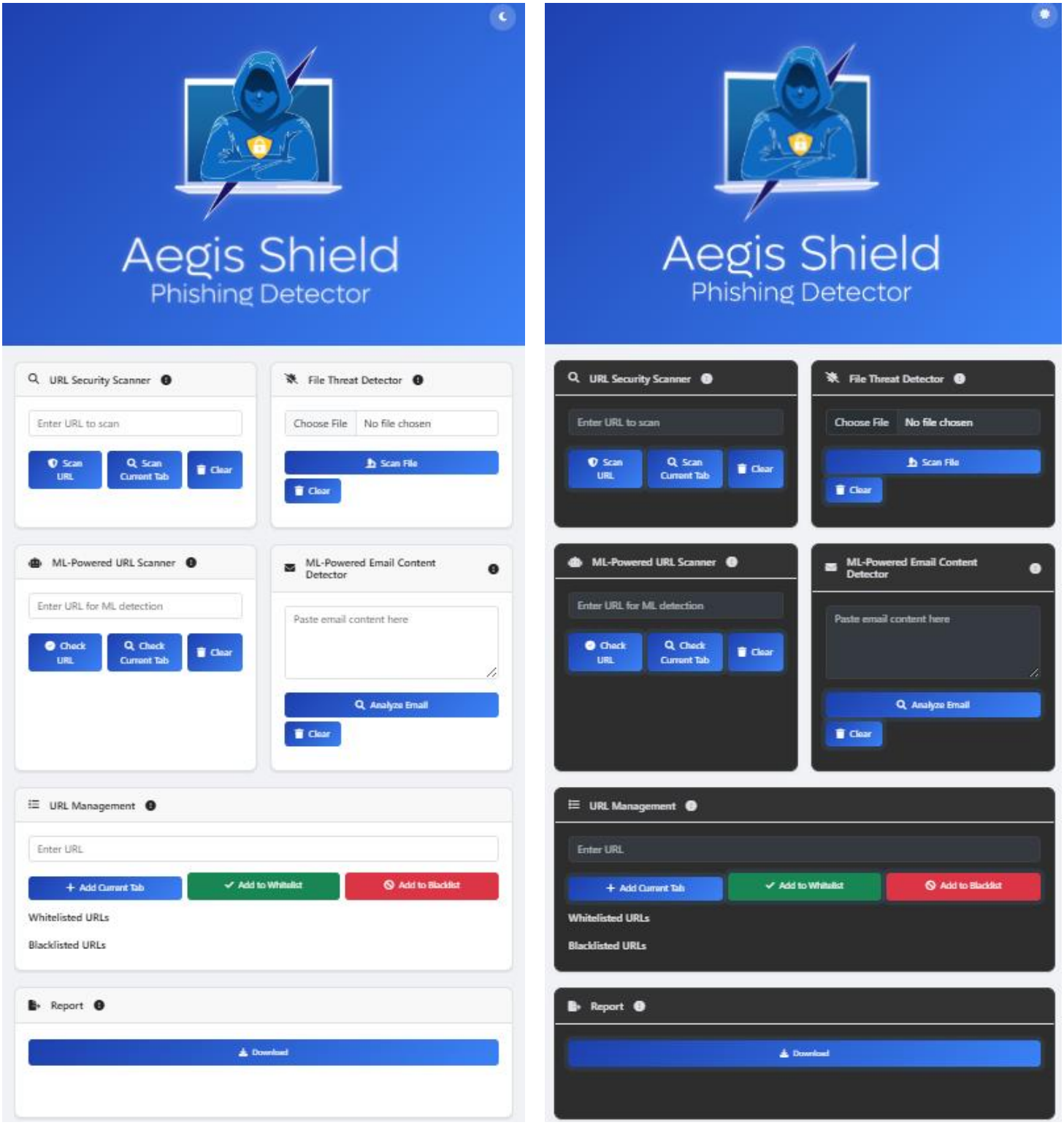

Figure 32 - UI Mockups

4.1 UI Design with HCI Perspective

Usability and accessible design was a primary design goal for the Aegis Shield user interface. Following the principles of Human-Computer Interaction (HCI), the design helps to create a smooth experience while reducing cognitive load and potential errors. The tables below highlight areas where the UI corresponds with major HCI principles.

4.1.1 HCI Principles in the UI Design

Table 3 - HCI Principles in the UI Design

HCI Principle	Implementation	HCI Alignment
1. Visibility of System Status	• Progress bars for scans. • Real-time results displayed for URL and email analysis.	• Excellent:- Providing users with visual feedback during scans reduces uncertainty.
2. Match Between System and Real-World Language	• Plain English labels (e.g., “Scan URL,” “Analyze Email”).	• Good:- Simple and familiar terms make the interface intuitive.
3. User Control and Freedom	• Clear buttons for actions (e.g., “Clear,” “Scan Current Tab”). • Whitelist/Blacklist management.	• Strong:- Users can reverse actions (e.g., remove URLs from the list) and control features.
4. Consistency and Standards	• Consistent styling across cards. Standard icons (e.g., Font Awesome).	• Very Strong:- Consistency across the UI enhances usability.
5. Error Prevention	• Input validation for URLs (e.g., requiring “http://” or “https://”).	• Good:- Basic validation reduces errors.
6. Recognition Rather Than Recall	• Buttons and tooltips provide actionable labels. • Whitelist/Blacklist management shows the current list of URLs.	• Good:- Displaying information (e.g., lists and tooltips) minimizes cognitive load.
7. Flexibility and Efficiency of Use	• Theme toggle for light/dark mode. • Input fields for manual entry and quick actions for scanning current tabs.	• Strong:- Options for both novices and experts improve usability.
8. Aesthetic and Minimalist Design	• Clean card-based layout with minimal distractions. • Color-coded statuses (green for safe, red for malicious).	• Excellent:- The interface avoids clutter and emphasizes critical actions.

9. Help Users Recognize, Diagnose and Recover from Errors	• Basic error handling (e.g., alerts for invalid URLs).	• Moderate:- The current implementation alerts users but doesn't provide detailed help.

4.1.2 HCI Principles for Color Usage

Table 4 - HCI Principles for Color Usage

HCI Principle	Implementation	HCI Alignment
1. Contrast and Accessibility	• Light and dark themes. Blue buttons and white text in both modes.	• Moderate:- Good color contrast, but some areas in dark mode (e.g., gray sections) may not meet WCAG standards.
2. Color Coding for Feedback	• Green for "Safe," red for "Phishing." Blue for primary actions.	• Good:- Familiar feedback colors improve usability.
3. Minimal Use of Bright Colors	• Minimal use of red, only for critical actions (e.g., blacklist).	• Excellent:- Avoids overwhelming the user with too many bright or alarming colors.
4. Consistent Color Hierarchy	• Reuse of blue for buttons and green for positive actions. Consistent styling across elements.	• Very Strong:- Consistency helps users quickly recognize and interact with elements.

5.0 Use Case Diagram

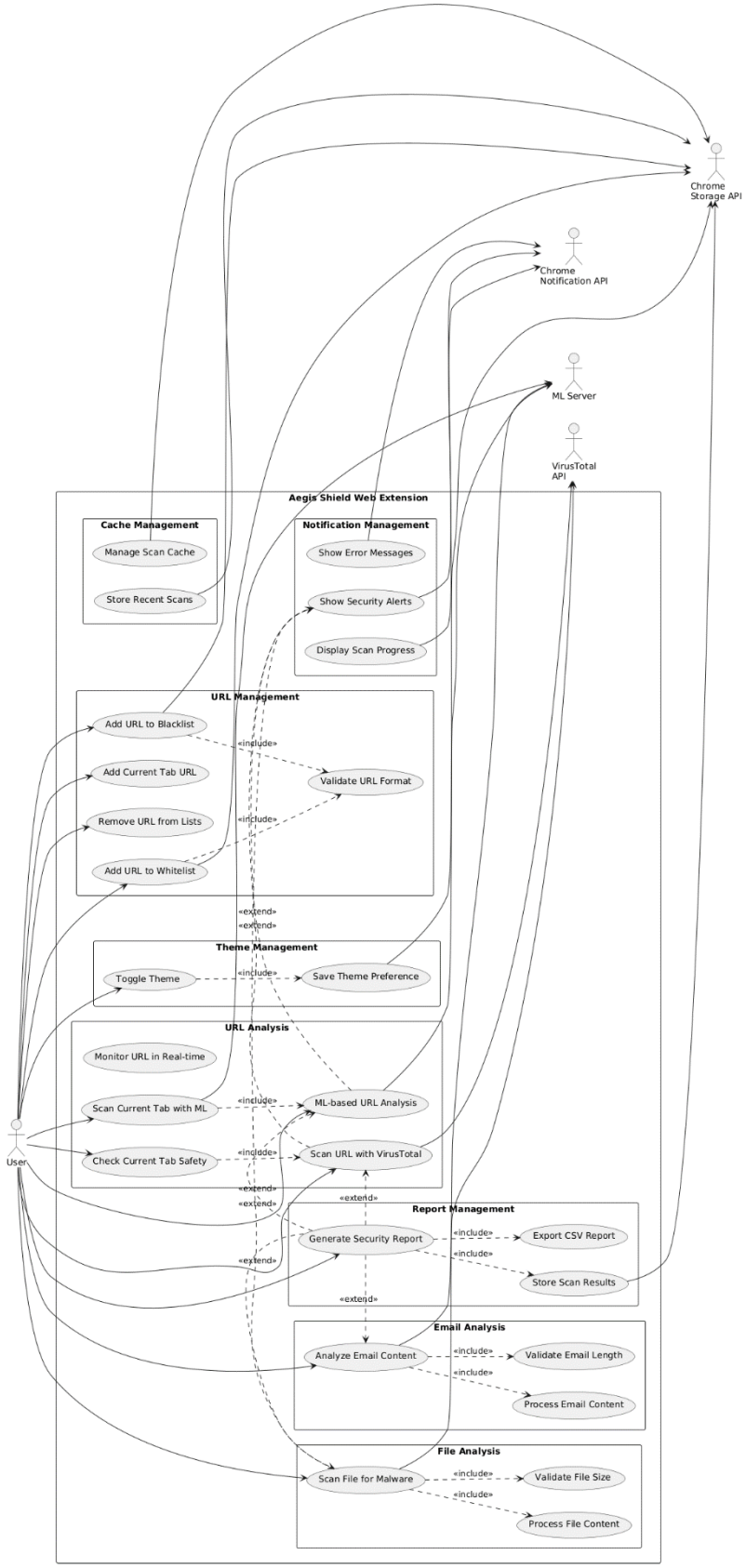

Figure 33 - Use Case Diagram

5.1 Actors

1. **User:-**
    - Primary actor who interacts with the extension's functionalities.
2. **Chrome Storage API:-**
    - External system where data like preferences and cached scans are stored.
3. **Chrome Notification API:-**
    - Used to show notifications to the user for events like errors or security alerts.
4. **ML Server:-**
    - Machine Learning server that analyzes URLs or files for security risks.
5. **VirusTotal API:-**
    - External API used to scan URLs for potential threats.

5.2 Modules and Use Cases

1. **Cache Management:-**
    - Manage Scan Cache: Handles the storage and removal of previously scanned URLs.
    - Store Recent Scans: Saves recent scan data for user reference.
2. **Notification Management:-**
    - Show Error Messages: Notifies users of errors during scanning.
    - Show Security Alerts: Displays alerts for potential security threats.
    - Display Scan Progress: Indicates the progress of an ongoing scan.
3. **URL Management:-**
    - Add URL to Blacklist: Allows users to add URLs to a blacklist for blocking.
    - Add Current Tab URL: Adds the currently open URL to a specified list (e.g., whitelist or blacklist).
    - Remove URL from Lists: Removes URLs from either whitelist or blacklist.
    - Add URL to Whitelist: Permits URLs to bypass scans.
    - Validate URL Format (Included Use Case): Ensures URLs follow proper format before processing.

4. **Theme Management:-**
    - Toggle Theme: Switch between light and dark mode.
    - Save Theme Preference: Stores the selected theme for future sessions.
5. **URL Analysis:-**
    - Monitor URL in Real-time: Watches URLs dynamically to detect threats.
    - Scan Current Tab with ML: Analyzes the open tab's URL using machine learning.
    - Check Current Tab Safety: Confirms whether the current tab is secure.
    - ML-based URL Analysis (Included Use Case): Uses machine learning to assess URLs.
    - Scan URL with VirusTotal: Leverages the VirusTotal API for a secondary analysis.
6. **Report Management:-**
    - Generate Security Report: Compiles a report of scan results.
    - Export CSV Report (Included Use Case): Allows the user to download reports in CSV format.
    - Store Scan Results (Included Use Case): Saves the analysis results.
7. **Email Analysis:-**
    - Analyze Email Content: Scans email data for suspicious links or text.
    - Validate Email Length (Included Use Case): Ensures proper email length for analysis.
    - Process Email Content (Included Use Case): Prepares email content for analysis.
8. **File Analysis:-**
    - Scan File for Malware: Checks uploaded files for malicious content.
    - Validate File Size (Included Use Case): Ensures file size is within limits for scanning.
    - Process File Content (Included Use Case): Prepares the file for malware analysis.

5.3 Relationships

Includes:- Indicates a required dependency for completing a use case. For example:

- Validate URL Format is required when adding URLs to the whitelist or blacklist.
- Validate File Size is essential before processing file content for malware scanning.

Extends:- Indicates optional functionality or specific scenarios. For example:

- Export CSV Report extends the Generate Security Report use case.
- Scan Current Tab with ML extends Monitor URL in Real-time for detailed ML analysis.

5.4 External System Interactions

- **Chrome Storage API:-** Used for caching scans, storing preferences and saving results.
- **Chrome Notification API:-**Triggers error and alert messages.
- **ML Server:-** Supports machine learning-based analyses for URLs and files.
- **VirusTotal API:-** Provides an additional layer of URL safety checks.

6.0 Sequence Diagram

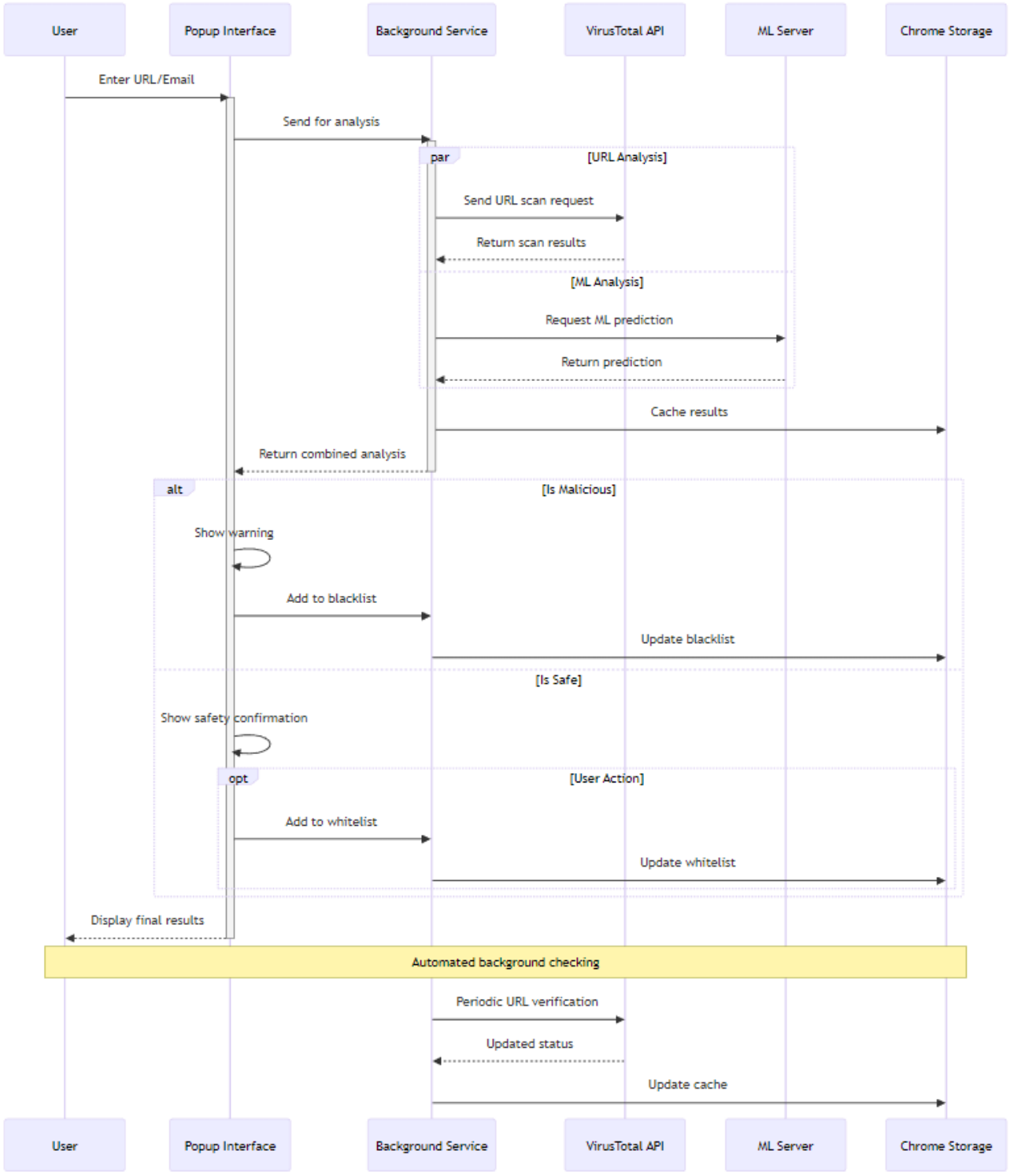

Figure 34 - Sequence Diagram

7.0 Folder Architecture

```
Aegis-Shield/
├── server/
│   ├── app.py
│   └── models/
│       ├── random_forest_model.pkl
│       └── sgd_classifier_pipeline.pkl
│
├── config/
│   └── rules.json
│
├── icons/
│   ├── icon16.png
│   ├── icon48.png
│   ├── icon128.png
│   ├── danger16.png
│   ├── danger48.png
│   └── danger128.png
│
├── logo/
│   ├── logov1.png
│   └── title.png
│
├── background.js
├── popup.js
├── bootstrap.bundle.min.js
├── popup.css
├── popup.html
├── manifest.json
└── requirments.txt
```

Figure 35 - Folder Architecture

8.0 Functionalities of codes

8.1 app.py:

- Implements a Flask server with API endpoints for URL and email phishing detection
- Handles URL validation and feature extraction for machine learning predictions
- Includes two main endpoints: /predict/url for URL analysis and /predict/email for email content analysis
- Uses pre-trained machine learning models (Random Forest for URLs and SGD Classifier for emails)
- Implements cross-origin resource sharing (CORS) for API access
- Includes API key authentication system

8.2 background.js:

- Core engine for the Chrome extension's background operations
- Implements real-time URL monitoring and security scanning
- Manages integration with VirusTotal API for threat detection
- Handles whitelist/blacklist management for URLs
- Implements file scanning capabilities for malware detection
- Includes a caching system to optimize performance
- Manages Chrome notifications for security alerts
- Handles URL filtering rules using Chrome's **declarativeNetRequest** API

8.3 bootstrap.bundle.min.js:

- Provides the minified version of Bootstrap framework with all its components
- Includes Popper.js for tooltip and popover functionality
- Contains all Bootstrap's JavaScript plugins in a single bundled file
- Enables responsive design and interactive UI components

8.4 manifest.json:

- Defines the Chrome extension's configuration and permissions
- Specifies version information and description
- Lists required permissions for functionality like storage and notifications
- Defines host permissions for API access
- Configures background scripts and popup interface

- Sets up declarative network request rules

8.5 popup.css:

- Defines the styling for the extension's popup interface
- Implements dark/light theme support
- Creates animations for logo and UI elements
- Styles cards, buttons and form elements
- Defines status indicators for scan results
- Implements responsive design for different screen sizes

8.6 popup.html:

- Creates the main user interface for the extension
- Organizes different scanning tools in card layouts
- Implements URL scanning interface
- Provides email content analysis section
- Includes file scanning capabilities
- Shows whitelist/blacklist management interface
- Implements report generation functionality

8.7 popup.js:

- Handles all user interactions in the popup interface
- Manages theme switching functionality
- Implements URL and file scanning logic
- Handles ML-based URL and email analysis
- Manages whitelist/blacklist operations
- Implements report generation and download
- Handles form validation and error display
- Manages communication with background script

8.8 rules.json:

- Defines default URL filtering rules for the extension
- Contains rules for blocking specific URLs
- Uses **declarativeNetRequest** API format

- Provides template for dynamic rule generation

9.0 Technical Requirements

9.1 Prerequisites

- Python 3.8 or higher
- Chrome web browser
- Active internet connection
- VirusTotal API key
- Anaconda Distribution (Visual Studio Code, JupyterLab)

9.2 Dependencies

- **Flask==2.1.3**
- **Flask-Cors==3.0.10**
- **joblib==1.3.2**
- **numpy==1.24.3**
- **pandas==2.1.4**
- **scikit-learn==1.3.2**
- **tldextract==3.4.0**
- **requests==2.31.0**
- **matplotlib==3.7.1**
- **seaborn==0.12.2**
- **xgboost==2.0.1**
- **wordcloud==1.9.2**
- **validators==0.20.0**

10.0 Setup Instructions

1. Clone the repository:
   - git clone:- https://github.com/sudamsanjula/Applied-Project.git
   - cd phishing-detection-extension
2. Install Python Dependecies
   - **Pip install -r requirments.txt**
3. Load the Extension

I. Open Chrome

II. Navigate to extensions page
III. Enable developer mode
IV. Load unpacked extension
V. Select the extension directory

4. Run the Flask Server
   - **python app.py**

11.0 Usage Steps

11.1 URL Scanning

1. Click the extension icon to open the interface
2. Enter a URL manually or scan the current tab
3. View real-time security analysis results
4. Manage URL whitelist/blacklist as needed

11.2 Email Analysis

1. Open the email analysis section
2. Paste email content into the analysis field
3. Click "Analyze Email" for instant evaluation
4. Review detailed security assessment

11.3 Security Reports

1. Access the reporting section
2. Select desired date range and security metrics
3. Generate comprehensive security report
4. Export results in CSV format

12.0 Architecture

The extension implements a modular architecture:-

- Frontend UI (Popup):- Handles user interaction and displays results (HTML, CSS, JS).
- Background Script:- Acts as a middleman to manage data flow between the UI, storage and APIs.
- Backend API:- Processes data and applies machine learning models.
- Declarative Ruleset:- Enforces network filtering rules

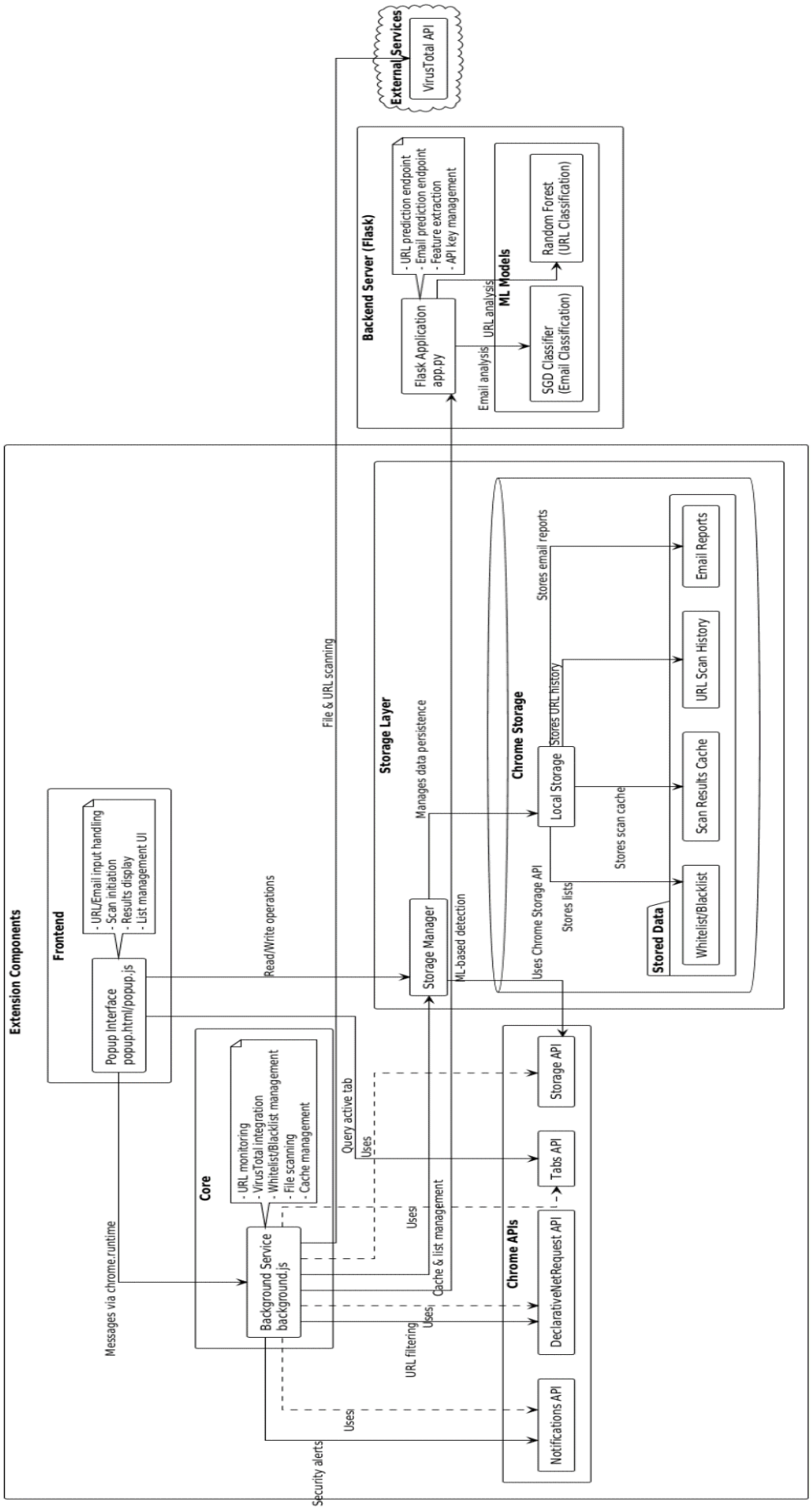

Figure 36 - Architecture

12.1 Extension Components

1. **Frontend (popup.html/popup.js):-**
    - Handles user interactions, such as inputting URLs/emails for scanning or managing whitelist/blacklist.
    - Provides a user interface for scan initiation, result display and list management.
    - Communicates with the core (background.js) using **chrome.runtime** messaging.

    Example functionalities:-

    - Allows users to initiate file and URL scans.
    - Displays results returned from the backend or local storage.
2. **Core (background.js):-**
    - Acts as the brain of the extension, managing real-time operations and communication between components.
    - Responsibilities:-
        1. URL Monitoring:- Tracks tab updates and scans URLs using VirusTotal or the backend API.
        2. Whitelist/Blacklist Management:- Adds/removes URLs to/from user-defined lists.
        3. File Scanning:- Handles malware detection for uploaded files.
        4. Cache Management:- Avoids redundant scans by using a caching mechanism.
    - Interacts with Chrome APIs for notifications, URL filtering and storage.

12.2 Chrome APIs

- Integrates the following APIs to enable core functionalities:-
    1. Notifications API:- Sends alerts for malicious URLs or successful scans.
    2. DeclarativeNetRequest API:- Dynamically blocks or allows URLs based on whitelist/blacklist rules.
    3. Storage API:- Saves and retrieves user-specific data like scan results, whitelist/blacklist and email reports.
    4. Tabs API:- Monitors active tabs and performs security checks on page loads.

12.3 Storage Layer

- Manages data persistence using Chrome’s local storage.

- Data types stored:-
    1. Whitelist/Blacklist:- Maintains user-defined lists for URL filtering.
    2. Scan Results Cache:- Caches results of recent scans to optimize performance.
    3. URL Scan History:- Logs past scans for user reference.
    4. Email Reports:- Stores analysis results of email content.

12.4 Backend Server (Flask)

- Provides a powerful processing engine for phishing detection using machine learning.
- Endpoints:
    1. URL Prediction:- Validates and classifies URLs as phishing or safe.
    2. Email Prediction:- Analyzes email content for phishing patterns.
- Machine Learning Models:
    1. Random Forest:- Used for URL classification based on extracted features like length, number of dots and special characters.
    2. SGD Classifier:- Used for email content analysis to detect phishing attempts.
- Feature Extraction:
    1. URL features like domain length, number of subdomains and special characters.
    2. Textual patterns in email content for phishing markers.

12.5 External Services

- Integrates with third-party services like VirusTotal for additional threat intelligence.
    - VirusTotal API is used to cross-check URLs and files for malicious activity.

12.6 Key Data Flow

1. User Input:-
    - Users interact with the popup interface to input URLs/emails or manage lists.
    - The popup sends messages to the background script using chrome.runtime.
2. Background Operations:-
    - The background script processes these inputs, interacts with storage for whitelist/blacklist updates or forwards scan requests to the backend.
    - For real-time monitoring, it uses Chrome APIs (e.g., Tabs API) to detect and scan URLs.
3. Backend Processing:-

- For advanced analysis, the background script forwards data to the Flask backend.
- The backend processes data using ML models and sends results back to the background script.

4. Output:-
    - Results are displayed in the popup or as browser notifications, depending on the operation.

13.0 Platform Justification and Choosing the Platform

13.1 Platform Justification

13.1.1 Browser Extension (Manifest V3)

- Purpose: The browser extension operates directly in the user's browsing environment, scanning URLs in real-time to prevent phishing attempts.
- Why Manifest V3?
  - Enhanced Security:- Replaces background pages with service workers, reducing the risk of malicious extensions.
  - Efficient Rule-Based Filtering:- The declarativeNetRequest API allows URL filtering at the browser level, optimizing performance by eliminating the need for external calls for common checks.
  - Cross-Browser Support:- While built for Chrome, the extension's structure can be adapted to other Chromium-based browsers.

Key Features Supported:

- URL Monitoring:- Integration with the tabs and webNavigation APIs for real-time scanning.
- Notification System:- Alerts users of potential phishing threats.
- Rule Management:- Dynamic whitelist/blacklist support using declarativeNetRequest.

13.1.2 Backend (Python Flask Framework)

- Purpose: Provides a lightweight API backend for hosting machine learning models and performing advanced phishing detection.
- Why Flask?
  - Lightweight:- Flask's simplicity ensures that only essential components are included, reducing overhead.
  - Scalable:- Can scale vertically with more powerful servers or horizontally using load balancers and additional instances.
  - Ease of Integration:- Flask works seamlessly with libraries like joblib for model loading and pandas for data manipulation.
  - CORS Support:- Cross-Origin Resource Sharing allows secure communication between the browser extension and backend APIs.

Core Functionalities:

- Hosts machine learning models (Random Forest for URL classification, SGD Classifier for email content analysis).
- Provides endpoints for real-time URL and email classification.
- Validates user input to ensure data integrity.

13.1.3 Machine Learning Models

- Purpose: Perform intelligent phishing detection using trained classifiers.
- Why Machine Learning?
  - Dynamic Detection:- ML models adapt to new patterns in phishing attempts.
  - Feature Engineering:- Extracts URL properties and textual content features for robust analysis.

Chosen Models:

- Random Forest:- High accuracy and resilience against overfitting for URL classification.
- SGD Classifier:- Lightweight and fast for email content classification.

Preprocessing Libraries:

- tldextract for URL parsing.
- TF-IDF vectorization for email content analysis.

13.1.4 Frontend (HTML, CSS, JavaScript, Bootstrap)

- Purpose: Provide a user-friendly interface for interacting with the extension.
- Why Bootstrap?
  - Responsiveness-: Ensures the UI works across various screen sizes.
  - Ease of Use:- Pre-built components accelerate development.
  - Consistency:- The cohesive design framework enhances user experience.

Key Functionalities:-

- URL and email input fields with real-time validation.
- Buttons for actions like scanning URLs and managing lists.
- Dynamic theme switching (light/dark mode).

13.1.5 VirusTotal API

- Purpose: Augments the local detection system with a trusted external source.
- Why VirusTotal?
  - Comprehensive Database:- Provides up-to-date threat intelligence on URLs.
  - Integration:- Easy to use with RESTful APIs for threat detection.
  - Added Value:- Complements ML-based detection with external verifications.

13.2. Choosing the Platform

13.2.1 Browser-Based Extension

- Requirement: A platform capable of monitoring user activity in real-time.
- Choice:- Chrome Extensions with Manifest V3.
- Reasoning:
  - DeclarativeNetRequest APIs are optimized for security tasks.
  - Chrome Web Store provides easy distribution to end users.

13.2.2 Python Backend

- Requirement:- A lightweight server for API handling and ML model hosting.
- Choice:- Flask Framework.
- Reasoning:-
  - Low resource consumption.
  - Supports integration with machine learning workflows.
  - Simplifies REST API development.

13.2.3 Machine Learning Environment

- Requirement:- Frameworks for training and deploying models.
- Choice:- scikit-learn, pandas, joblib and xgboost.
- Reasoning:-
  - scikit-learn simplifies ML workflows from preprocessing to evaluation.
  - xgboost enhances model performance with gradient boosting.
  - joblib ensures quick loading and deployment of pre-trained models.

13.2.4 Frontend Frameworks

- Requirement:- A responsive UI framework for the extension popup.
- Choice:- HTML, CSS, JavaScript and Bootstrap.

- Reasoning:-
  - HTML and JavaScript provide broad compatibility.
  - Bootstrap accelerates UI development and ensures consistency.

13.2.5 Integration Strategy

- Requirement:- Secure and seamless communication between extension and backend.
- Solution:-
  - CORS-enabled Flask APIs.
  - HTTPS protocol for data integrity and security.

14.0 Console Error Logging

This section provides information on identifying and resolving common errors encountered while developing, deploying or using the Aegis Shield extension.

1. Debugging in Browser Console
   - Open the Developer Tools in your browser (usually accessible via F12 or Ctrl + Shift + I).
   - Navigate to the Console tab to monitor error messages.
   - Use the Network tab to inspect API requests and responses.
2. Common Errors and Solutions

Error 1: “Failed to fetch”

- Cause: This error occurs when the extension cannot connect to the Flask API for phishing detection.
- Steps to Resolve:

  1.Ensure the Flask server is running locally (http://127.0.0.1:5000).

  2.Verify that the correct host permissions are listed in manifest.json under host_permissions.

  3.Check for CORS errors in the console and ensure the Flask app includes CORS middleware.

Error 2: "Rule ID exceeds maximum limit"

- Cause: The Chrome extension's dynamic rule limit (1,000 rules) is exceeded.
- Steps to Resolve:

  1.Remove unused rules from the declarativeNetRequest API.

  2.Optimize rule storage by combining overlapping conditions.

Error 3: "Network error: Request blocked by CORS policy"

- Cause: Cross-origin requests to the API are blocked.
- Steps to Resolve:

  1.Enable CORS in your Flask app using:

  "python

  CopyEdit

  from flask_cors import CORS

  CORS(app)"

  2.Verify that the extension's background script includes appropriate headers.

3. Whitelist Management
   - Errors encountered while adding URLs to the whitelist:

```
} catch (error) {
    console.error('Error adding to whitelist:', error);
    alert(error.message);
}
```

   - Purpose: Logs issues when URLs fail to be added to the whitelist.
4. Blacklist Management
   - Errors encountered while adding URLs to the blacklist:

```
} catch (error) {
    console.error('Error adding to blacklist:', error);
    alert(error.message);
}
```

- Purpose: Captures errors during blacklist updates.

5. Current Tab URL Retrieval
   - Errors during the process of fetching the current tab's URL:

```
catch (error) {
  console.error('Error getting current tab:', error);
```

   - Purpose: Logs retrieval failures to debug invalid or inaccessible tab URLs.
6. URL Scanning Errors
   - Errors encountered during URL scanning:

```
} catch (error) {
  console.error('Error scanning URL:', error);
  createNotification('Error', 'Failed to scan URL');
}
```

   - Purpose: Logs scanning-related errors.
7. API Key Fetching Errors
   - Errors encountered while fetching the VirusTotal API key:

```
    console.error('Failed to fetch API key: No key in response');
  }
})
.catch(error => {
  console.error('Error fetching API key:', error);
});
```

   - Purpose: Logs errors during the API key fetching process.
8. URL Detection and Analysis
   - Errors encountered during URL detection:

```
} catch (error) {
    resultElement.textContent = "Error detecting URL!";
    console.error("Error:", error);
}
```

   - Purpose: Logs issues during the detection process.
9. Machine Learning (ML) URL Analysis
   - Errors that occur while analyzing URLs using ML models:

```
} catch (error) {
    resultElement.textContent = `Error: ${error.message}`;
    resultElement.className = "error";
    console.error("Error analyzing URL:", error);
}
```

- Purpose: Captures issues within the ML URL analysis pipeline.

10. Scanning Progress Handling

- Errors that arise during the scanning process:

```
} catch (error) {
    // Handle and display any errors that occur during scanning
    resultDiv.textContent = `Error: ${error.message}`;
    resultDiv.className = 'error';
    console.error('Error:', error);
} finally {
    // Hide progress bar after scanning
    progressBar.style.display = 'none';
}
```

- Purpose: Ensures any issues during scanning are logged.

15.0 Security Considerations

- API keys are securely managed through the backend server
- All URL and email analyses are performed server-side
- Secure communication protocols between extension and backend
- Regular model updates for emerging threat patterns

16.0 Performance

- Real-time scanning with < 1 second response time
- Intelligent caching system for frequently accessed URLs
- Optimized ML models for rapid classification
- Minimal browser resource utilization

17.0 Future Enhancements

- Integration with additional security APIs
- Enhanced ML model retraining capabilities
- Extended support for email attachment analysis

- Advanced visualization for security metrics
- Mobile browser support

18.0 Project Team

- 4-member development team
- 10-week development cycle (Agile Methodology)
- Applied project implementation

19.0 Third-party libraries:

- **Flask**:- For creating the API backend (app.py)
- **Flask-CORS**:-To enable Cross-Origin Resource Sharing in the Flask app
- **Chrome Extension APIs**:-Includes declarativeNetRequest, notifications, storage, tabs and webNavigation APIs for managing Chrome extension features (background.js, manifest.json)
- **VirusTotal API**:-For integrating threat detection capabilities (background.js, manifest.json)
- **Font Awesome 6.4.0**:-For icons used in the popup UI (popup.html)
- **Bootstrap 5.3.2**:-For styling the popup and UI components (popup.html, popup.css, bootstrap.bundle.min.js)
- **tldextract**:-For extracting domain-related features from URLs
- **pandas**:-For data manipulation and feature extraction
- **joblib**:-For loading machine learning models
- **scikit-learn**:- Preprocessing, modeling and evaluation.
- **xgboost**:- Advanced gradient boosting classifier
- **validators:-** Ensure that the input data adheres to specific rules or formats

20.0 Acknowledgments

- VirusTotal API for security scanning capabilities.
- Pandas, Scikit-learn, Xgboost, Matplotlib, Seaborn, Joblibfor ML implementation.
- Flask for backend framework.
- Chrome extension development communities.
- Project mentors, peers and team members for their invaluable feedback, collaboration.

- Contributors of the Phishing Email Dataset and the Phishing URL Dataset for providing high-quality, real-world datasets.

21.0 Useful Links

1. Datasets:

   Kaitholikkal, J. K. S., & B, A. (2024). Phishing URL dataset. *Mendeley Data*.

   https://doi.org/10.17632/vfszbj9b36.1

   Phishing email Detection. (2023, July 6). Kaggle.

   https://www.kaggle.com/datasets/subhajournal/phishingemails

2. APIs:

   VirusTotal Intelligence Introduction. (n.d.). VirusTotal.

   https://docs.virustotal.com/docs/virustotal-intelligence-introduction

3. Libraries and Frameworks:

   Flask-CORS — Flask-Cors 3.0.10 documentation. (n.d.).

   https://flask-cors.readthedocs.io/en/latest/

   scikit-learn: machine learning in Python — scikit-learn 0.16.1 documentation. (n.d.).

   https://scikit-learn.org/

   Welcome to Flask — Flask Documentation (3.1.x). (n.d.).

   https://flask.palletsprojects.com/

   XGBoost Documentation — xgboost 2.1.3 documentation. (n.d.).

   https://xgboost.readthedocs.io/

4. Frontend:

   Font awesome docs. (n.d.). Font Awesome Docs.

   https://fontawesome.com/docs

   Https://getbootstrap.com/docs/5.3/getting-started/introduction/. (n.d.).

   https://getbootstrap.com/docs/5.3/

5. Visualization:

Matplotlib — Visualization with Python. (n.d.).

https://matplotlib.org/

seaborn: statistical data visualization — seaborn 0.13.2 documentation. (n.d.).

https://seaborn.pydata.org/

WordCloud for Python documentation — wordcloud 1.8.1 documentation. (n.d.).

https://amueller.github.io/word_cloud/index.html

6. Other Tools:

Joblib: running Python functions as pipeline jobs — joblib 1.4.2 documentation. (n.d.).

https://joblib.readthedocs.io/

tldextract. (2024, November 5). PyPI.

https://pypi.org/project/tldextract/

JCharisTech. (2018, March 24). *Machine learning web app with Flask+Bootstrap (Gender Classifier App)* [Video]. YouTube.

https://www.youtube.com/watch?v=JteK3xvpg3Q

Stats Wire. (2021, April 10). *Deploy Machine Learning Model Flask* [Video]. YouTube.

https://www.youtube.com/watch?v=MxJnR1DMmsY

validators. (2024, September 3). PyPI. https://pypi.org/project/validators/