# Malicious URL Classification Using Supervised Machine Learning Models

## Introduction

Our group wishes to utilize the Malicious URL Dataset on Kaggle to train a machine-learning model to identify and classify potentially malicious links. The dataset has approximately 650,000 URLs, including ~430,000 benign URLs, 96,000 defacement URLs, ~94,000 phishing URLs, and ~33,000 malware URLs [1].

We observe related work and inspiration from the Stanford Machine Learning Final Projects [7]. The project aims to classify URLs as benign or malicious, utilizing 1,000 website URLs scraped from web as well as known malicious URLs from a malware database. We choose an alternative approach to solve the same problem. Instead of a binary classification, we wish to utilize four categories, as described in the utilized dataset, and intend to use more datasets - approximately three orders of magnitude greater.

## Problem Definition

Over the past few years, Georgia Tech has seen a surge in malicious emails targeting the student body. These emails often masquerade as being sent from a trusted source such as the IT department or an administrative office on campus. These phishing attempts commonly include urgent notifications about a student's account, such as a full Outlook inbox, mandatory password resets, or waitlist status updates. These emails aim to create a sense of urgency for the students so they act quickly and click the embedded link.

Once the link is clicked, students are prompted to log in through the Georgia Tech SSO portal. The process appears legitimate, so students do not hesitate to enter their authentication credentials to resolve the problem. Instead of redirecting the student to the service responsible for the emails, students are typically redirected to OSCAR or a different landing page, which has no connection to the problem the student is experiencing. Student data is captured by this fraudulent site in its backend operations, thereby compromising the security of the account of the student in real-time.

The consequences of a security breach involving Institute log-in information are serious. Compromised accounts could grant unauthorized access to sensitive academic records, personal information, financial details, or highly regulated services such as the AI-Makerspace. Furthermore, bad actors could use these accounts to execute further phishing campaigns, malware distribution, or identity theft. We recognize the need for effective mechanisms of detection and prevention against these types of attacks to protect the Georgia Tech community.

We wish to meet this growing trend of cybersecurity threats with a supervised machine learning model that will be capable of detecting embedded links within emails and other forms of digital communication that is potentially malicious.

## Methods

### Dataset

We utilized the **Malicious URLs Dataset** from Kaggle, which contains a variety of URLs labeled as benign, phishing, malware, or defacement. The dataset provides numerous examples for both malicious and benign URLs, making it suitable for training and evaluating our models.

### Data Preprocessing

#### Feature Extraction

To convert URLs into a format suitable for machine learning algorithms, we extracted several enhanced features that could potentially indicate malicious intent [6]:

- **Length-Based Features:**
  - `url_length` : Total length of the URL.
  - `domain_length` : Length of the domain part of the URL.
  - `path_length` : Length of the path after the domain.
- **Character-Based Features:**
  - `num_digits` : Number of digits in the URL.
  - `num_special` : Number of special characters (non-alphanumeric).
- **Domain-Based Features:**
  - `num_dots` : Number of dots in the URL.
  - `num_hyphens` : Number of hyphens.
  - `num_underscores` : Number of underscores.
  - `num_subdomains` : Number of subdomains.
- **Path-Based Features:**
  - `num_slashes` : Number of slashes.
  - `num_params` : Number of URL parameters.

    - `has_query` : Presence of a query string (1 if present, 0 otherwise).
- **Protocol-Based Features:**
    - `uses_https` : Indicates if the URL uses HTTPS protocol.
- **IP Address Presence:**
    - `has_ip_address` : Indicates if the URL contains an IP address.
- **Suspicious Words:**
    - `suspicious_words` : Presence of words like 'login', 'secure', 'account', etc.
- **Entropy of URL:**
    - `url_entropy` : Measures the randomness in the URL string.

```
 # Feature extraction function
def extract_url_features(url):
    """Extract enhanced features from URLs."""
    features = {}

    # Parse URL
    parsed = urlparse(url)

    # Length-based features
    features['url_length'] = len(url)
    features['domain_length'] = len(parsed.netloc)
    features['path_length'] = len(parsed.path)

    # Character-based features
    features['num_digits'] = sum(c.isdigit() for c in url)
    features['num_special'] = len(re.findall('[^A-Za-z0-9]', url))

    # Domain-based features
    features['num_dots'] = url.count('.')
    features['num_hyphens'] = url.count('-')
    features['num_underscores'] = url.count('_')
    features['num_subdomains'] = len(parsed.netloc.split('.')) - 2  # Adjust based on TLDs

    # Path-based features
    features['num_slashes'] = url.count('/')
    features['num_params'] = len(parsed.params)
    features['has_query'] = int(len(parsed.query) > 0)

    # Protocol-based features
    features['uses_https'] = int(parsed.scheme == 'https')

    # IP address presence
    features['has_ip_address'] = int(bool(re.match(r'\d+\.\d+\.\d+\.\d+', parsed.netloc)))
    # Suspicious words
    suspicious_words = ['login', 'secure', 'account', 'update', 'free', 'gift', 'verification']
    features['suspicious_words'] = int(any(word in url.lower() for word in suspicious_words))

    # Entropy of URL
    import math
    def calculate_entropy(s):
        prob = [float(s.count(c)) / len(s) for c in dict.fromkeys(list(s))]
        entropy = - sum([p * math.log(p) / math.log(2.0) for p in prob])
        return entropy
    features['url_entropy'] = calculate_entropy(url)

    return features
```

These features were extracted for all URLs in the dataset and then converted into a DataFrame for further processing.

Initially, the model was built using only the first four feature categories. To enhance accuracy, the final four features were subsequently added. These additional features improved model accuracy by an average of 3% and demonstrated unexpectedly high importance compared to the original features. Detailed results and analysis of these features will be presented in later sections.

### Encoding Labels

We converted the categorical labels ( `'benign'` , `'phishing'` , `'malware'` , `'defacement'` ) into numerical format using label encoding, which is necessary for training the machine learning models.

```
 # Encode labels
le = LabelEncoder()
y = le.fit_transform(df['type'])
```

### Data Splitting and Standardization

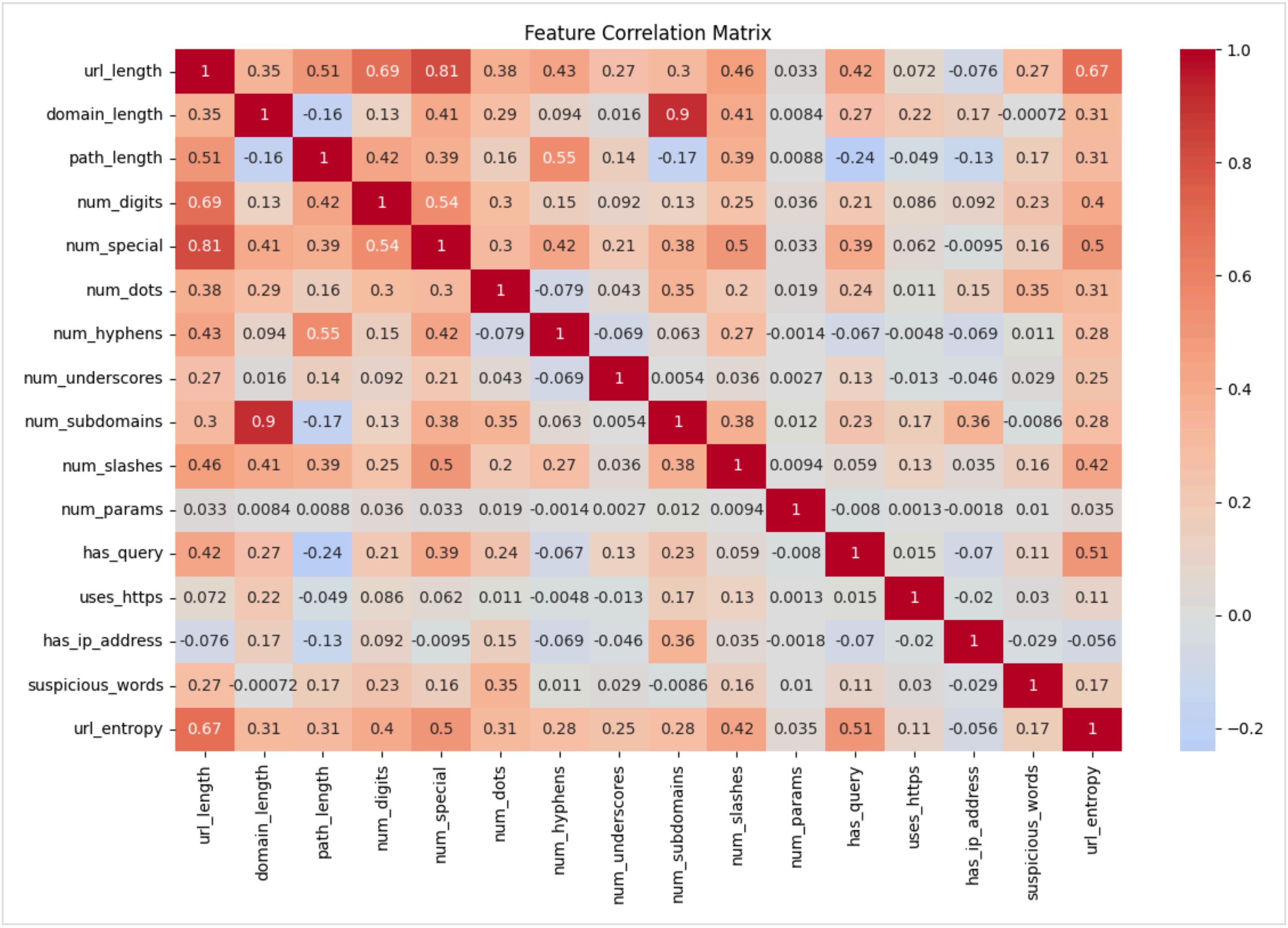
The dataset was then split into training and testing sets using an 80-20 split to evaluate the models' performance on unseen data [8]. The features were then subsequently standardized to have zero mean and unit variance, which is essential for algorithms like Logistic Regression and SVM.

```
 # Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

## Exploratory Data Analysis

To understand the relationships between features and identify any potential issues, such as multicollinearity, we utilized a correlation heatmap and feature distribution dot plot to better understand the dataset [9].



| | url_length | domain_length | path_length | num_digits | num_special | num_dots | num_hyphens | num_underscores | num_subdomains | num_slashes | num_params | has_query | uses_https | has_ip_address | suspicious_words | url_entropy |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| url_length | 1 | 0.35 | 0.51 | 0.69 | 0.81 | 0.38 | 0.43 | 0.27 | 0.3 | 0.46 | 0.033 | 0.42 | 0.072 | -0.076 | 0.27 | 0.67 |
| domain_length | 0.35 | 1 | -0.16 | 0.13 | 0.41 | 0.29 | 0.094 | 0.016 | 0.9 | 0.41 | 0.0084 | 0.27 | 0.22 | 0.17 | -0.00072 | 0.31 |
| path_length | 0.51 | -0.16 | 1 | 0.42 | 0.39 | 0.16 | 0.55 | 0.14 | -0.17 | 0.39 | 0.0088 | -0.24 | -0.049 | -0.13 | 0.17 | 0.31 |
| num_digits | 0.69 | 0.13 | 0.42 | 1 | 0.54 | 0.3 | 0.15 | 0.092 | 0.13 | 0.25 | 0.036 | 0.21 | 0.086 | 0.092 | 0.23 | 0.4 |
| num_special | 0.81 | 0.41 | 0.39 | 0.54 | 1 | 0.3 | 0.42 | 0.21 | 0.38 | 0.5 | 0.033 | 0.39 | 0.062 | -0.0095 | 0.16 | 0.5 |
| num_dots | 0.38 | 0.29 | 0.16 | 0.3 | 0.3 | 1 | -0.079 | 0.043 | 0.35 | 0.2 | 0.019 | 0.24 | 0.011 | 0.15 | 0.35 | 0.31 |
| num_hyphens | 0.43 | 0.094 | 0.55 | 0.15 | 0.42 | -0.079 | 1 | -0.069 | 0.063 | 0.27 | -0.0014 | -0.067 | -0.0048 | -0.069 | 0.011 | 0.28 |
| num_underscores | 0.27 | 0.016 | 0.14 | 0.092 | 0.21 | 0.043 | -0.069 | 1 | 0.0054 | 0.036 | 0.0027 | 0.13 | -0.013 | -0.046 | 0.029 | 0.25 |
| num_subdomains | 0.3 | 0.9 | -0.17 | 0.13 | 0.38 | 0.35 | 0.063 | 0.0054 | 1 | 0.38 | 0.012 | 0.23 | 0.17 | 0.36 | -0.0086 | 0.28 |
| num_slashes | 0.46 | 0.41 | 0.39 | 0.25 | 0.5 | 0.2 | 0.27 | 0.036 | 0.38 | 1 | 0.0094 | 0.059 | 0.13 | 0.035 | 0.16 | 0.42 |
| num_params | 0.033 | 0.0084 | 0.0088 | 0.036 | 0.033 | 0.019 | -0.0014 | 0.0027 | 0.012 | 0.0094 | 1 | -0.008 | 0.0013 | -0.0018 | 0.01 | 0.035 |
| has_query | 0.42 | 0.27 | -0.24 | 0.21 | 0.39 | 0.24 | -0.067 | 0.13 | 0.23 | 0.059 | -0.008 | 1 | 0.015 | -0.07 | 0.11 | 0.51 |
| uses_https | 0.072 | 0.22 | -0.049 | 0.086 | 0.062 | 0.011 | -0.0048 | -0.013 | 0.17 | 0.13 | 0.0013 | 0.015 | 1 | -0.02 | 0.03 | 0.11 |
| has_ip_address | -0.076 | 0.17 | -0.13 | 0.092 | -0.0095 | 0.15 | -0.069 | -0.046 | 0.36 | 0.035 | -0.0018 | -0.07 | -0.02 | 1 | -0.029 | -0.056 |
| suspicious_words | 0.27 | -0.00072 | 0.17 | 0.23 | 0.16 | 0.35 | 0.011 | 0.029 | -0.0086 | 0.16 | 0.01 | 0.11 | 0.03 | -0.029 | 1 | 0.17 |
| url_entropy | 0.67 | 0.31 | 0.31 | 0.4 | 0.5 | 0.31 | 0.28 | 0.25 | 0.28 | 0.42 | 0.035 | 0.51 | 0.11 | -0.056 | 0.17 | 1 |

*Figure 1: Heatmap showing the correlation between extracted features.*

```
 # Create correlation heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(X.corr(), annot=True, cmap='coolwarm', center=0)
plt.title('Feature Correlation Matrix')
plt.tight_layout()
plt.show()
```

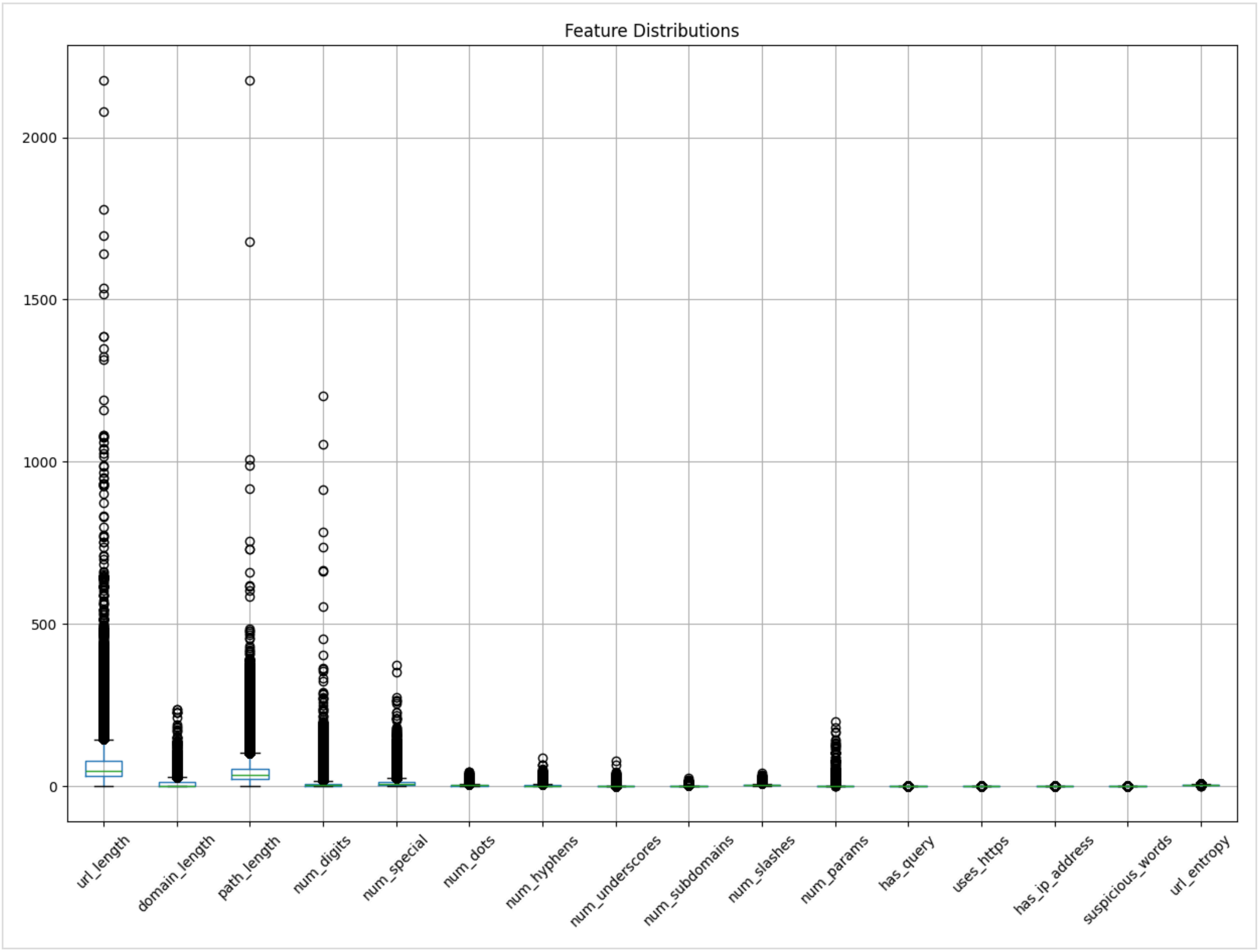


*Figure 2: Box plot showing the distribution of extracted features.*

```
 # Display feature statistics
print("\nFeature statistics:")
display(X.describe())

# Visualize feature distributions
```

```
plt.figure(figsize=(15, 10))
X.boxplot()
plt.xticks(rotation=45)
plt.title('Feature Distributions')
plt.show()
```

## Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis is a supervised learning algorithm used for classification and dimensionality reduction. It projects the data onto a lower-dimensional space while maximizing the separation between different classes [2].

LDA classifies the URLs by utilizing the most discriminative features. These qualities make LDA particularly interesting since we utilize 16 features for models. With LDA, we can observe what linear combination of features results in the best separation of classes. Within the context of this project, this will allow us to see the separation between benign and various types of malicious URLs.

### Model Training and Evaluation

```
 from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Train LDA model
lda = LinearDiscriminantAnalysis()
lda.fit(X_train, y_train)

# Evaluate the model
y_pred = lda.predict(X_test)
```

After training the model, we then computed the accuracy, generated a classification report, and plotted the confusion matrix.

```
 # Compute accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of LDA model: {:.2f}%".format(accuracy * 100))
```

```
# Print classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=le.classes_))
```

## Output:

```
Accuracy of LDA model: 81.05%

Classification Report:
              precision    recall  f1-score   support

      benign       0.83      0.97      0.90     85778
  defacement       0.78      0.91      0.84     19104
     malware       0.66      0.57      0.61      6521
    phishing       0.44      0.05      0.10     18836

    accuracy                           0.81    130239
   macro avg       0.68      0.63      0.61    130239
weighted avg       0.76      0.81      0.76    130239
```

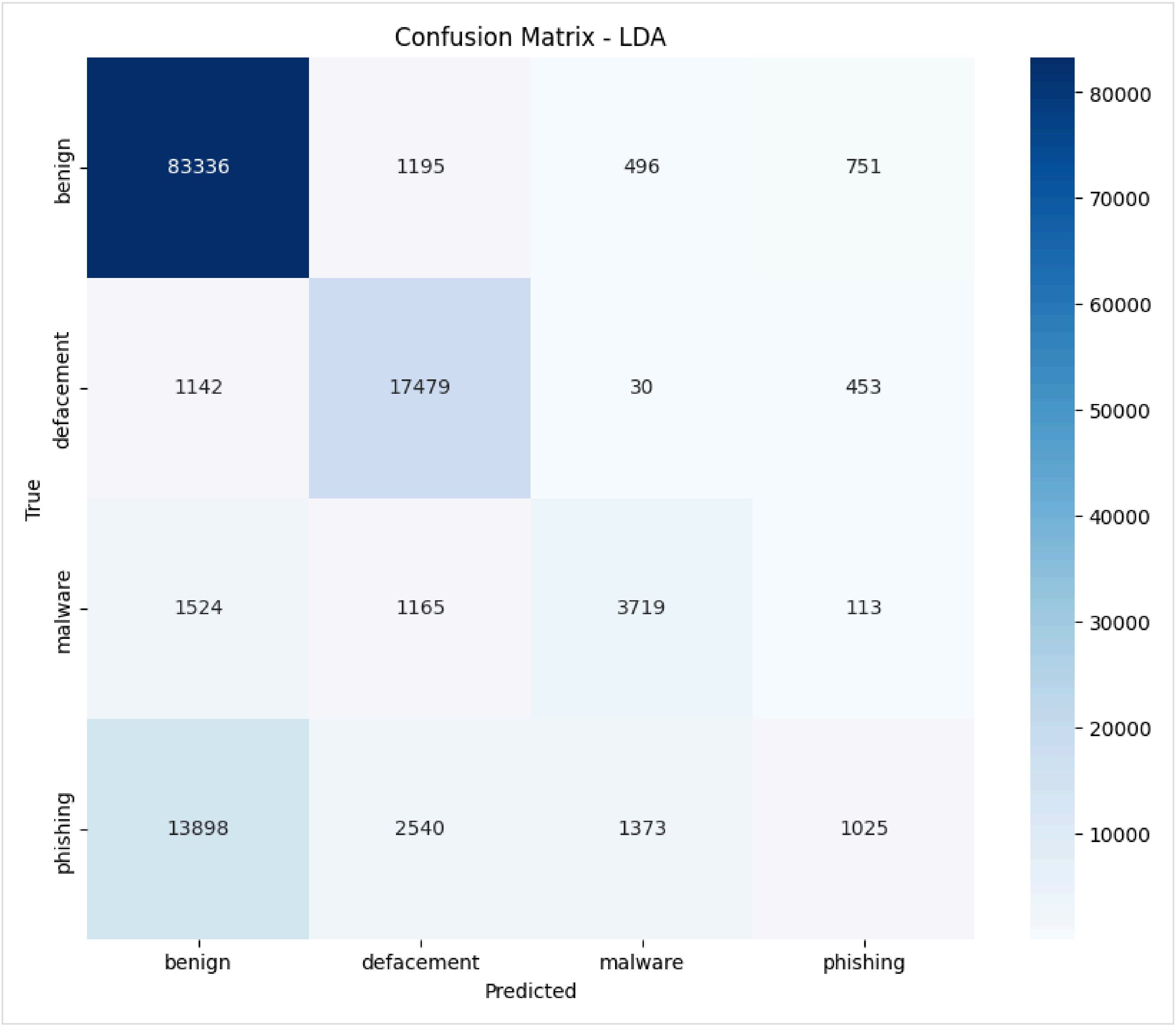


*Figure 3: Confusion matrix displaying the LDA model's performance across different classes.*

```
 # Create confusion matrix
plt.figure(figsize=(10, 8))
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=le.classes_,
            yticklabels=le.classes_)
plt.title('Confusion Matrix - LDA')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

### Feature Importance

Analyzing the coefficients from the LDA model provides insights into which features are most influential in distinguishing between classes.

Given LDA's dimensionality reduction, it is not surprising to observe the exponential nature of the feature importance. The top three features listed in the Feature Importance chart have some of the lowest feature correlations as found in Figure 1. With the subsequent models' Feature Importance graph, we do not observe a similar exponential trend.

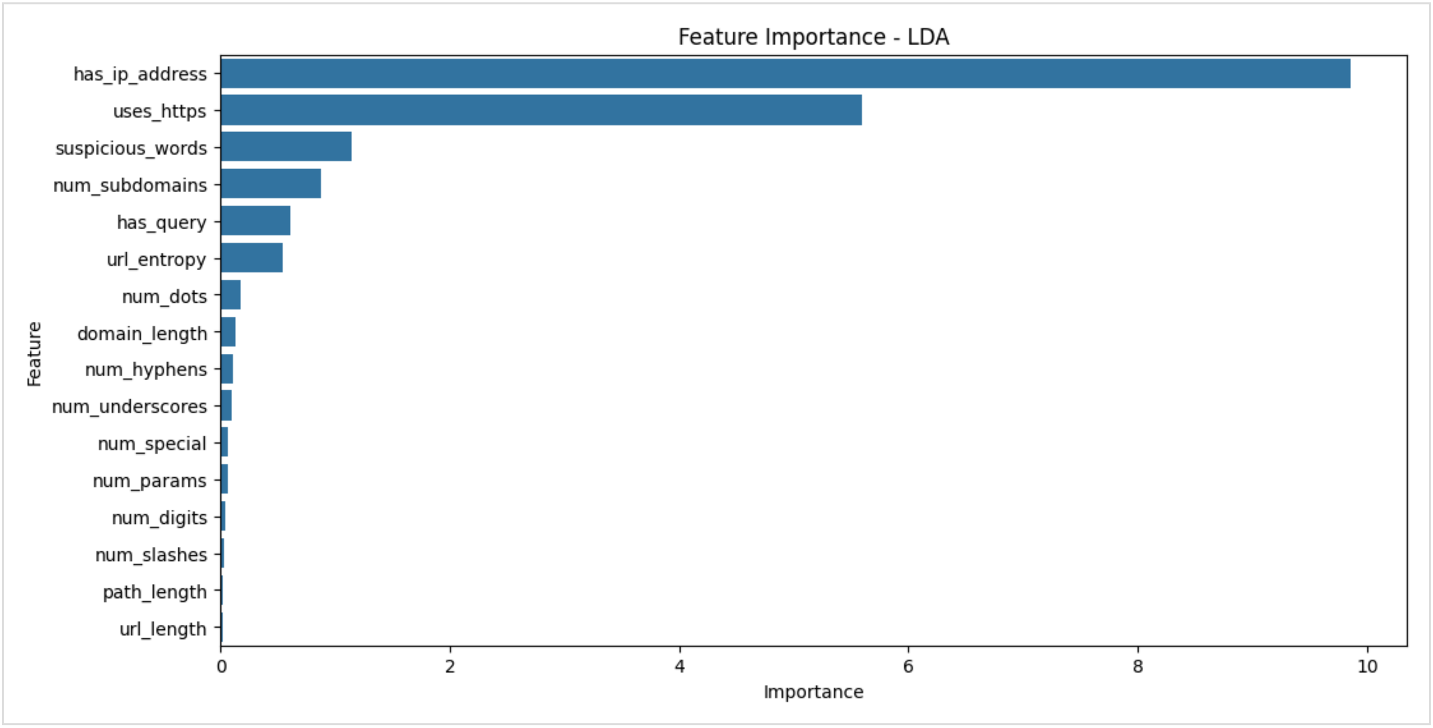


*Figure 4: Bar chart showing the importance of each feature based on the LDA coefficients.*

```
 # Feature importance analysis
feature_importance = pd.DataFrame({
    'Feature': X.columns,
    'Importance': np.abs(lda.coef_).mean(axis=0)
}).sort_values('Importance', ascending=False)

plt.figure(figsize=(12, 6))
sns.barplot(data=feature_importance, x='Importance', y='Feature')
plt.title('Feature Importance - LDA')
plt.show()
```

### LDA Transformation Visualization

As seen below, we are to visualize the data after LDA transformation to understand how well the classes are separated. It is easily seen there is not a clear divide between data classes. Most of the data is centered around (0,0); this makes an accurate model (>85% accuracy) difficult if the model is not complex. *This is a foreshadowing of our Random Forest model implementation.*

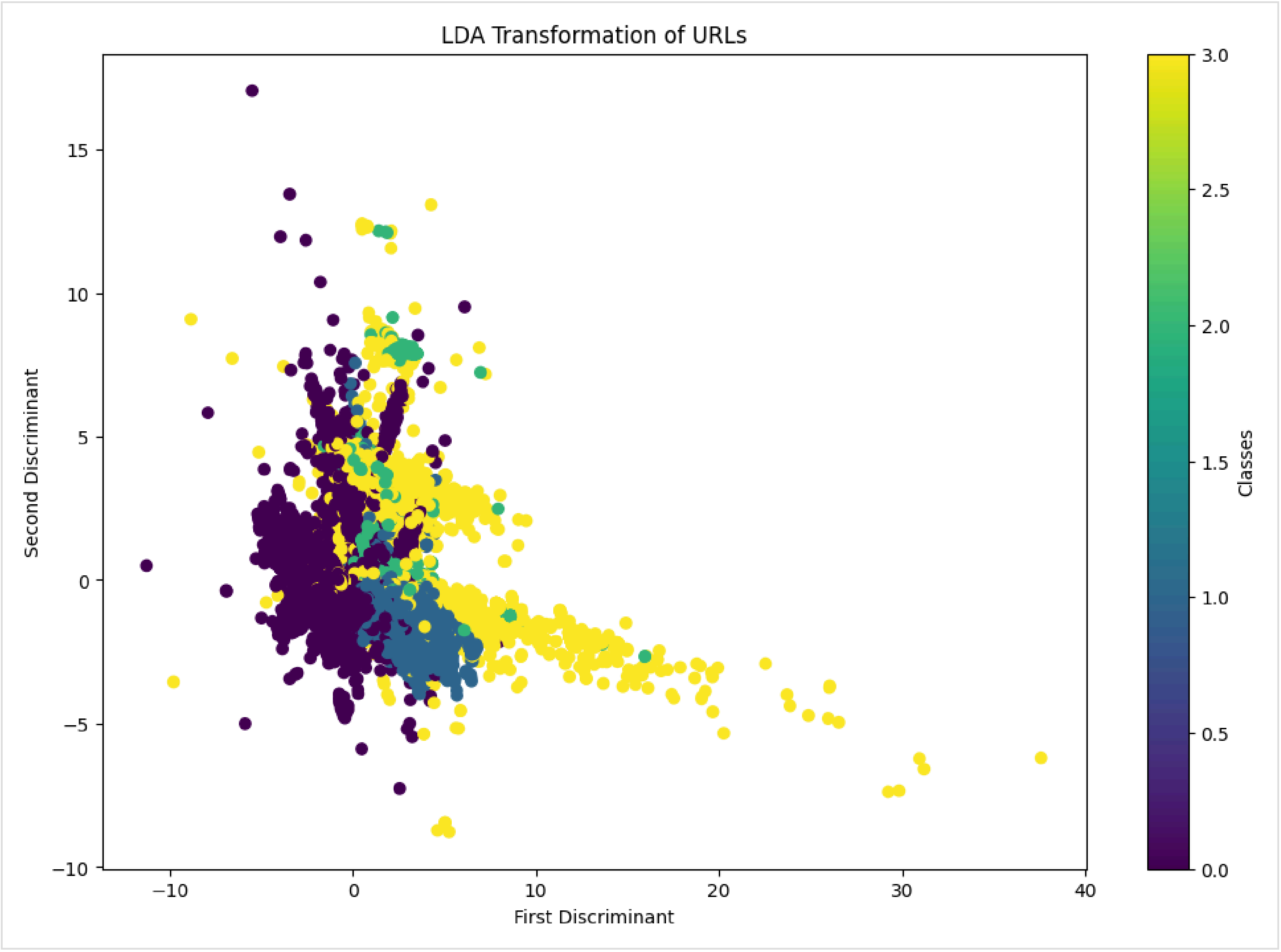


*Figure 5: Scatter plot of the URLs after LDA transformation.*

```
 # Visualize LDA transformation
X_lda = lda.transform(X_train)
if X_lda.shape[1] >= 2:
    plt.figure(figsize=(12, 8))
    scatter = plt.scatter(X_lda[:, 0], X_lda[:, 1], c=y_train, cmap='viridis')
    plt.title('LDA Transformation of URLs')
    plt.xlabel('First Discriminant')
    plt.ylabel('Second Discriminant')
    plt.colorbar(scatter, label='Classes')
    plt.show()
```

### LDA Analysis

Webpage URLs are not intuitive to classify by humans and even less intuitive to classify by machine learning algorithms with a low number of features. It is evident through previous visualizations LDA chooses the most important features through how uncorrelated they are. However, URLs are, by design, not complicated. Therefore, it is not surprising that the classes in Figure 5 do not have clear divisions, and most of the points lie around (0,0).

The model struggled significantly with the classification of phishing links (precision: 0.44, recall: 0.05, f1-score: 0.10). This difficulty is likely due to LDA having difficulty distinguishing between classes, which is not surprising given the results found in Figure 5.

The performance of the algorithm can likely be attributed toward the assuptions LDA makes; the model assumes the features are normally distributed within the class and that all classes share the same covariance matrix. By observing Figure 1 and Figure 2, we find this is not the case. Additionally, LDA is inherently sensitive to class imbalance - predicting the majority class and leading to underperformance on underrepresented classes (e.x. malware and phishing). The results of LDA support this claim and to better abide by these assuptions, we wish to utilize SMOTE or perform more involved data preprocessing to better capture maximum variance, such as PCA.

## Logistic Regression

Logistic Regression is a model that uses a logistic function to model a binary dependent variable. This classification task uses a one-vs-rest scheme to handle multiple classes.

### Model Training and Evaluation

```
 from sklearn.linear_model import LogisticRegression

# Train and evaluate logistic regression
```

```
model = LogisticRegression()
model.fit(X_train_scaled, y_train)

# Evaluate the model
y_pred = model.predict(X_test_scaled)
```

We computed the accuracy, generated a classification report, and plotted the confusion matrix. These metrics, paired with the Feature Importance Chart, represent the consistencies that we use to compare all models to obtain a baseline and better compare the models.

```
 # Compute accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of Logistic Regression model: {:.2f}%".format(accuracy * 100))

# Print classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=le.classes_))
```

**Output:**

```
Accuracy of Logistic Regression model: 83.24%

Classification Report:
              precision    recall  f1-score   support

      benign       0.85      0.97      0.91     85778
  defacement       0.82      0.93      0.87     19104
     malware       0.83      0.67      0.74      6521
    phishing       0.52      0.15      0.23     18836

    accuracy                           0.83    130239
   macro avg       0.76      0.68      0.69    130239
weighted avg       0.80      0.83      0.80    130239
```

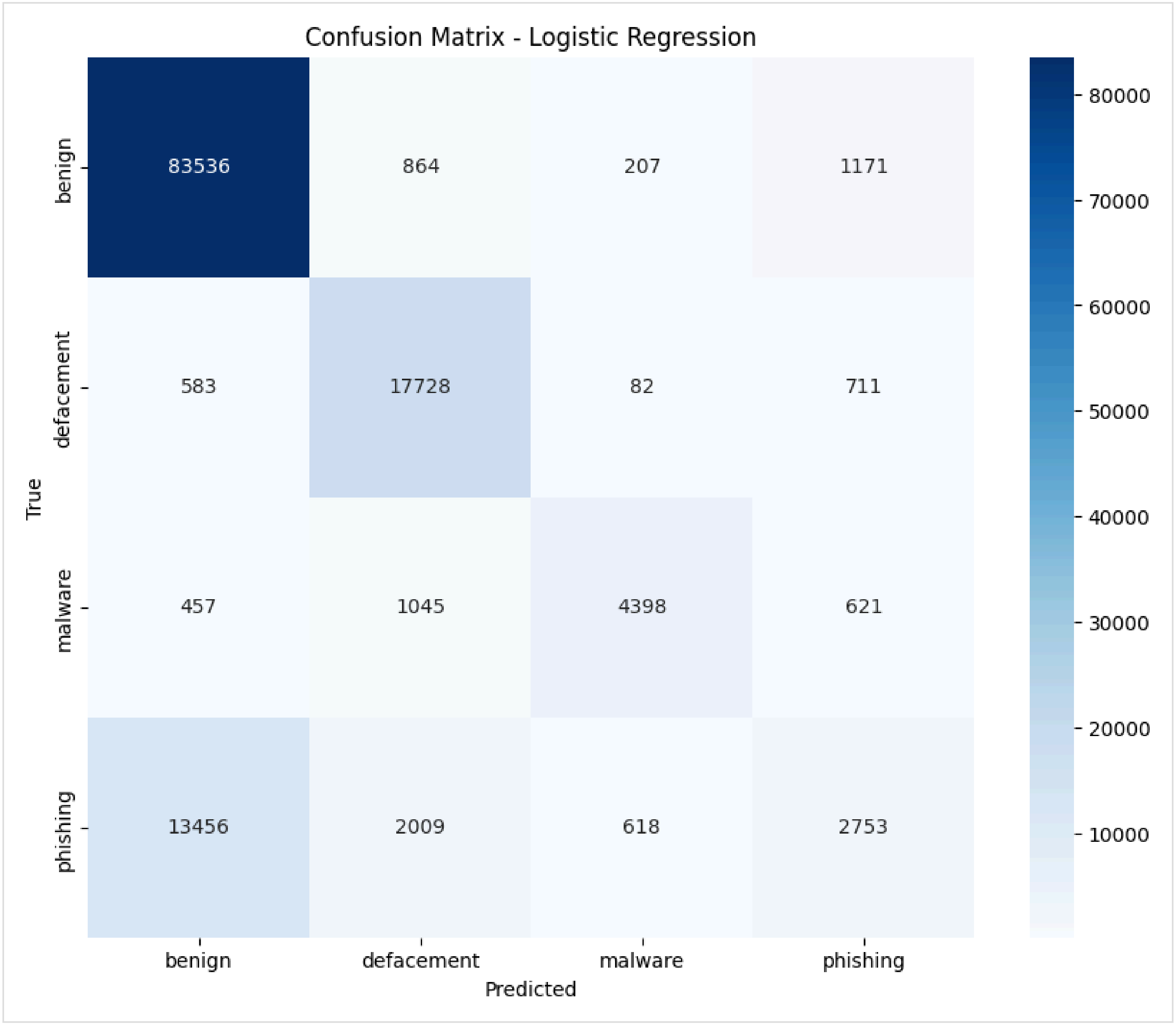


*Figure 6: Confusion matrix displaying the Logistic Regression model's performance across different classes.*

**Feature Importance**

Analyzing the coefficients from the Logistic Regression model helps identify the most significant features influencing the predictions. Compared to LDA, we observe the Feature Importance is more linear, not representing an exponential trend. This linear quality is akin to what will be present in subsequent models' Feature Importance charts.

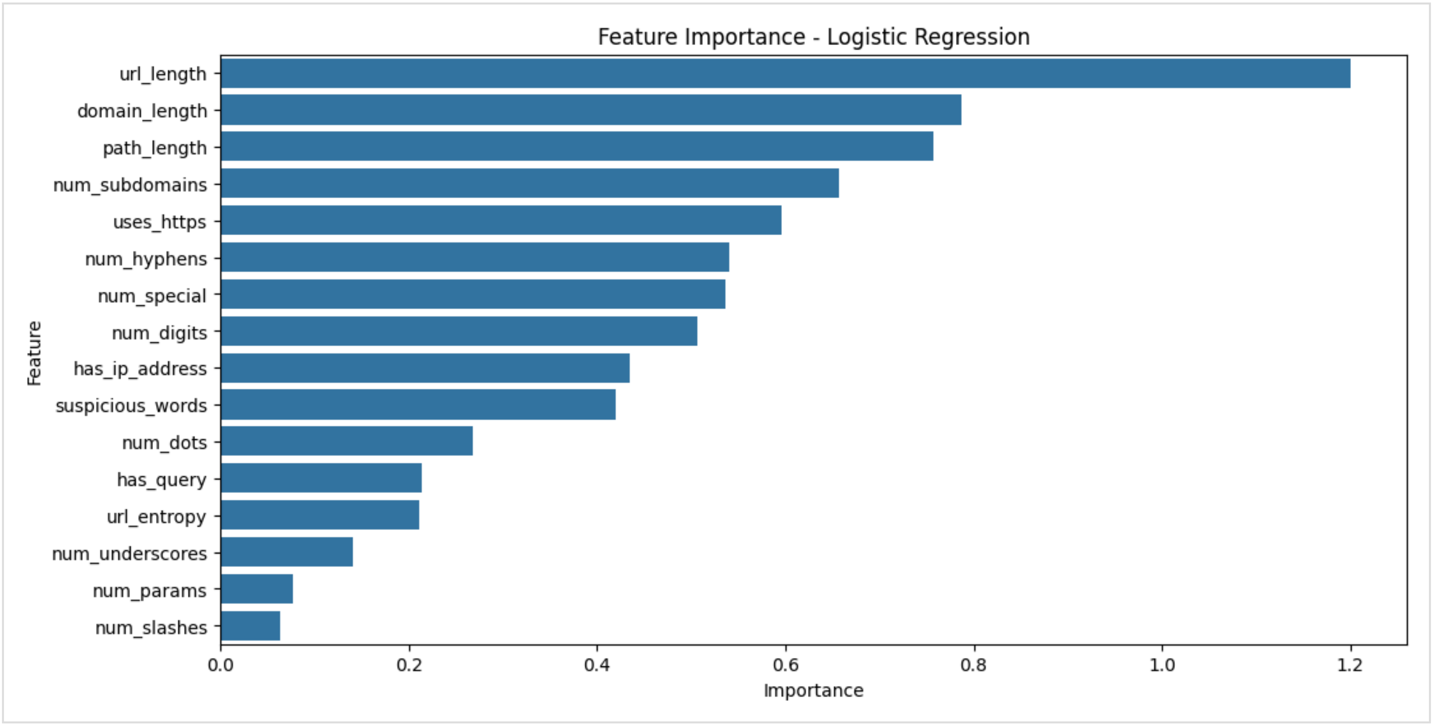


*Figure 7: Bar chart showing the importance of each feature based on the Logistic Regression coefficients.*

We observe Logistic Regression is a relatively simple model that is useful for prioritizing certain classifications based on risk levels. For example, if we wanted the model to be overly cautious and ensure it captures all malicious links - with the downside of having false positives for benign links - we would be able to do so. Logistic Regression is particularly good for large datasets, however, its simplicity does not best result in a great performance with complex datasets. The simplicity of the model and its ability to make multi-class classifications establish it as an effective baseline model. Results of this model provide a promising insight into what will be observed in future models.

A caveat of Logistic Regression's simplicity, however, is its assumption of linearity. We observe the input features and the log-odds of the target class are non-linear. Therefore, it is not surprising the model is unable to effectively distinguish between minority or otherwise complex classes (e.x. phishing and malware) [4]. This is likely due to overlapping feature distributions or insufficient discriminations between features.

Furthermore, we observe the model is highly sensitive to highly correlated features, which may impact the model's stability. The feature with the highest importance, according to Figure 7, was url_length. This feature, according to Figure 1, has consistently high correlations across features compared to other features. This made the model biased towards the url_length feature, which likely distorted the feature coefficients.

To resolve these issues, we find applications utilizing SMOTE or L1/L2 regularization to handle multicollinearity among features.

## Linear Support Vector Classifier (LinearSVC)

LinearSVC is a linear classifier using Support Vector Machines, suitable for classification tasks with large datasets. Initially, we wished to implement SVC. However, time and resource constraints established consistently running the algorithm as expensive. Instead, we switched to LinearSVC which, compared to SVC, uses a linear kernel, making it faster and scalable for high-dimensional data.

### Model Training and Evaluation

```
from sklearn.svm import LinearSVC

# Train LinearSVC model
svm = LinearSVC(penalty='l2', loss='squared_hinge', dual=False,
                C=1.0, tol=1e-4, max_iter=1000, random_state=42)
svm.fit(X_train_scaled, y_train)

# Evaluate the model
y_pred = svm.predict(X_test_scaled)
```

Similar to previous models, we computed the accuracy, generated a classification report, and plotted the confusion matrix. An interesting note is with LinearSVC and the previous two models, we have observed significantly low metrics for phishing compared to the other classes. We observe this is likely due to feature overlap and added complexity in which the models are unable to effectively compute.

```
# Compute accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of LinearSVC model: {:.2f}%".format(accuracy * 100))

# Print classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=le.classes_))
```

**Output:**

```
Accuracy of LinearSVC model: 82.65%

Classification Report:
              precision    recall  f1-score   support

      benign       0.84      0.98      0.90     85778
 defacement       0.80      0.95      0.87     19104
     malware       0.85      0.54      0.66      6521
    phishing       0.62      0.08      0.14     18836

    accuracy                           0.83    130239
   macro avg       0.78      0.64      0.64    130239
weighted avg       0.80      0.83      0.78    130239
```

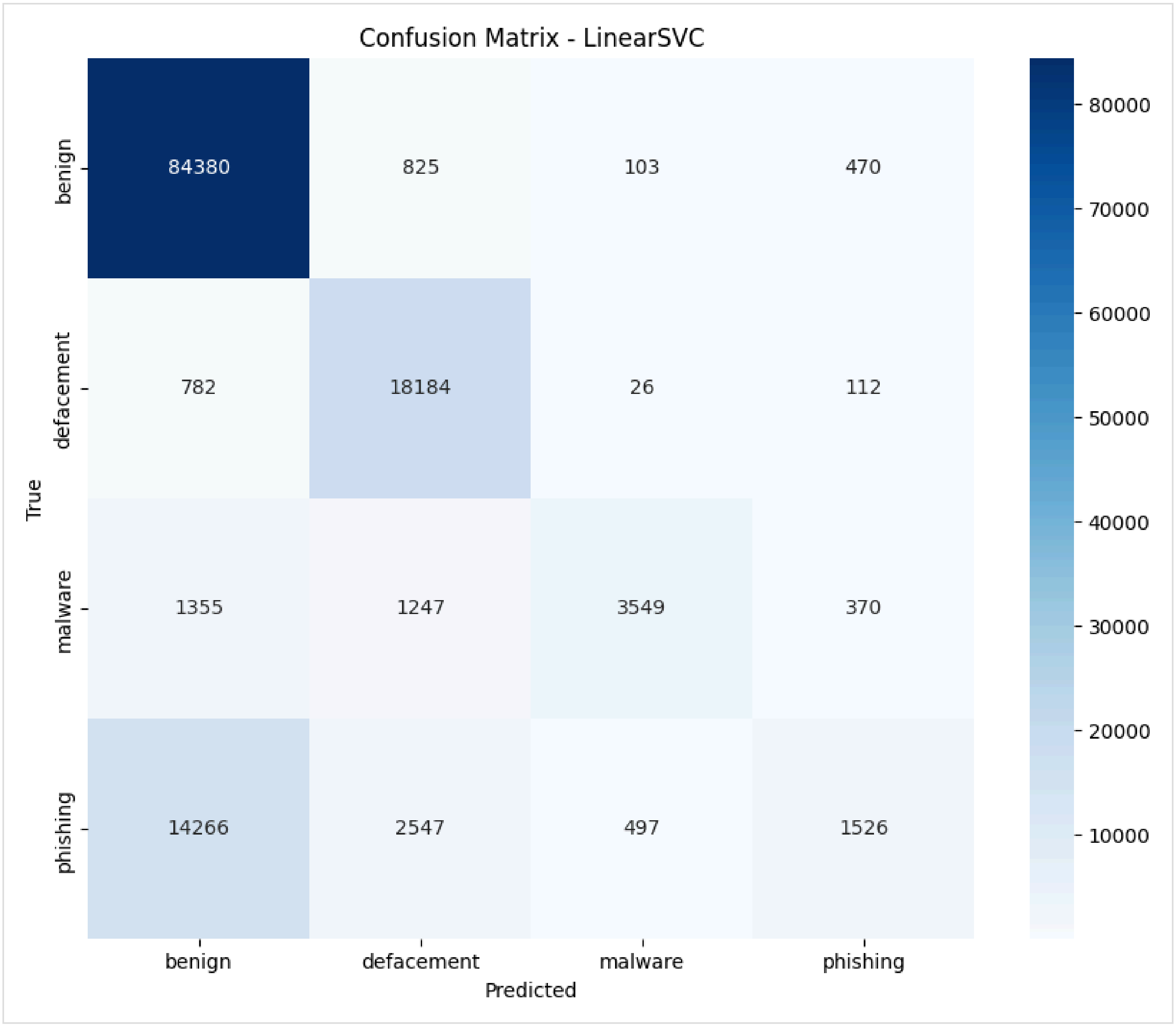


*Figure 8: Confusion matrix displaying the LinearSVC model's performance across different classes.*

### Feature Importance

The coefficients from the LinearSVC model indicate the importance of each feature in the classification.

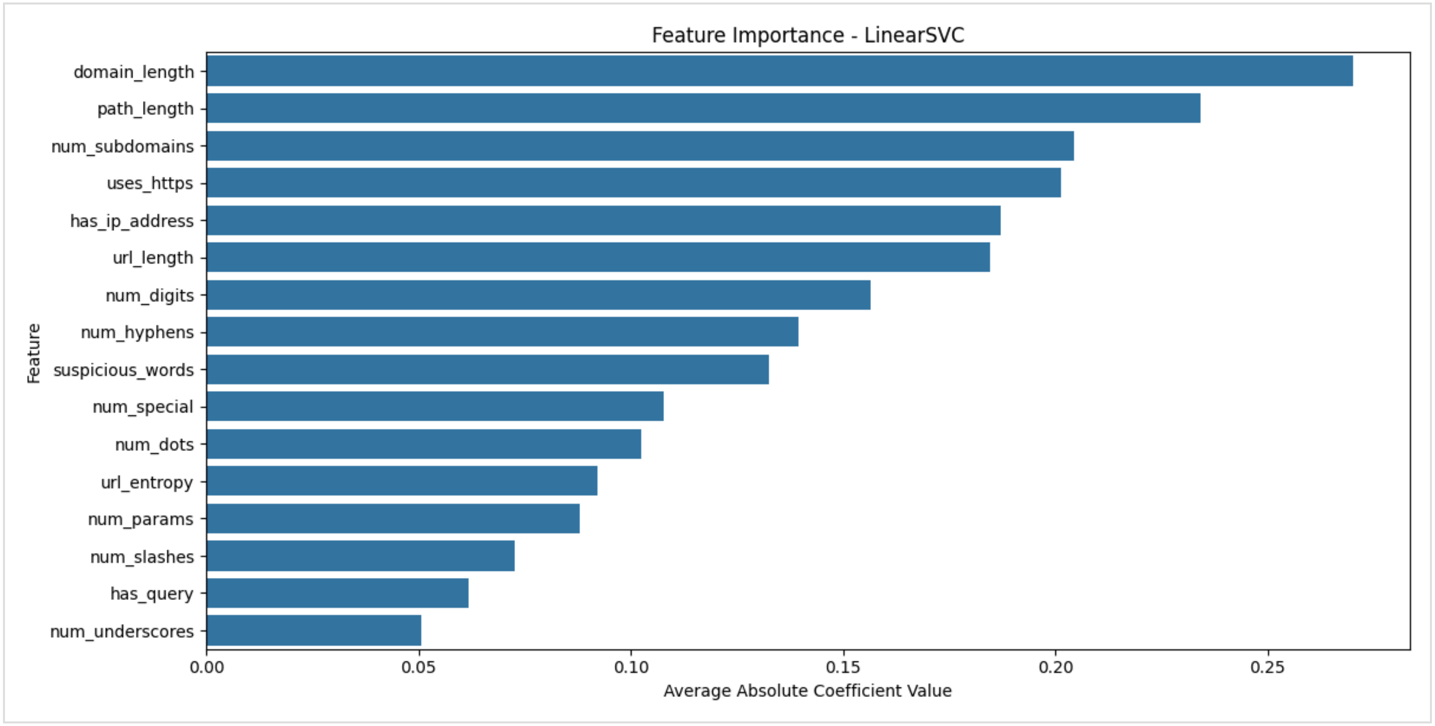


*Figure 9: Bar chart showing the importance of each feature based on the LinearSVC coefficients.*

We observe LinearSVC wishes to maximize the margin between different classes, which should make the model more able to generalize unseen data [3]. However, the complexity of the dataset may establish the primary benefit of LinearSVM as unhelpful. We observe a primary assumption of LinearSVC is that classes can be separated by a linear decision line. We find this assumption does not hold true and therefore does not provide a high accuracy score. Nevertheless, we recognize LinearSVC's efficiency, and could be a better model to utilize if time and space are a concern.

A trend with LinearSVC and the previous two models is their limited performance on minority and complex cases. With a large, complex dataset, we are able to directly see the ineffectiveness of utilizing models that struggle with underperforming and complex classes on such a dataset. We continue to observe bias towards benign URLs as a result of their predominance within the dataset.

To mitigate these issues, we attempted to apply feature scaling, however, we still are able to see in Figure 8 the effects of class imbalance paired with ineffective feature scaling.

We believe we would be able to improve the model by using a resampling technique such as SMOTE or by exploring non-linear options, despite them being more expensive.

## Random Forest Classifier

Random Forest is an ensemble learning method that operates by constructing multiple decision trees during training and outputting the class which is the mode of the classes of the individual trees [5]. The model is highlighted by its use of bootstrap aggregating to create diverse trees by training each tree on a random subset of the data. Additionally, we observe as a result of feature randomness, Random Forest can effectively handle high dimensionality without overfitting.

### Model Training and Evaluation

```
 from sklearn.ensemble import RandomForestClassifier

# Define the model with default parameters
rf = RandomForestClassifier(random_state=42)
rf.fit(X_train_scaled, y_train)

# Evaluate the model
y_pred = rf.predict(X_test_scaled)

# Compute accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of Random Forest model: {:.2f}%".format(accuracy * 100))
```

Finally, we computed the accuracy, generated a classification report, and plotted the confusion matrix. It is interesting to note throughout this report, the complexity of the models has been increasing, and the feature importance scores have been decreasing. We observe more simple models, such as LDA, highlight a few dominant features and use those in the decision-making process. More complicated models, such as Random Forest, distribute this importance across features, allowing for a more balanced utilization of all available data.

```
 # Print classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=le.classes_))
```

**Output:**

```
Accuracy of Random Forest model: 93.59%

Classification Report:
                precision    recall  f1-score   support
```

```
      benign       0.95      0.97      0.96     85778
  defacement       0.97      0.99      0.98     19104
     malware       0.98      0.94      0.96      6521
    phishing       0.81      0.75      0.78     18836

    accuracy                           0.94    130239
   macro avg       0.93      0.91      0.92    130239
weighted avg       0.93      0.94      0.93    130239
```

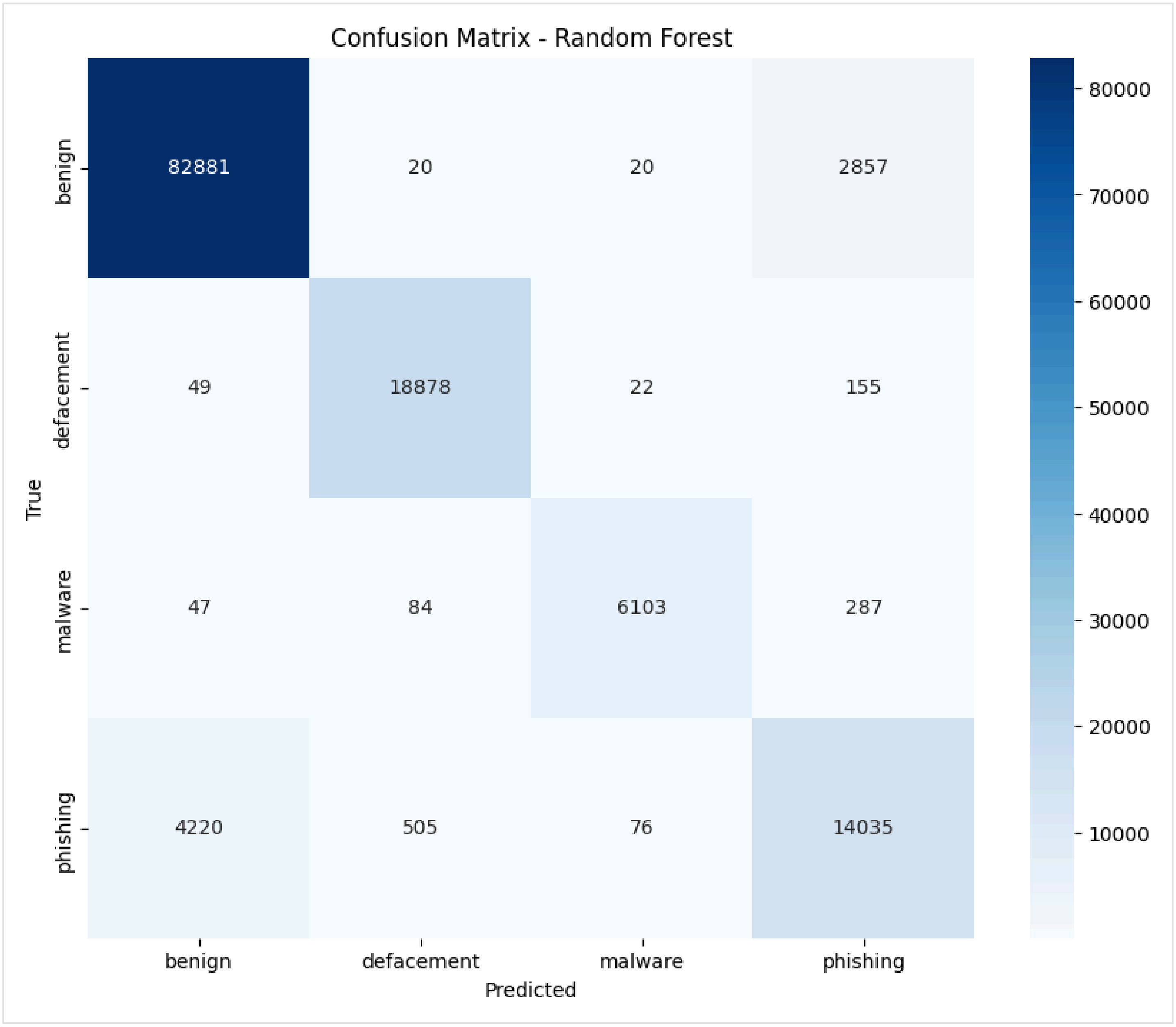


*Figure 10: Confusion matrix displaying the Random Forest model's performance across different classes.*

## Feature Importance

The feature importances from the Random Forest model indicate which features contribute most to the model's decisions.

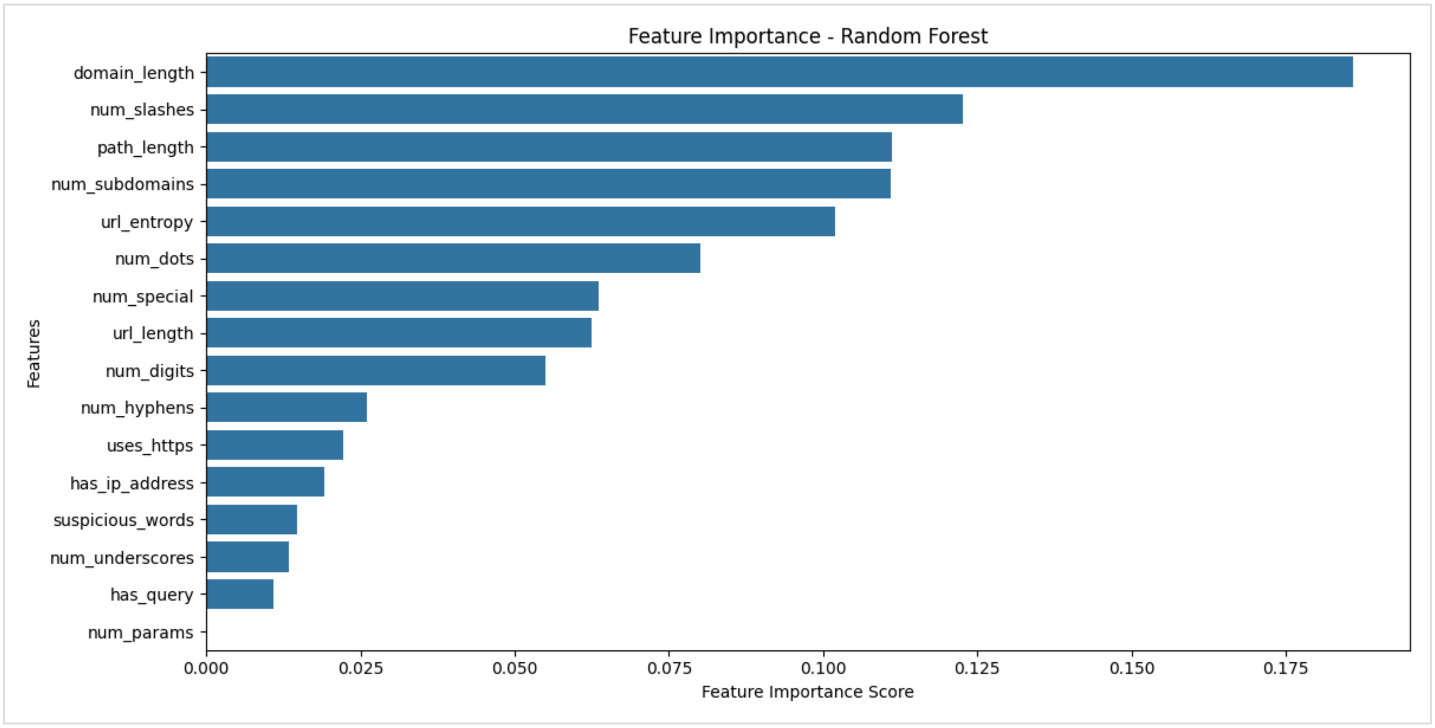


*Figure 11: Bar chart showing the importance of each feature based on the Random Forest model.*

We observe Random Forest has the highest accuracy scores compared to the other models discussed in this report. Compared to other models, Random Forest observed maintaining high precision and recall across all classes - even minority classes such as phishing. We find that since Random Forest learns from minority classes without performance degradation, it effectively can handle a class imbalance.

Furthermore, a primary motivation behind the usage of Random Forest is its ability to capture complex interactions and non-linear relationships between features. We observe this as an essential component of the model as the dataset is complex in nature.

While the model did perform the best, we do observe a significantly longer training time, thereby making it more computationally expensive compared to other models. Multiple trees must also be stored, which contributes to the resource strain with regard to space and memory usage.

## Model Comparison

We compared the performance of all four models using metrics like accuracy, precision, recall, and F1-score.

| Model | Accuracy (%) | Highest F1-Score | Lowest F1-Score | Eecution Time |
|---|---|---|---|---|
| LDA | 81.05 | 0.90 | 0.10 | 0.9s |
| Logistic Regression | 83.24 | 0.91 | 0.23 | 2.7s |
| LinearSVC | 82.65 | 0.90 | 0.14 | 11.5s |
| Random Forest | 93.59 | 0.98 | 0.78 | 80.1s |

In this project, we observed the performance of four supervised machine learning models - Linear Discriminant Analysis (LDA), Logistic Regression, Linear Support Vector Classifier (LinearSVC), and Random Forest Classifier. These models were used to classify URLs as benign or malicious. Each model was evaluated on its accuracy, precision, recall, F1-Score, and runtime. In this section, we will primarily be using metrics of accuracy, min and max F1-Score, and runtime to formulate analyses.

**Linear Discriminant Analysis (LDA)**

LDA achieved the lowest recorded accuracy, with significant misidentification for phishing URLs, resulting in the lowest F1-Score of 0.10. The model performed well in the identification of benign URLs but struggled with minority classes. We find the model assumes a normal distribution of features which does not hold for the dataset; furthermore, we find LDA underperformes on classes with overlapping feature distributions - something that is evident in Figure 3.

**Logistic Regression**

We find Logistic Regression was the second most effective supervised model, achieving an accuracy of 83.24%. This model outperformed LDA but still had issues classifying minority classes such as malware and phishing URLs. We observe Logistic Regression assumes a linear relationship between features and odd-logs, which does not capture complex patterns as is necessary for representing this dataset. We do, however, find surprising results with regard to the high accuracy and relatively low runtime.

**Linear Support Vector Classifier (LinearSVC)**

LinearSVC achieved results very similar to Logistic Regression, which is not surprising due to the similar linear nature of the two models. We largely attribute the shortcomings of LinearSVC as a result of the dataset not having a clearly defined margin between classes - a violation of the primary assumption of the model. We observe additional similarities with Logistic Regression in the results. While Logistic Regression exceeds LinearSVC primarily with regards to the lowest F1-Score, we observe other metrics (e.g. accuracy, highest F1-Score, and runtime) are all similar. Had the data been processed more to allow linear separability, we expect the results of Logistic Regression and LinearSVC to increase dramatically.

**Random Forest Classifier**

The Random Forest Classifier significantly outperformed all other models with an accuracy of 93.59. We observe this model to be the most complex, thereby being the most effective in handling the complexities of the dataset. A primary downside of the dataset is the higher computational complexity and longer training times, with a runtime of over a minute. While this may not seem significant, visualizations of the model require computation, on average, two orders of magnitude more than the runtime. Since the Random Forest Classifier was the best-performing model, we wished to generate a visualization for the model. However, after training for 10 hours without a result, the visualization effort was aborted.

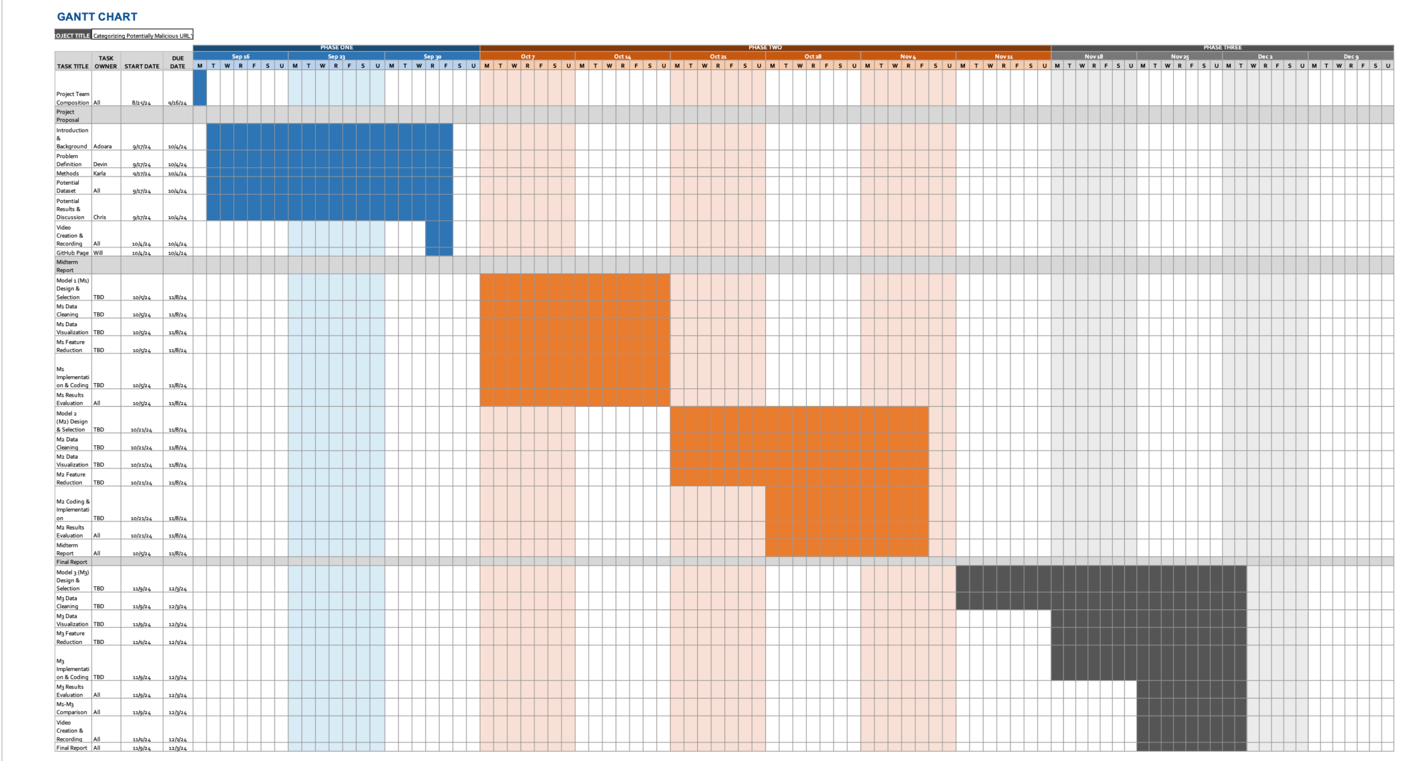
**Real-World Performance Analysis**

The performance of the models suggests Random Forest is the most suitable for the URL classification task. A large consideration, however, are use cases of the model. We intended to classify models found in emails. Having users wait times between 50-70 seconds for the Random Forest Classifier model to return output is not realistic. An alternative, however, may be utilizing a Random Forest Classifier for passive identification (i.e. when an email arrives in an inbox and is not immediately checked) and a faster model, such as Logistic Regression, for active monitoring. We observe this solution as the least invasive to users and would provide the best balance between safety and seamless usage of email services.

## Next Steps

- **Hyperparameter Tuning:** Fine-tune the hyperparameters of each model, especially Random Forest, to potentially improve performance further.
- **Cross-Validation:** Implement k-fold cross-validation to ensure the models generalize well to unseen data.
- **Feature Engineering:** Explore additional features or alternative feature extraction methods, such as text analysis of the URL strings.
- **Ensemble Methods:** Consider combining models to create an ensemble classifier for better performance.
- **Data Augmentation** Utilize SMOTE to balance class distribution and mitigate oversampling concerns.

## Project Timeline



OJECT TITLE: Categorizing Potentially Malicious URL's

| TASK TITLE | TASK OWNER | START DATE | DUE DATE |
|---|---|---|---|
| Project Team Composition | All | 8/25/24 | 9/16/24 |
| Project Proposal | | | |
| Introduction & Background | Adoara | 9/17/24 | 10/4/24 |
| Problem Definition | Devin | 9/17/24 | 10/4/24 |
| Methods | Karla | 9/17/24 | 10/4/24 |
| Potential Dataset | All | 9/17/24 | 10/4/24 |
| Potential Results & Discussion | Chris | 9/17/24 | 10/4/24 |
| Video Creation & Recording | All | 10/4/24 | 10/4/24 |
| GitHub Page | Will | 10/4/24 | 10/4/24 |
| Midterm Report | | | |
| Model 1 (M1) Design & Selection | TBD | 10/5/24 | 11/8/24 |
| M1 Data Cleaning | TBD | 10/5/24 | 11/8/24 |
| M1 Data Visualization | TBD | 10/5/24 | 11/8/24 |
| M1 Feature Reduction | TBD | 10/5/24 | 11/8/24 |
| M1 Implementation & Coding | TBD | 10/5/24 | 11/8/24 |
| M1 Results Evaluation | All | 10/5/24 | 11/8/24 |
| Model 2 (M2) Design & Selection | TBD | 10/21/24 | 11/8/24 |
| M2 Data Cleaning | TBD | 10/21/24 | 11/8/24 |
| M2 Data Visualization | TBD | 10/21/24 | 11/8/24 |
| M2 Feature Reduction | TBD | 10/21/24 | 11/8/24 |
| M2 Coding & Implementation | TBD | 10/21/24 | 11/8/24 |
| M2 Results Evaluation | All | 10/21/24 | 11/8/24 |
| Midterm Report | All | 10/5/24 | 11/8/24 |
| Final Report | | | |
| Model 3 (M3) Design & Selection | TBD | 11/9/24 | 12/3/24 |
| M3 Data Cleaning | TBD | 11/9/24 | 12/3/24 |
| M3 Data Visualization | TBD | 11/9/24 | 12/3/24 |
| M3 Feature Reduction | TBD | 11/9/24 | 12/3/24 |
| M3 Implementation & Coding | TBD | 11/9/24 | 12/3/24 |
| M3 Results Evaluation | All | 11/9/24 | 12/3/24 |
| M1-M3 Comparison | All | 11/9/24 | 12/3/24 |
| Video Creation & Recording | All | 11/9/24 | 12/3/24 |
| Final Report | All | 11/9/24 | 12/3/24 |

*Figure 12: Gantt chart showing the project timeline.*

## Group Contributions

| Group Member | Contributions |
|---|---|
| Eickman, William | Spearheaded data preprocessing, LDA implementation, and worked on proposal video presentation |
| De Jesus, Karla | Implemented Logistic Regression, assisted in LDA implementation, and worked on final video presentation |
| Fromond, Devin | Developed LinearSVC model, GitHub pages deliverable, and model visualizations |
| Njoku, Adaora | Implemented Random Forest Classifier, assisted in troubleshooting of models, and worked on final video presentation |
| Parker, Christopher | Assisted in data preprocessing, model evaluation, and optimized model performances |

## References

[1] M. Siddhartha, "Malicious URLs dataset," Kaggle.com, 2016. Accessed Oct. 04, 2024. https://www.kaggle.com/datasets/sid321axn/malicious-urls-dataset.

[2] "LinearDiscriminantAnalysis," scikit-learn, 2024. Accessed Nov. 13, 2024. https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html.

[3] "LinearSVC," scikit-learn, 2024. Accessed Nov. 17, 2024. https://scikit-learn.org/dev/modules/generated/sklearn.svm.LinearSVC.html.

[4] "LogisticRegression," scikit-learn, 2014. Accessed Nov. 20, 2024. https://scikit-learn.org/1.5/modules/generated/sklearn.linear_model.LogisticRegression.html.

[5] "RandomForestClassifier," scikit-learn, 2024. Accessed Nov. 27, 2024. https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[6] Scikit-learn, "6.3. Preprocessing Data," Accessed Oct. 04, 2024. https://scikit-learn.org/stable/modules/preprocessing.html.

[7] C. Chong, D. Liu, W. Lee, and Neustar, "Malicious URL Detection," Stanford University, 2012. Accessed Oct. 04, 2024. https://cs229.stanford.edu/proj2012/ChongLiu-MaliciousURLDetection.pdf.

[8] N. Buhl, "Training, Validation, Test Split for Machine Learning Datasets," Encord.com, Jun. 13, 2023. Accessed Nov. 04, 2024. https://encord.com/blog/train-val-test-split/.

[9] Z. Luna, "Feature Selection in Machine Learning: Correlation Matrix | Univariate Testing | RFECV," *Medium*, Jul. 27, 2021. Accessed Nov. 07, 2024. https://medium.com/geekculture/feature-selection-in-machine-learning-correlation-matrix-univariate-testing-rfecv-1186168fac12.