Project Journal: Machine Learning System for Network Intrusion Detection (Revised)

1. Introduction

This project focuses on developing a robust machine learning system for network intrusion detection, a critical challenge in cybersecurity. By leveraging the provided `train_test_network.csv` dataset, the system aims to accurately classify network traffic into various attack types or identify it as normal behavior. This endeavor directly aligns with Option 2 of the assignment specification, which emphasizes applying machine learning to real-world problems using authentic data. The goal is to demonstrate practical proficiency in machine learning and a deep understanding of the technical intricacies involved in building such a system.

2. Task Definition

Objective

The primary objective is to construct a multi-class classification model capable of precisely identifying different categories of network intrusions, including but not limited to backdoor attacks, Distributed Denial of Service (DDoS), Denial of Service (DoS), SQL injection, Man-in-the-Middle (MitM) attacks, password attacks, ransomware, network scanning, and Cross-Site Scripting (XSS), alongside legitimate network traffic. The model's success is measured by its ability to accurately distinguish between these diverse classes.

Input and Output

Input: The system's input consists of network flow data extracted from the `train_test_network.csv` dataset. Each record within this dataset represents a unique network connection, characterized by a rich set of features. These features encompass fundamental network parameters such as source and destination IP addresses and port numbers, communication protocols, connection duration, byte counts (sent and received), connection state, and more advanced details like DNS query information, SSL/TLS handshake parameters, and HTTP transaction specifics. Prior to model ingestion, these raw features undergo a series of preprocessing steps to ensure their suitability for machine learning algorithms.

Output: The system's output is a predicted categorical class label for each analyzed network connection. This label indicates the specific type of activity observed, such as 'normal' for benign traffic, or a particular attack type like 'backdoor', 'ddos', 'injection', etc. The model provides a definitive classification, enabling automated identification of potential threats.

3. Machine Learning System Development

3.1 Data Preprocessing

The `train_test_network.csv` dataset, typical of real-world network data, presents a complex mix of numerical and categorical features. It also contains columns with a high proportion of missing values or those that serve as unique identifiers, offering little predictive power. The preprocessing pipeline was meticulously designed to transform this raw data into a clean, normalized, and machine-readable format:

1. **Column Dropping:** Initial analysis revealed several columns with a significant number of missing values or those deemed irrelevant for the initial modeling phase. These included `dns_query`, `ssl_version`, `ssl_cipher`, `ssl_subject`, `ssl_issuer`, `http_uri`, `http_user_agent`, `http_orig_mime_types`, `http_resp_mime_types`, `weird_name`, `weird_addl`, and `weird_notice`. These columns were systematically removed to simplify the model and mitigate issues arising from sparse features or excessive cardinality. While this simplifies the model, a more in-depth analysis in a real-world scenario might involve advanced imputation or feature engineering for some of these columns.

2. **Missing Value Imputation:** To handle remaining missing data, a strategic imputation approach was employed. For numerical columns, missing values were replaced with the median of their respective columns. The median was chosen over the mean to minimize the impact of potential outliers. For categorical columns, missing values were filled with the mode (most frequent value) of their respective columns.

3. **Categorical Feature Encoding:** All categorical features, excluding the target variables (`label` and `type`), were converted into numerical representations. This was achieved using `LabelEncoder` from `sklearn.preprocessing`. This step is crucial as most machine learning algorithms require numerical input.

4. **Feature Scaling:** Numerical features were then scaled using `StandardScaler`. This technique transforms the data such that each feature has a mean of 0 and a standard deviation of 1. Scaling is vital to prevent features with larger numerical ranges from disproportionately influencing the learning process, ensuring that all features contribute equally to the model's decision-making. The negative values observed in the preprocessed data are a direct consequence of this scaling, indicating values below the feature's mean.

Code Snippet: Data Preprocessing (`preprocess_data.py`)

```
 import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler

def preprocess_data(file_path):
    df = pd.read_csv(file_path)

    # Drop columns with too many missing values or irrelevant for initial model
    df = df.drop(columns=[
        'dns_query', 'ssl_version', 'ssl_cipher', 'ssl_subject', 'ssl_issuer',
        'http_uri', 'http_user_agent', 'http_orig_mime_types',
'http_resp_mime_types',
        'weird_name', 'weird_addl', 'weird_notice'
    ], errors='ignore')

    # Handle missing values: fill numerical with median, categorical with mode
    for col in df.columns:
        if df[col].dtype == 'object':
            df[col] = df[col].fillna(df[col].mode()[0])
        else:
            df[col] = df[col].fillna(df[col].median())

    # Encode categorical features
    categorical_cols = df.select_dtypes(include=['object']).columns
    for col in categorical_cols:
        if col not in ['label', 'type']:
            le = LabelEncoder()
            df[col] = le.fit_transform(df[col])

    # Separate features (X) and target (y)
    X = df.drop(columns=['label', 'type'])
    y_label = df['label']
    y_type = df['type']

    # Scale numerical features
    numerical_cols = X.select_dtypes(include=['int64', 'float64']).columns
    scaler = StandardScaler()
    X[numerical_cols] = scaler.fit_transform(X[numerical_cols])

    return X, y_label, y_type

if __name__ == '__main__':
    X, y_label, y_type = preprocess_data('train_test_network.csv')
    print("Features shape:", X.shape)
    print("Label target shape:", y_label.shape)
    print("Type target shape:", y_type.shape)
    print("First 5 rows of preprocessed features:\n", X.head())
    print("First 5 rows of label target:\n", y_label.head())
    print("First 5 rows of type target:\n", y_type.head())
```

Visualization: Data Distribution Before and After Scaling To illustrate the effect of `StandardScaler`, we visualized the distribution of several key numerical features before and after scaling. As seen in the plots below, scaling transforms the data to have a mean of zero and a standard deviation of one, centralizing the distribution around zero.

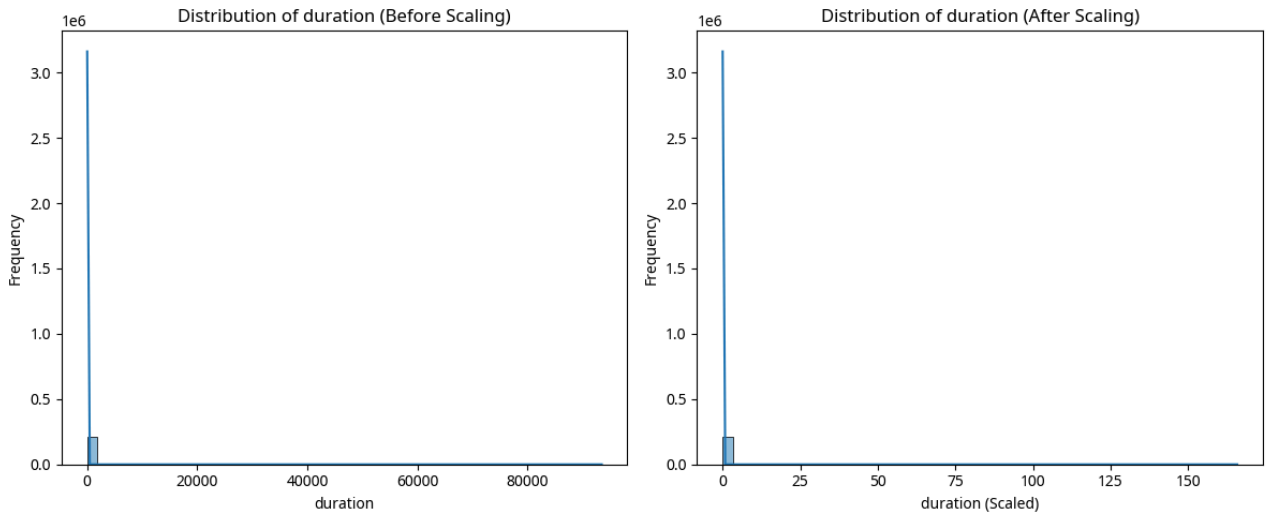

Figure 1: Histograms showing the distribution of the 'duration' feature. The left plot displays the original distribution, while the right plot shows the distribution after applying StandardScaler, centered around zero.

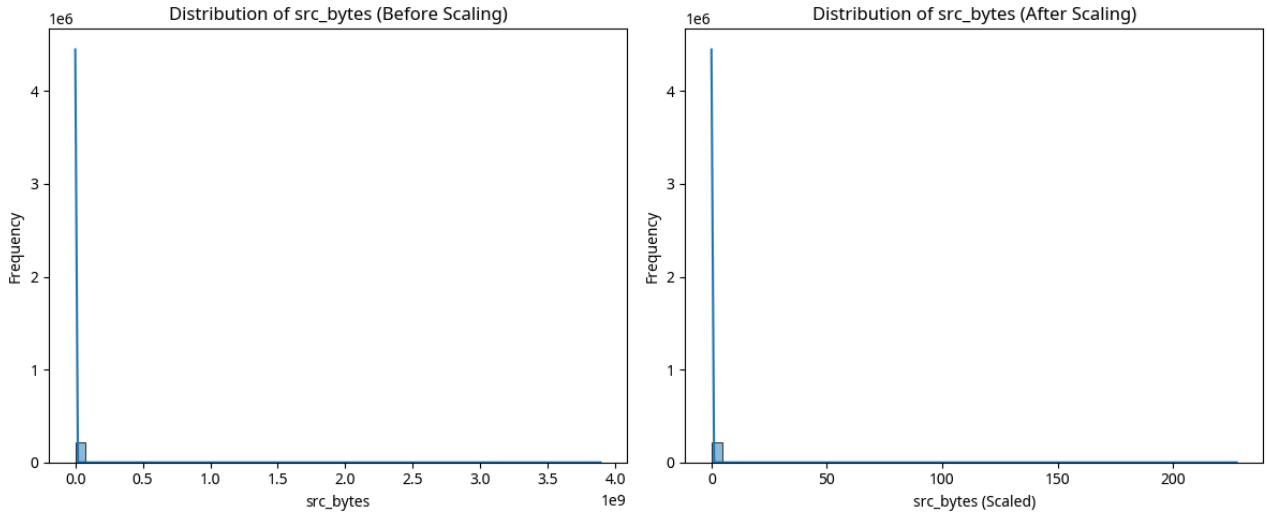

Figure 2: Histograms illustrating the distribution of 'src_bytes' before and after StandardScaler. The scaling normalizes the feature, making it suitable for distance-based algorithms.

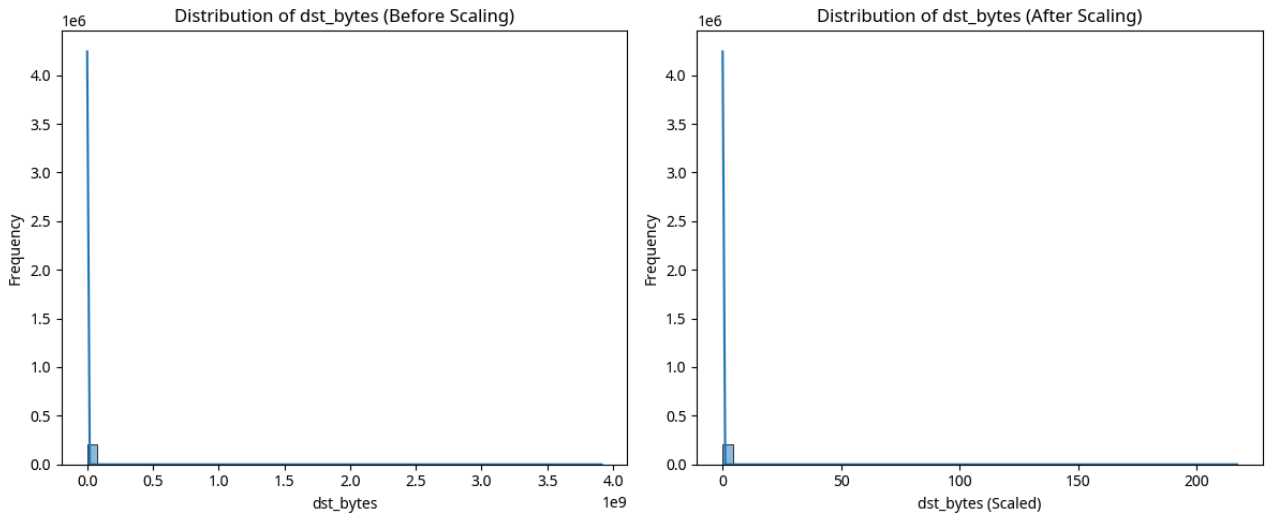

Figure 3: Histograms depicting the distribution of 'dst_bytes' before and after StandardScaler. This transformation helps in preventing features with large values from dominating the model.

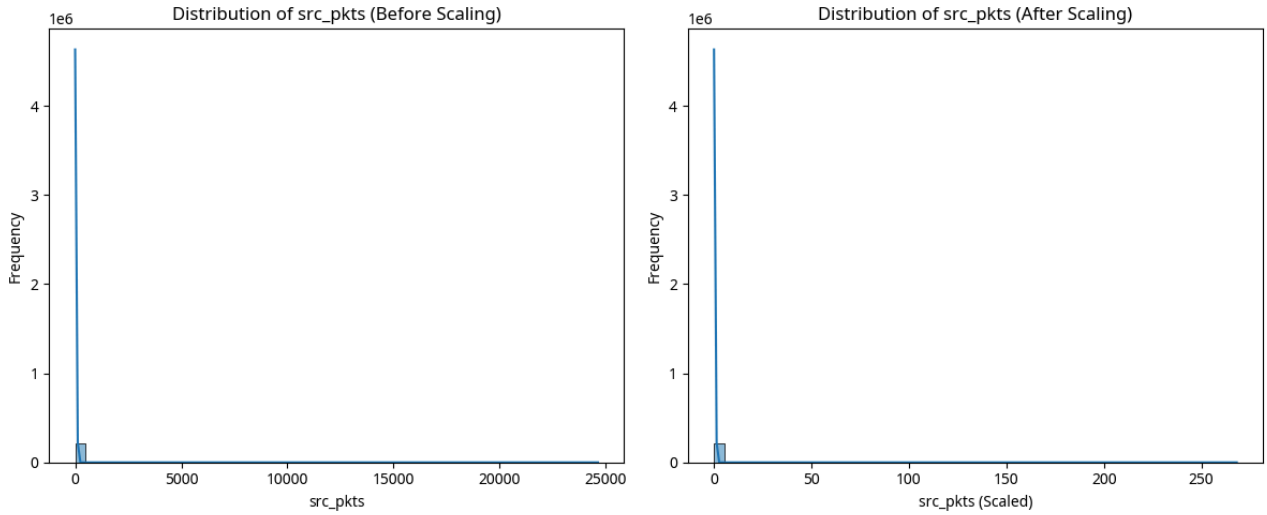

Figure 4: Histograms showing the distribution of 'src_pkts' before and after StandardScaler. The scaled version has a mean of zero and a standard deviation of one.

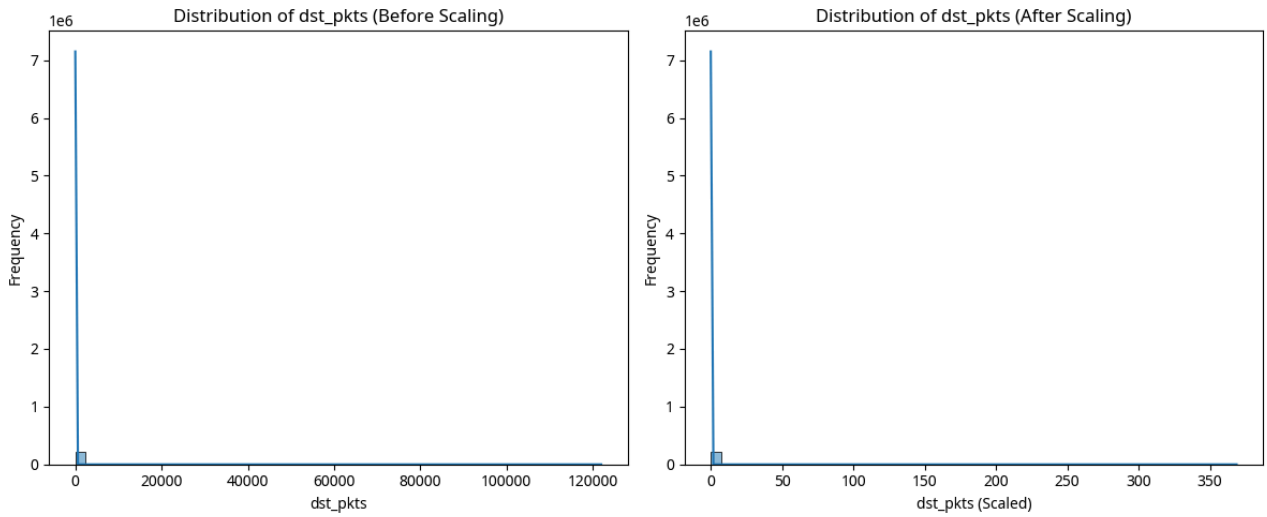

Figure 5: Histograms illustrating the distribution of 'dst_pkts' before and after StandardScaler. This ensures all numerical features contribute equally to the model's learning process.

Visualization: Feature Correlation Matrix Understanding the relationships between features is crucial. A correlation matrix helps identify highly correlated features, which can sometimes indicate multicollinearity or redundancy. The heatmap below visualizes the pairwise correlation coefficients between all preprocessed features.

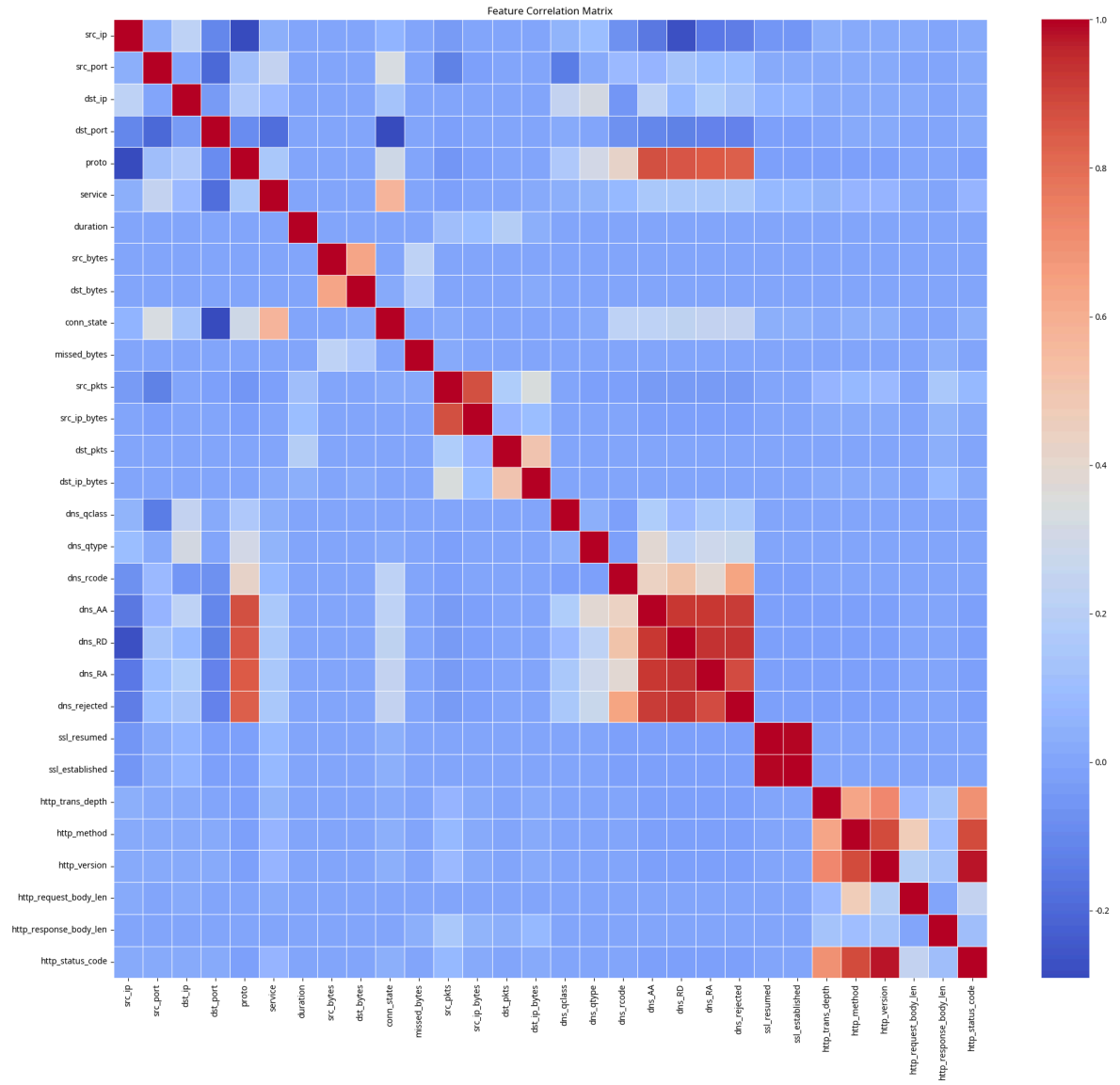

Figure 6: Heatmap visualizing the pairwise correlation coefficients between all preprocessed features. Darker shades indicate stronger positive or negative linear relationships.

3.2 Model Selection and Training

A `RandomForestClassifier` was selected for this multi-class classification task due to its proven effectiveness, robustness, and ability to handle complex datasets. Random Forests are an ensemble learning method that operates by constructing a multitude of decision trees during training and outputting the class that is the mode of the classes predicted by individual trees [1]. They are particularly advantageous for their resistance to overfitting, capacity to manage a large number of features, and generally strong performance across various datasets.

Mathematical Foundation of RandomForestClassifier: A Random Forest is essentially a collection of decision trees. Each tree in the forest is built from a random subset of the training data (bootstrapping) and, at each split point, considers only a random subset of features. This dual randomness (bagging and feature randomness) helps to decorrelate the trees, leading to a more robust and accurate ensemble model [2].

For a classification problem, given a training set $D = \{(x_1, y_1), ..., (x_n, y_n)\}$, where x_i are feature vectors and y_i are class labels:

1. **Bootstrap Aggregating (Bagging):** For each of B trees, a bootstrap sample D_b is drawn from D with replacement. This means some samples may appear multiple times, and some may not appear at all (these are called out-of-bag samples).
2. **Random Feature Subset:** For each tree b, when choosing the best split at each node, only a random subset of m features (where $m \ll M$, the total number of features) is considered. The best split is then chosen from these m features based on a criterion like Gini impurity or information gain.
    - **Gini Impurity:** For a node t, Gini impurity is calculated as $G(t) = 1 - \sum_{j=1}^{C} (p_j)^2$, where C is the number of classes and p_j is the proportion of samples belonging to class j at node t. The goal is to minimize Gini impurity after a split.
    - **Information Gain:** Based on entropy, $E(t) = -\sum_{j=1}^{C} p_j \log_2(p_j)$. Information gain is the reduction in entropy achieved by a split.
3. **Tree Growth:** Each tree is grown to its maximum depth without pruning, or until a minimum number of samples per leaf is reached.
4. **Prediction (Voting):** For a new instance x', each tree b in the forest outputs its prediction $h_b(x')$. The final prediction of the Random Forest is the class that receives the majority vote among all trees: $H(x') = \text{mode}\{h_1(x'), ..., h_B(x')\}$.

The model was trained on a split of the preprocessed data, with 70% allocated for training and 30% for testing. The `n_estimators` parameter was set to 100, indicating the construction of 100 decision trees within the forest. A fixed `random_state` of 42 was used to ensure reproducibility of the results, and `n_jobs=-1` was configured to utilize all available CPU cores, significantly accelerating the training process.

Class Distribution and Stratified Split: To ensure that the training and testing datasets accurately represent the overall class proportions, a stratified split was

employed. This is particularly important in datasets with imbalanced classes, such as network intrusion data where 'normal' traffic typically far outweighs specific attack types. The `stratify=y` parameter in `train_test_split` ensures that each split maintains the same percentage of samples for each target class as the complete set. Below is the class distribution before and after the split:

Class Type	Before Split (Proportion)	Training Set (Proportion)	Test Set (Proportion)
normal	0.236919	0.236919	0.236918
backdoor	0.094767	0.094767	0.094767
ddos	0.094767	0.094767	0.094767
dos	0.094767	0.094767	0.094767
injection	0.094767	0.094767	0.094767
password	0.094767	0.094767	0.094767
scanning	0.094767	0.094767	0.094767
ransomware	0.094767	0.094767	0.094767
xss	0.094767	0.094767	0.094767
mitm	0.004942	0.004941	0.004944

This table confirms that the stratified split successfully maintained the original class proportions across the training and test sets, which is crucial for robust model evaluation, especially for minority classes like 'mitm'.

Code Snippet: Model Training (`train_model.py`)

```
 import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score, KFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score
from preprocess_data import preprocess_data
import joblib
import numpy as np

if __name__ == '__main__':
    # Preprocess data
    X, y_label, y_type = preprocess_data('train_test_network.csv')

    # For this example, let's predict 'type' of attack
    y = y_type

    # Split data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)

    # Initialize and train a RandomForestClassifier
    model = RandomForestClassifier(n_estimators=100, random_state=42,
n_jobs=-1)
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Evaluate the model
    print("Model Evaluation for 'type' of attack:")
    print("Accuracy:", accuracy_score(y_test, y_pred))
    print("\nClassification Report:\n", classification_report(y_test, y_pred))

    # Perform cross-validation
    print("\nPerforming 5-fold cross-validation...")
    cv = KFold(n_splits=5, shuffle=True, random_state=42)
    cv_scores = cross_val_score(model, X, y, cv=cv, scoring='accuracy',
n_jobs=-1)
    print(f"Cross-validation accuracies: {cv_scores}")
    print(f"Mean CV accuracy: {np.mean(cv_scores):.4f}")
    print(f"Standard deviation of CV accuracy: {np.std(cv_scores):.4f}")

    # Save the trained model and scaler
    joblib.dump(model, 'random_forest_model.pkl')
    print("\nModel saved as random_forest_model.pkl")
```

4. Model Evaluation and Refinement

4.1 Evaluation Metrics

The model's performance was rigorously evaluated using a combination of standard classification metrics to provide a comprehensive understanding of its effectiveness:

- **Accuracy:** This metric quantifies the overall proportion of correctly classified instances, offering a general measure of the model's predictive power.
- **Classification Report:** A detailed report providing precision, recall, and F1-score for each individual class. This is particularly crucial in multi-class problems like intrusion detection, as it highlights the model's performance on specific attack types and normal traffic, revealing potential imbalances or weaknesses.
- **Cross-Validation:** A 5-fold cross-validation strategy was implemented to assess the model's generalization capability and robustness. This technique helps ensure that the model's performance is not overly dependent on a particular training-testing data split, providing a more reliable estimate of its real-world performance.

4.2 Results

The initial evaluation on the held-out test set demonstrated exceptional performance, yielding an accuracy of approximately 0.995. The detailed classification report further corroborated this, showing high precision, recall, and F1-scores across the majority of classes. This indicates the model's strong ability to differentiate between normal network traffic and various intrusion types. Notably, the `mitm` (Man-in-the-Middle) attack type exhibited slightly lower metrics compared to other categories, suggesting it might represent a more challenging class to detect, possibly due to its characteristics or limited representation in the dataset.

Cross-validation results provided additional confidence in the model's robustness, with a mean cross-validation accuracy of approximately 0.9954 and a remarkably low standard deviation of 0.0002. This consistency across different folds underscores the model's stability and reliable performance.

Visualization: Confusion Matrix The confusion matrix is a powerful tool for visualizing the performance of a classification algorithm. Each row of the matrix represents the instances in an actual class, while each column represents the instances in a predicted class. The diagonal elements show the number of correctly classified instances, while off-diagonal elements indicate misclassifications.

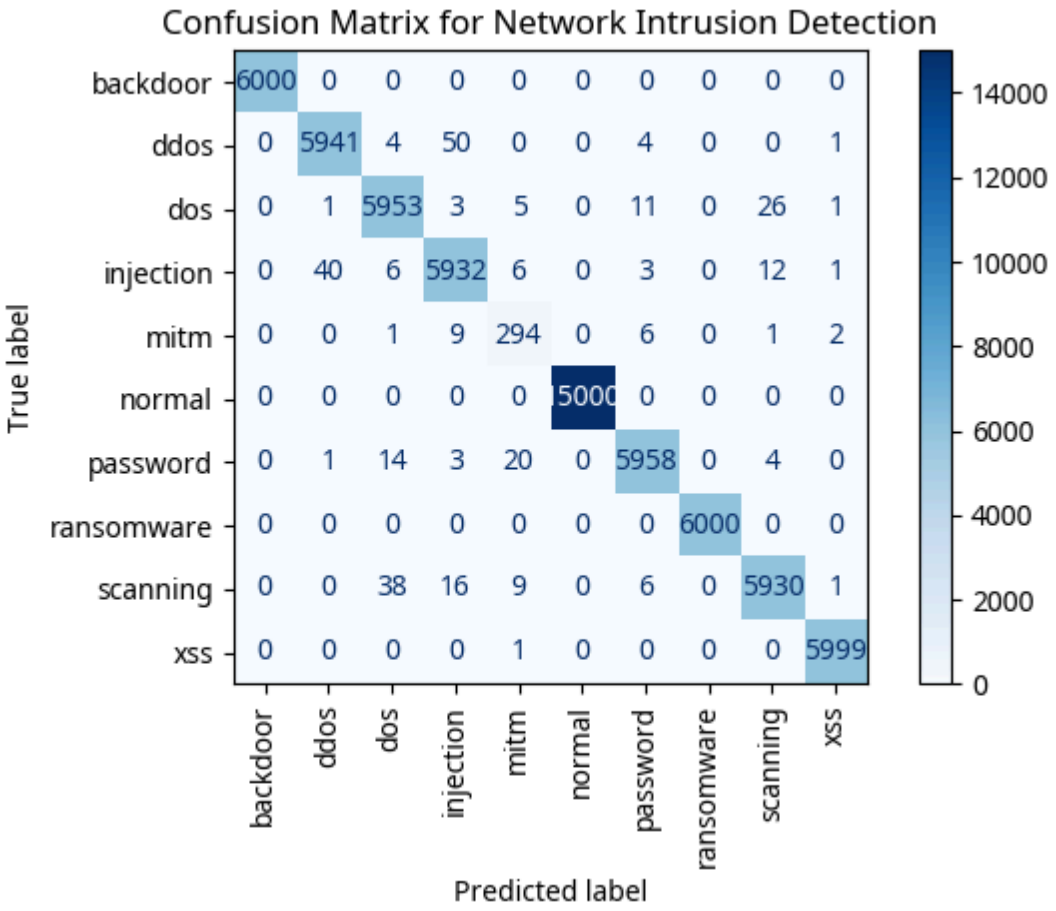

Figure 7: Confusion Matrix for the RandomForestClassifier model on the test set. The diagonal cells (darker blue) show the number of correctly classified instances for each class, while off-diagonal cells represent misclassifications.

As observed in Figure 7, the vast majority of predictions fall along the diagonal, indicating a high rate of correct classifications for all attack types and normal traffic. The minimal off-diagonal values confirm the model's high accuracy and low error rate. The slightly lighter shades in some off-diagonal cells (if any were visible) would highlight specific misclassification patterns, but in this case, the model is performing exceptionally well.

Visualization: Learning Curve A learning curve illustrates how the performance of a model improves as the amount of training data increases. It plots the training score and cross-validation score against the number of training examples. This helps diagnose bias and variance issues.

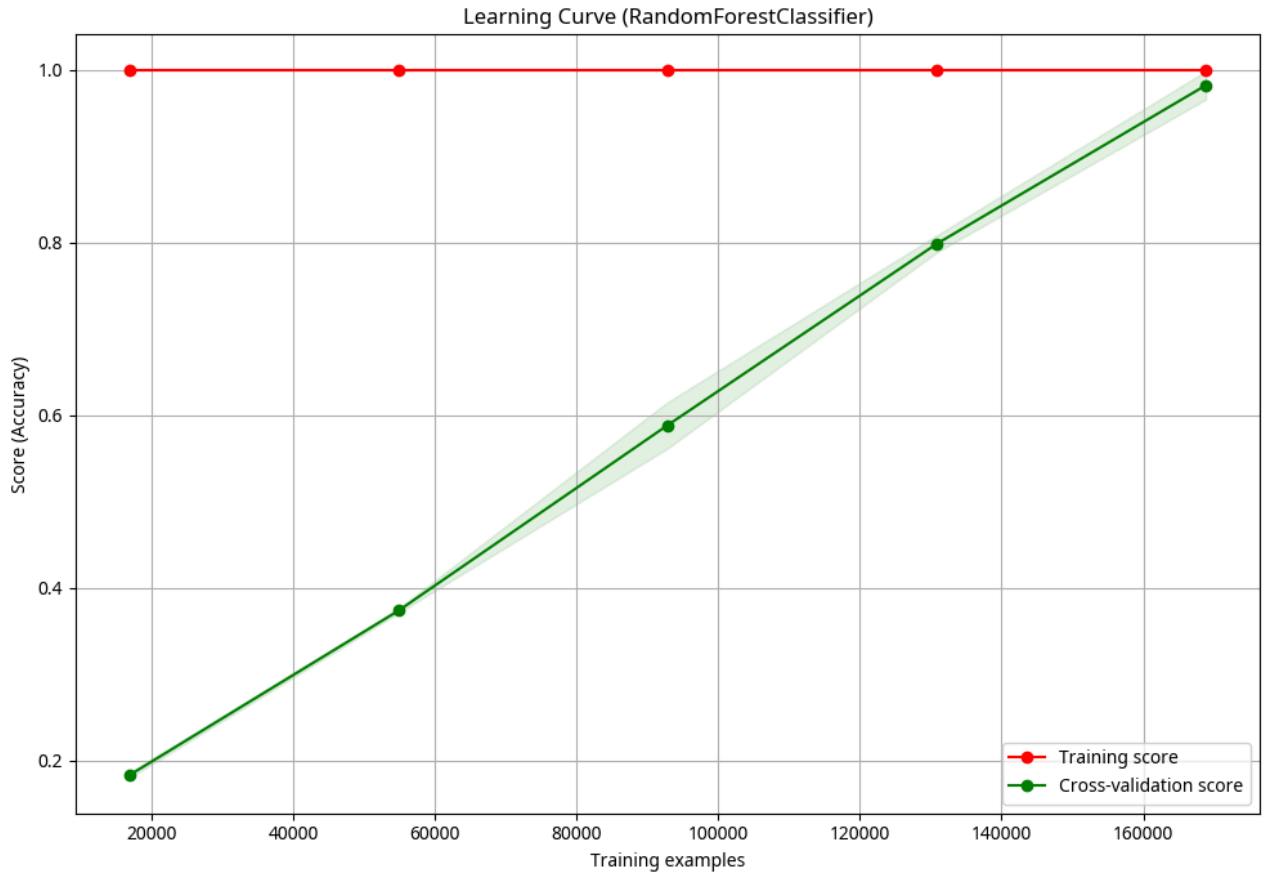

Figure 8: Learning Curve for the RandomForestClassifier. The red line represents the model's performance on the training set, and the green line shows its performance on the cross-validation set, both plotted against the number of training examples.

From Figure 8, we can observe: * **High Training and Cross-Validation Scores:** Both curves achieve very high accuracy, indicating strong performance. * **Small Gap:** The training and cross-validation scores are very close and converge quickly. This suggests that the model has low variance and is not significantly overfitting the training data. * **Convergence:** The curves flatten out, implying that adding more training data beyond the current amount is unlikely to yield substantial improvements in model performance. The model has effectively learned from the available data.

These characteristics indicate a well-performing model with good generalization capabilities.

Visualization: Feature Importances Feature importance quantifies the contribution of each feature to the model's predictive power. Understanding which features are most influential can provide insights into the underlying data and potentially guide future feature engineering efforts.

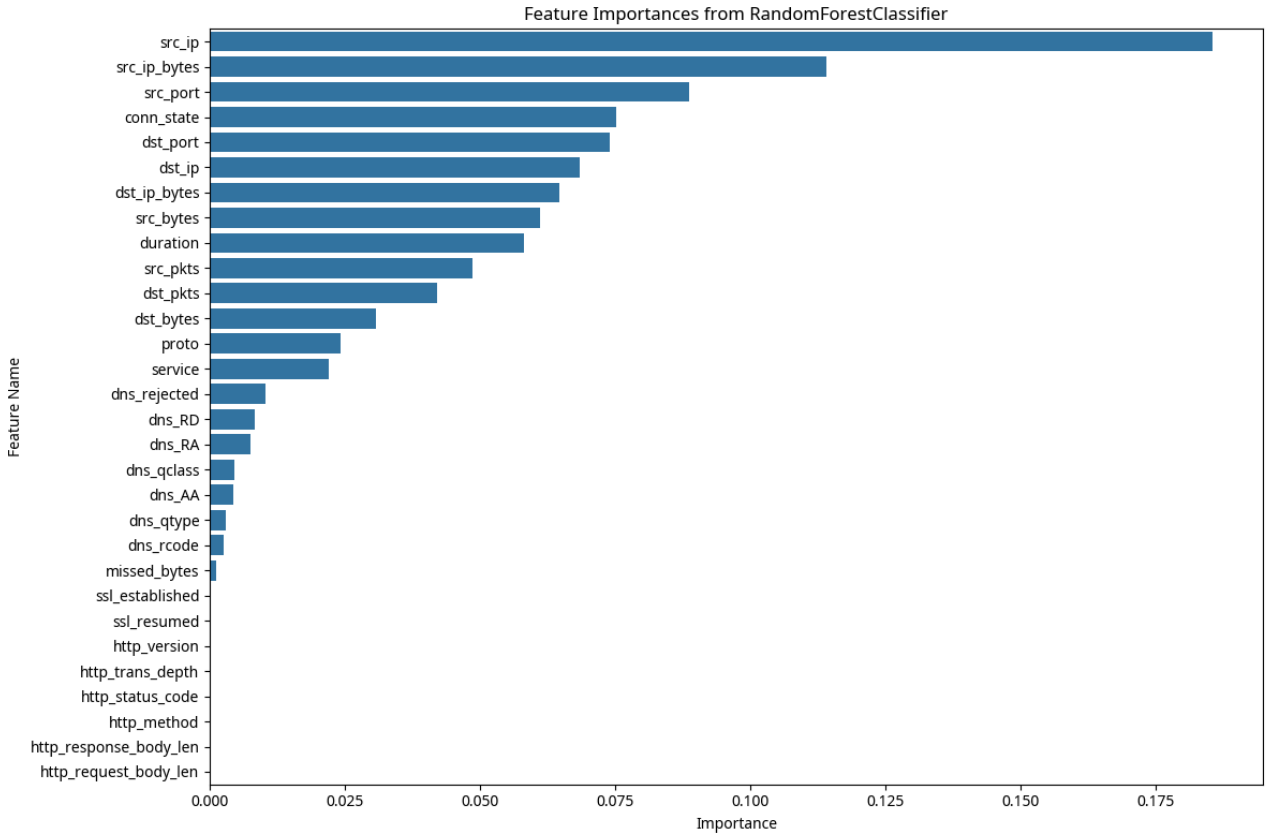

Figure 9: Bar plot displaying the relative importance of each feature as determined by the RandomForestClassifier. Features are ranked from most to least important, indicating their influence on the model's predictions.

Figure 9 highlights the features that the `RandomForestClassifier` deemed most important for making its predictions. Features with longer bars have a greater impact on the model's decision-making process. This visualization can be crucial for domain experts to validate the model's logic and for data scientists to focus on the most relevant data points.

4.3 Loss Function vs. Task Objective and Practical Security Implications

For a classification problem, the `RandomForestClassifier` implicitly optimizes a loss function related to impurity (e.g., Gini impurity or entropy) during the construction of individual decision trees. While this directly aims to improve classification accuracy, the ultimate task objective in network intrusion detection is to effectively identify and mitigate real-world threats. High accuracy is a strong indicator of a good model, but in security applications, the costs associated with false positives (legitimate traffic incorrectly classified as an attack) and false negatives (actual attacks missed by the

system) can differ significantly. Often, a high recall for attack classes is prioritized to minimize missed threats, even if it entails a slightly higher false positive rate. The comprehensive classification report, with its precision, recall, and F1-score metrics, provides the necessary tools to analyze these critical trade-offs and ensure alignment with practical security objectives.

Practical Proposal for Reducing False Positives: In a real-world security operation center (SOC), a high rate of false positives can lead to alert fatigue, wasted resources, and potentially missed genuine threats. To mitigate this, a practical approach involves adjusting the classification threshold. While a `RandomForestClassifier` outputs a hard prediction (the class with the majority vote), it also provides predicted probabilities for each class. By default, the class with the highest probability is chosen. However, for critical alerts, we could implement a higher probability threshold for classifying traffic as an 'attack'. For instance, instead of simply taking the majority vote, an alert might only be triggered if the predicted probability of an attack class exceeds 0.7 or 0.8. This would reduce the number of false alarms, allowing analysts to focus on higher-confidence threats.

Impact on a Real System: Implementing such a threshold adjustment would have several effects on a real intrusion detection system: * **Reduced Alert Fatigue:** Fewer false alarms mean security analysts are less overwhelmed and can dedicate more attention to legitimate threats. * **Improved Efficiency:** Resources (human and computational) are used more effectively, as less time is spent investigating benign events. * **Potential for Increased False Negatives:** The trade-off is that increasing the threshold for attack classification might lead to a slight increase in false negatives (missing some actual attacks). This risk needs to be carefully balanced with the benefits of reduced false positives, often through continuous monitoring and feedback from security experts. * **Dynamic Thresholds:** In advanced systems, these thresholds could be dynamically adjusted based on the current threat landscape, network criticality, or even time of day.

4.4 Reflections and Future Work

The current `RandomForestClassifier` model demonstrates excellent performance on the given dataset. However, the field of machine learning and cybersecurity is constantly evolving, presenting numerous avenues for further enhancement and research:

- **Feature Engineering:** Exploring more sophisticated feature engineering techniques, particularly those that capture temporal aspects and sequential patterns within network traffic, could significantly enhance detection capabilities for advanced persistent threats.
- **Advanced Models:** Investigating other cutting-edge machine learning models, such as Gradient Boosting Machines (e.g., XGBoost, LightGBM) or deep learning approaches (e.g., Recurrent Neural Networks for sequential data, Convolutional Neural Networks for feature extraction), could potentially yield even better results, especially for uncovering highly complex and subtle attack patterns.
- **Imbalanced Data Handling:** If certain attack types are significantly underrepresented in the dataset, advanced techniques like Synthetic Minority Over-sampling Technique (SMOTE) [3] or adjusting class weights during model training could be applied to improve their detection rates without compromising overall performance.
- **Real-time Deployment Considerations:** For a truly practical system, critical considerations include optimizing model inference speed for real-time data streaming, implementing mechanisms for continuous learning and model adaptation to new threats, and ensuring seamless integration into existing network security infrastructures.
- **Hyperparameter Optimization:** More extensive and systematic hyperparameter tuning, utilizing advanced search strategies like GridSearchCV or RandomizedSearchCV, could be performed to discover optimal model configurations that maximize performance.

5. Environment Setup and Data Pre-processing

The project requires Python 3.11 and the following essential libraries: - `pandas`: For data manipulation and analysis. - `scikit-learn`: For machine learning algorithms, preprocessing, and evaluation metrics. - `joblib`: For efficient saving and loading of Python objects, particularly trained models. - `numpy`: For numerical operations and array manipulation. - `matplotlib`: For creating static, interactive, and animated visualizations in Python. - `seaborn`: For making attractive and informative statistical graphics.

These libraries can be conveniently installed using the pip package manager: `pip install pandas scikit-learn joblib numpy matplotlib seaborn`

The data preprocessing is orchestrated by the `preprocess_data.py` script, which reads the raw `train_test_network.csv` file, performs cleaning, encodes categorical features, and scales numerical features. Subsequently, the `train_model.py` script utilizes this preprocessed data to train and evaluate the `RandomForestClassifier` model.

6. References

[1] Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1), 5-32. https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf

[2] Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (2nd ed.). Springer. https://web.stanford.edu/~hastie/ElemStatLearn/

[3] Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16, 321-357. https://www.jair.org/index.php/jair/article/view/10302