Artificial Intelligence

CSL 411

Project Report: Anomaly Detection in Network Traffic Using Explainable AI

Submitted by: Muhammad Ali Bukhari (01-135222-096)

Course: Introduction to Artificial Intelligence

Instructor: Farwa Kazmi

Semester: Spring 2025

Department of Computer Science

BAHRIA UNIVERSITY, ISLAMABAD

Abstract

This project addresses the challenge of detecting network anomalies using machine learning (ML) and explainable AI (XAI). A Random Forest classifier was trained on the NSL-KDD dataset to distinguish between normal and malicious network traffic. To enhance transparency, SHAP (SHapley Additive exPlanations) was employed to interpret model decisions. A Streamlit dashboard was developed to visualize real-time predictions, performance metrics, and SHAP-based explanations. The system achieves 99.48% recall and 92.09% accuracy , demonstrating robustness in identifying attacks while minimizing false positives.

Table of Contents

1. Introduction

1.1. Problem Statement

The increasing sophistication of cyber threats, coupled with the prevalence of encrypted network traffic, poses significant challenges for traditional anomaly detection systems. These systems often struggle to identify novel or unseen attack patterns and provide little insight into the reasons behind a detection, making it difficult for cybersecurity analysts to respond effectively. This project aims to bridge this gap by developing a robust and interpretable anomaly detection solution.

1.2. Project Objectives

The primary objectives of this project are:

- To effectively detect network anomalies using machine learning techniques.
- To explain the reasons behind detected anomalies, providing human-understandable insights.
- To develop a dashboard for real-time predictions and visualizations.
- To provide a strong foundation for future research, including zero-shot learning.

2. Dataset Description: NSL-KDD Dataset

The project utilizes the **NSL-KDD Dataset**. This dataset is a refined version of the KDD'99 dataset, which is widely used for network intrusion detection research. NSL-KDD addresses several inherent problems of the KDD'99 dataset, such as redundant records in the training set and duplicate records in the test set, which can lead to biased evaluation results and prevent models from learning effectively.

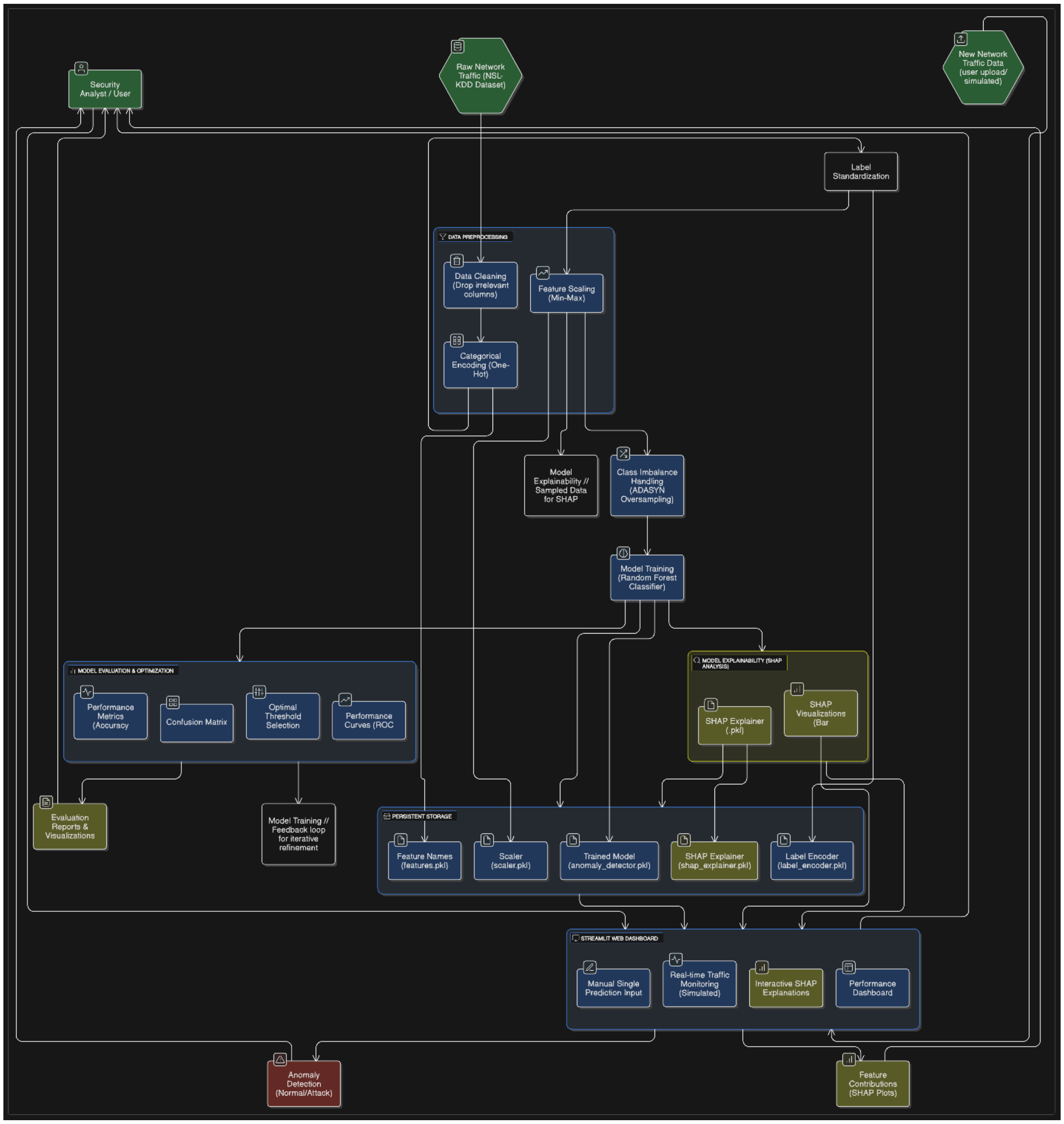
The dataset contains a variety of network connection records, each characterized by 41 features and labeled as either 'normal' or a specific type of 'attack'. The features capture various attributes of network connections, including:

- **Basic Features:** Duration of connection, protocol type (e.g., TCP, UDP, ICMP), service (e.g., http, ftp, smtp), flag (normal or error status), source and destination bytes.
- **Content Features:** Information about the data content, such as number of failed login attempts, number of file creations, etc..
- **Time-based Traffic Features:** Statistical properties calculated over a two-second window, like the count of connections to the same host.
- **Host-based Traffic Features:** Statistical properties computed over a window of 100 connections to the same destination host or service, capturing historical behavior.

For this project, the KDDTrain+.TXT and KDDTest+.TXT files are used for training and testing, respectively. The difficulty_level column is dropped, and all non-'normal' labels are standardized to 'attack' for binary classification (Normal vs. Attack).

3. System Architecture

The overall system architecture of the Advanced Network Anomaly Detection System with Explainable AI follows a structured pipeline, as depicted in the diagram below. This design emphasizes a clear flow from data ingestion to actionable, explainable insights, ensuring robust detection and transparency.


Security Analyst / User
Raw Network Traffic (NSL-KDD Dataset)
New Network Traffic Data (user upload/ simulated)
Label Standardization
DATA PREPROCESSING
Data Cleaning (Drop irrelevant columns)
Feature Scaling (Min-Max)
Categorical Encoding (One-Hot)
Model Explainability // Sampled Data for SHAP
Class Imbalance Handling (ADASYN Oversampling)
Model Training (Random Forest Classifier)
MODEL EVALUATION & OPTIMIZATION
Performance Metrics (Accuracy
Confusion Matrix
Optimal Threshold Selection
Performance Curves (ROC
MODEL EXPLAINABILITY (SHAP ANALYSIS)
SHAP Explainer (.pkl)
SHAP Visualizations (Bar
Evaluation Reports & Visualizations
Model Training // Feedback loop for iterative refinement
PERSISTENT STORAGE
Feature Names (features.pkl)
Scaler (scaler.pkl)
Trained Model (anomaly_detector.pkl)
SHAP Explainer (shap_explainer.pkl)
Label Encoder (label_encoder.pkl)
STREAMLIT WEB DASHBOARD
Manual Single Prediction Input
Real-time Traffic Monitoring (Simulated)
Interactive SHAP Explanations
Performance Dashboard
Anomaly Detection (Normal/Attack)
Feature Contributions (SHAP Plots)

Explanation of the Architecture:

1. **Raw Network Traffic (NSL-KDD Dataset):** This represents the initial input source for the system. It's the raw, heterogeneous network connection data that needs to be analyzed for anomalies.
2. **Data Preprocessing:** This is the foundational stage that prepares the raw data for machine learning.
    - **Data Cleaning:** Involves dropping irrelevant columns (e.g., difficulty_level).
    - **Categorical Encoding (One-Hot):** Converts nominal features (like protocol_type, service, flag) into numerical binary vectors, suitable for model input.

    - **Label Standardization (Binary: Normal/Attack):** Unifies diverse attack labels into a single 'attack' class, simplifying the problem to binary classification.
    - **Feature Scaling (Min-Max):** Normalizes numerical features to a common scale (0 to 1), preventing features with larger values from dominating the learning process.
3. **Class Imbalance Handling (ADASYN Oversampling):** Addressing the common issue in cybersecurity datasets where normal instances far outnumber attack instances.
    - **ADASYN (Adaptive Synthetic Sampling):** This technique generates synthetic samples for the minority (attack) class, focusing on those minority instances that are harder to classify. This balances the dataset, allowing the model to learn robustly from both normal and attack patterns.
4. **Model Training (Random Forest Classifier):** The core machine learning component where the detection intelligence is built.
    - A **Random Forest Classifier** is trained on the balanced and preprocessed training data. This ensemble method builds multiple decision trees and combines their predictions, enhancing accuracy and reducing overfitting.
    - The outcome of this stage is the **Trained Model** (anomaly_detector.pkl) and **Model Metrics** that detail its performance on the training data.
5. **Model Evaluation & Optimization:** This crucial phase assesses the trained model's generalization capability on unseen data.
    - **Performance Metrics:** Calculations of Accuracy, Precision, Recall, F1-Score, False Positive Rate (FPR), and Average Precision (PR-AUC) provide a comprehensive view of the model's effectiveness.
    - **Confusion Matrix:** Summarizes correct and incorrect predictions.
    - **Optimal Threshold Selection:** Determines the best probability threshold to convert model outputs into binary predictions, typically maximizing the F1-score to balance false positives and negatives.
    - **Performance Curves:** Visualizations like ROC (Receiver Operating Characteristic) and Precision-Recall curves aid in understanding model performance across different thresholds.
6. **Model Explainability (SHAP Analysis):** The Explainable AI (XAI) layer that provides transparency into the model's predictions.
    - **SHAP (SHapley Additive exPlanations):** Utilizes game theory to assign a contribution (SHAP value) to each feature for every individual prediction, indicating how much that feature pushed the prediction towards a particular outcome.
    - This stage generates a **SHAP Explainer** (shap_explainer.pkl) and various **SHAP Visualizations** (Bar, Beeswarm, Waterfall Plots) that explain feature importance globally and locally.
7. **Persistent Storage:** A central repository for saving essential model artifacts and preprocessing tools.
    - **Artifacts Stored:** This includes the Trained Model (.pkl), Scaler (.pkl), Feature Names (.pkl), Label Encoder (.pkl), and the SHAP Explainer (.pkl).
    - Storing these artifacts ensures the trained model and its associated components can be loaded efficiently for deployment without needing to retrain or reconfigure.
8. **Streamlit Web Dashboard:** The deployment and user interface layer, making the system accessible and interactive.
    - **Real-time Traffic Monitoring (Simulated):** Allows users to upload new network traffic data for immediate analysis.

    - **Interactive SHAP Explanations:** Enables users to explore the *why* behind each anomaly detection by visualizing feature contributions.
    - **Performance Dashboard:** Displays key performance metrics and visualizations from the evaluation phase.
    - The dashboard loads the necessary **Model Artifacts** from persistent storage to perform predictions and explanations.
9. **Security Analyst / User:** The human element interacting with the system.
    - Receives **Anomaly Detection** alerts (Normal/Attack classifications) and detailed **Feature Contributions (SHAP Plots)**.
    - This interaction enables informed decision-making and allows for a crucial **Feedback Loop** where insights from analysis can be used to further refine and improve the system over time, adapting to evolving threat landscapes.

4. Methodology: Project Pipeline Steps
The project follows a structured pipeline, encompassing several key steps from data preparation to model deployment and explanation. Each step is implemented using specific Python libraries and techniques.

4.1. Data Loading and Preprocessing
This initial phase prepares the raw NSL-KDD dataset for machine learning. It's critical for ensuring data quality and suitability for model training.

4.1.1. Column Definition and Data Loading
The NSL-KDD dataset files (KDDTrain+.TXT and KDDTest+.TXT) are loaded using pandas.read_csv. Column names are explicitly defined to ensure proper data interpretation.

```
# Define column names based on KDD99 dataset
column_names = [
    'duration', 'protocol_type', 'service', 'flag', 'src_bytes',
'dst_bytes',
    'land', 'wrong_fragment', 'urgent', 'hot', 'num_failed_logins',
    'logged_in', 'num_compromised', 'root_shell', 'su_attempted',
'num_root',
    'num_file_creations', 'num_shells', 'num_access_files',
'num_outbound_cmds',
    'is_host_login', 'is_guest_login', 'count', 'srv_count',
'serror_rate',
    'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate',
'same_srv_rate',
    'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_count',
'dst_host_srv_count',
    'dst_host_same_srv_rate', 'dst_host_diff_srv_rate',
'dst_host_same_src_port_rate',
    'dst_host_srv_diff_host_rate', 'dst_host_serror_rate',
'dst_host_srv_serror_rate',
    'dst_host_rerror_rate', 'dst_host_srv_rerror_rate', 'label',
'difficulty_level'
]
```

```
# Load raw data
train_df = pd.read_csv(os.path.join(dataset_path, 'KDDTrain+.TXT'),
header=None)
test_df = pd.read_csv(os.path.join(dataset_path, 'KDDTest+.TXT'),
header=None)

train_df.columns = column_names
test_df.columns = column_names
```

Behind the scenes of pd.read_csv: When pd.read_csv is called, Pandas performs several operations:

1. **File Opening:** It opens the specified .TXT file for reading.
2. **Row-by-Row Reading:** It reads the file line by line. Since header=None is specified, it treats the first line as data, not as column headers.
3. **Delimiter Detection:** It infers the delimiter (often comma, tab, or space) if not explicitly provided. For .TXT files, it often defaults to a comma or space as a separator. Given the column names structure with commas, it likely uses commas as delimiters.
4. **Data Parsing:** Each line is split into individual values based on the delimiter.
5. **DataFrame Construction:** These parsed values are then assembled into a tabular structure, forming a Pandas DataFrame.
6. **Column Assignment:** Finally, df.columns = column_names explicitly assigns the provided list of names to the DataFrame's columns.

4.1.2. Data Cleaning and Label Standardization

The difficulty_level column, which is not relevant for the classification task, is dropped. All attack-related labels (e.g., 'dos', 'probe', 'r2l', 'u2r') are consolidated into a single 'attack' label, while 'normal' remains 'normal' for binary classification.

```
for df in [train_df, test_df]:
    df.drop(['difficulty_level'], axis=1, inplace=True)
    df['label'] = df['label'].apply(lambda x: 'normal' if x ==
'normal' else 'attack')
```

Behind the scenes of df.drop and df.apply:

- df.drop(['difficulty_level'], axis=1, inplace=True):
    - axis=1 indicates that a column should be dropped.
    - inplace=True modifies the DataFrame directly, avoiding the creation of a new DataFrame.
    - Pandas identifies the column by name and physically removes it from the DataFrame's underlying data structure.
- df['label'].apply(lambda x: 'normal' if x == 'normal' else 'attack'):
    - This iterates through each value x in the 'label' column.
    - The lambda function is applied to each value: if x is 'normal', it remains 'normal'; otherwise, it's changed to 'attack'.
    - This effectively transforms multi-class labels into a binary classification problem.

4.1.3. Categorical Feature Encoding (One-Hot Encoding)
Categorical features (protocol_type, service, flag) are converted into a numerical format using one-hot encoding. This is crucial as machine learning models typically require numerical input. pd.get_dummies creates new binary columns for each unique category.

```
combined_df = pd.concat([train_df, test_df], axis=0)
combined_df = pd.get_dummies(combined_df, columns=['protocol_type',
'service', 'flag'])
train_df = combined_df.iloc[:len(train_df)].copy()
test_df = combined_df.iloc[len(train_df):].copy()
```

Behind the scenes of pd.get_dummies:

- For each categorical column, pd.get_dummies identifies all unique categories.
- For each unique category, a new column is created (e.g., protocol_type_tcp, service_http).
- In each new column, a 1 is placed if the original row had that category, and 0 otherwise.
- This transformation increases the dimensionality of the dataset but ensures that the model can process categorical information without implying an ordinal relationship between categories. Concatenating train and test before one-hot encoding ensures that all possible categories across both sets are captured, preventing inconsistencies in feature sets.

4.1.4. Label Encoding
The binary label column ('normal'/'attack') is converted into numerical representation (0/1) using sklearn.preprocessing.LabelEncoder. The code ensures that 'normal' is always mapped to 0 and 'attack' to 1, even if the alphabetical order of LabelEncoder would initially map them differently.

```
le = LabelEncoder()
le.fit(pd.concat([train_df['label'], test_df['label']]).unique())
train_df['label'] = le.transform(train_df['label']).astype(int)
test_df['label'] = le.transform(test_df['label']).astype(int)

# Ensure 'normal' is 0 and 'attack' is 1
if 'normal' in le.classes_ and 'attack' in le.classes_:
    if le.transform(['normal'])[0] == 1 and
le.transform(['attack'])[0] == 0:
        train_df['label'] = 1 - train_df['label']
        test_df['label'] = 1 - test_df['label']
        le.classes_ = np.array(['normal', 'attack'])
```

Behind the scenes of LabelEncoder:

- le.fit(data): LabelEncoder scans the unique values in the provided data (pd.concat([train_df['label'], test_df['label']]).unique()) and assigns an integer to each unique string label. By default, it assigns integers alphabetically.
- le.transform(data): For each string label in the input data, it replaces it with its corresponding integer mapping learned during the fit phase.
- The manual swapping logic (1 - df['label']) is a clever way to enforce the desired mapping (0 for 'normal', 1 for 'attack') regardless of LabelEncoder's default alphabetical assignment, ensuring consistency for subsequent model interpretation where '1' typically denotes the positive class (attack).

4.1.5. Feature Scaling (Min-Max Scaling)
Numerical features are scaled to a range of 0 to 1 using sklearn.preprocessing.MinMaxScaler. This prevents features with larger numerical ranges from dominating the learning process.

```
scaler = MinMaxScaler()
train_df[feature_columns] =
scaler.fit_transform(train_df[feature_columns])
test_df[feature_columns] = scaler.transform(test_df[feature_columns])
```

Behind the scenes of MinMaxScaler:

- scaler.fit_transform(X_train):
    - fit: The scaler calculates the minimum and maximum values for each feature in the training data (X_train).
    - transform: For each feature, it applies the formula: X_scaled = (X - X_min) / (X_max - X_min). This scales all values to be between 0 and 1.
- scaler.transform(X_test): The same min and max values *learned from the training data* are applied to the test data. This is crucial to prevent data leakage from the test set into the training process.

4.2. Class Imbalance Handling: ADASYN Oversampling
Network anomaly datasets often suffer from severe class imbalance, where normal instances significantly outnumber attack instances. This can lead to models that perform well on the majority class but poorly on the minority (attack) class. **ADASYN (Adaptive Synthetic Sampling)** is used to mitigate this by generating synthetic samples for the minority class.

```
adasyn = ADASYN(random_state=42)
X_res, y_res = adasyn.fit_resample(X_train, y_train)
```

How ADASYN Works (Behind the function call adasyn.fit_resample):

1. **Identify Minority and Majority Classes:** ADASYN first identifies the minority class (attacks) and the majority class (normal).
2. **Density Distribution:** Unlike simpler oversampling methods like SMOTE, ADASYN focuses on generating synthetic samples for minority class instances that are harder to learn. It does this by considering the density distribution of minority examples. It calculates a 'ratio' for each minority sample, which indicates how many majority neighbors it has. This effectively means more synthetic samples are generated for minority instances that are near majority instances (i.e., on the decision boundary).
3. **Synthetic Sample Generation:**
    - For each minority sample x_i, ADASYN finds its k nearest neighbors from the minority class.
    - For each neighbor x_zi (where x_zi is a minority instance), a synthetic sample is generated along the line segment connecting x_i and x_zi.
    - The new synthetic sample x_new is calculated as: x_new = x_i + (x_zi - x_i) * lambda, where lambda is a random number between 0 and 1.
4. **Adaptive Weighting:** The number of synthetic samples generated for each minority instance is inversely proportional to its density, meaning minority samples that are harder to classify (those surrounded by more majority samples) get more synthetic examples. This adaptively shifts the decision boundary to focus on the difficult-to-learn minority samples.

This process helps create a more balanced dataset, allowing the Random Forest model to learn more effectively from the underrepresented attack class.

4.3. Model Training: Random Forest Classifier

A **Random Forest Classifier** from sklearn.ensemble is chosen for its robustness, ability to handle high-dimensional data, and resistance to overfitting.

```
model = RandomForestClassifier(
    n_estimators=150,
    max_depth=20,
    min_samples_split=5,
    class_weight='balanced',
    random_state=42,
    n_jobs=-1
)
model.fit(X_train, y_train)
```

How Random Forest Works (Behind the function call model.fit): A Random Forest is an ensemble learning method that builds a "forest" of decision trees. It operates on the principle of "wisdom of the crowd," where the collective decision of many individual trees is more accurate and robust than a single tree.

1. **Bootstrap Aggregating (Bagging):**
    - **Data Sampling (Bootstrap):** For each tree in the forest, a random subset of the training data (with replacement) is sampled. This means some samples may be repeated, and some may be left out (called "out-of-bag" samples). This introduces diversity among the trees.
    - **Feature Randomness:** When building each tree, at every split point, only a random subset of features is considered. This further decorrelates the trees, making them less prone to making the same errors. This is particularly important for high-dimensional data like network traffic, where many features might be correlated.
2. **Tree Construction (n_estimators):**
    - n_estimators=150: This parameter specifies that 150 individual decision trees will be constructed.
    - Each tree is grown to its maximum depth (or until max_depth=20 or min_samples_split=5 is met).
    - max_depth=20: Controls the maximum depth of each tree. Deeper trees can capture more complex patterns but risk overfitting.
    - min_samples_split=5: A node will only be split if it contains at least 5 samples. This helps to prevent overfitting by requiring a minimum number of samples for a split.
    - class_weight='balanced': This parameter automatically adjusts weights inversely proportional to class frequencies, giving more importance to the minority class (attack) during training. This is another mechanism to combat class imbalance, complementing ADASYN.
    - random_state=42: Ensures reproducibility of the random sampling and tree construction.
    - n_jobs=-1: Utilizes all available CPU cores for parallel training of the trees, significantly speeding up the training process.
3. **Prediction (Voting):**

    - When a new data point needs to be classified, it is fed through all 150 decision trees.
    - Each tree independently makes a prediction (e.g., 'normal' or 'attack').
    - For classification, the Random Forest combines these predictions using a majority vote (or averages predicted probabilities). The class that receives the most votes across all trees is the final prediction.

Advantages of Random Forest:

- **High Accuracy:** Generally performs very well, especially on complex datasets.
- **Robustness to Overfitting:** The randomness introduced during bagging and feature selection makes it less prone to overfitting than individual decision trees.
- **Handles High Dimensionality:** Can manage datasets with many features without explicit feature selection.
- **Feature Importance:** Can inherently provide feature importances, although SHAP offers a more nuanced explanation.

4.4. Model Evaluation

After training, the model's performance is rigorously evaluated on the unseen test set to assess its generalization capabilities.

4.4.1. Performance Metrics

Standard classification metrics are calculated to provide a comprehensive view of the model's performance:

- **Accuracy:** The proportion of correctly classified instances (both normal and attack) out of the total.
- **Precision:** The proportion of correctly identified attack instances among all instances predicted as attack. High precision means fewer false alarms.
- **Recall (Sensitivity/True Positive Rate):** The proportion of correctly identified attack instances among all actual attack instances. High recall means fewer missed attacks.
- **F1-Score:** The harmonic mean of precision and recall, providing a balance between the two. It's especially useful for imbalanced datasets.
- **Average Precision (PR-AUC):** The area under the Precision-Recall curve. This is a robust metric for imbalanced datasets, as it focuses on the performance of the positive class (attack).
- **False Alarm Rate (FPR):** The proportion of normal instances incorrectly classified as attack out of all actual normal instances. Minimizing FPR is crucial in cybersecurity to avoid alert fatigue.
- **Confusion Matrix:** A table that summarizes the number of correct and incorrect predictions for each class, showing True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).
- **Classification Report:** A text summary providing precision, recall, f1-score, and support for each class.

4.4.2. Optimal Threshold Selection

For binary classification, model typically output probabilities. A threshold is then applied to convert these probabilities into binary class labels (0 or 1). Instead of using a default 0.5 threshold, an optimal threshold is determined by maximizing the F1-score from the Precision-Recall curve. This ensures a better balance between precision and recall, particularly important in anomaly detection where both false positives and false negatives have significant implications.

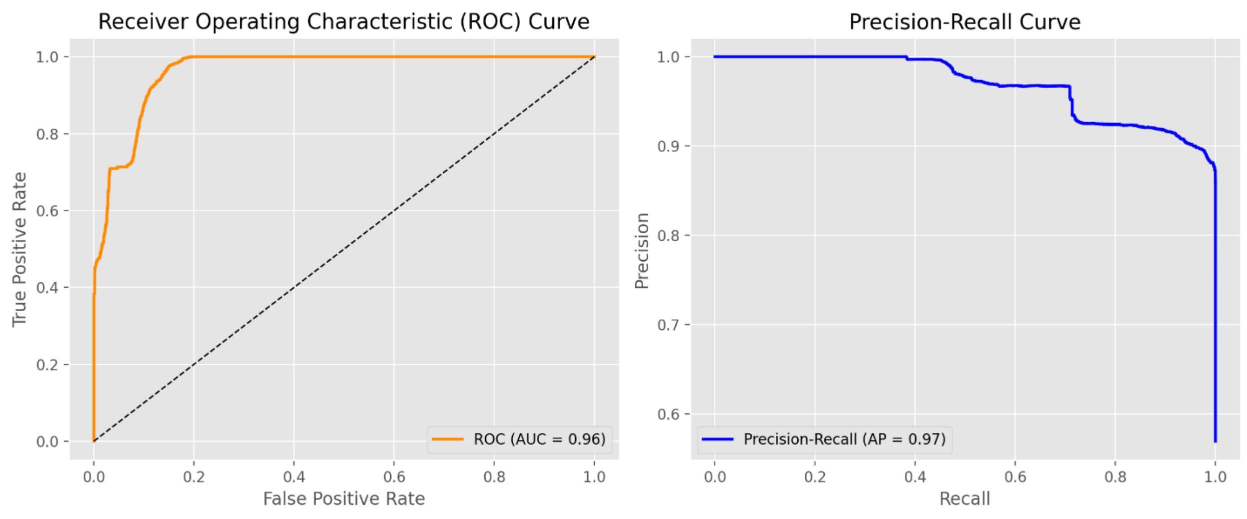
```
y_probs = model.predict_proba(X_test)[:, 1] # Probabilities for the
positive class (attack=1)

precision, recall, thresholds = precision_recall_curve(y_test,
y_probs)
f1_scores = 2 * (precision * recall) / (precision + recall + 1e-9)
# Added epsilon to avoid division by zero
optimal_idx = np.argmax(f1_scores)
optimal_threshold = thresholds[optimal_idx]

y_pred_optimal = (y_probs >= optimal_threshold).astype(int)
```

Behind the scenes of model.predict_proba and thresholding:

- model.predict_proba(X_test): For each instance in X_test, the Random Forest model calculates the probability of belonging to each class. Since it's an ensemble of trees, it averages the probabilities predicted by individual trees. It typically returns an array where each row contains [probability_of_class_0, probability_of_class_1]. [:, 1] extracts only the probabilities for the positive class (attack).
- precision_recall_curve(y_test, y_probs): This Scikit-learn function computes precision-recall pairs for different probability thresholds. It outputs three arrays: precision values, recall values, and the corresponding thresholds.
- f1_scores = 2 * (precision * recall) / (precision + recall + 1e-9): The F1-score is calculated for each precision-recall pair. The 1e-9 is added to prevent division by zero in cases where both precision and recall are zero.
- np.argmax(f1_scores): This finds the index of the maximum F1-score, which corresponds to the optimal threshold.
- optimal_threshold = thresholds[optimal_idx]: Retrieves the threshold value that yielded the highest F1-score.
- y_pred_optimal = (y_probs >= optimal_threshold).astype(int): This is the final step where the continuous probabilities (y_probs) are converted into discrete binary predictions (0 or 1) using the calculated optimal threshold.

4.4.3. Performance Curves (ROC & Precision-Recall)

Visualizations like ROC (Receiver Operating Characteristic) and Precision-Recall (PR) curves are generated to visually represent the model's performance across different thresholds.



Explanation of Performance Curves:

- **ROC Curve (Left Plot):**
    - Plots the True Positive Rate (TPR, or Recall) against the False Positive Rate (FPR) at various threshold settings.
    - The **Area Under the Curve (AUC)** is a single metric that summarizes the ROC curve; a higher AUC (closer to 1.0) indicates better model performance, meaning it can distinguish between positive and negative classes well. An AUC of 0.96 suggests excellent discriminatory power.
    - The dashed diagonal line represents a random classifier; a good model's curve will be significantly above this line.
- **Precision-Recall Curve (Right Plot):**
    - Plots Precision against Recall.
    - The **Average Precision (AP)** is the area under the Precision-Recall curve. This curve is particularly informative for imbalanced datasets because it focuses on the positive class and is not influenced by the number of true negatives. A high AP (closer to 1.0) indicates that the model performs well in identifying positive instances without generating too many false positives. An AP of 0.96 indicates strong performance on the attack class.
    - The shape of the curve provides insights into the trade-off between precision and recall.

4.5. Model Explanation: SHAP Analysis

SHAP (SHapley Additive exPlanations) is a powerful framework that explains the output of any machine learning model. It connects optimal credit allocation with local explanations using the classic Shapley values from game theory. SHAP values tell us how much each feature contributes to a prediction, both positively and negatively.

```
explainer = shap.TreeExplainer(model)
# ... (calculation of shap_values_attack)
# ... (generation of SHAP plots
```

How SHAP Works (Behind the function calls shap.TreeExplainer, explainer.shap_values, shap.summary_plot, shap.plots.waterfall):

SHAP values represent the average marginal contribution of a feature value across all possible coalitions of features. In simpler terms, for a single prediction, SHAP calculates how much each feature value pushes the prediction away from the average prediction (base value).

1. **Shapley Values:**
    - Inspired by cooperative game theory, Shapley values distribute the "gain" (the difference between the model's prediction and the average prediction) among the features.
    - The core idea is to consider all possible ways to "build up" a prediction by adding features one by one. For each feature, its contribution is calculated by seeing how much the prediction changes when that feature is added to a subset of other features, averaged over all possible orderings (permutations) of features.
    - This ensures that the feature's contribution is fairly attributed, accounting for interactions with other features.
2. **Additivity Property:** The sum of the SHAP values for all features, plus the base_value (the average prediction for the dataset), equals the actual prediction for that instance. This property makes SHAP explanations very intuitive: Sum of SHAP values + Base Value = Model Output

4.5.2. SHAP Explainer Types (TreeExplainer vs. KernelExplainer)

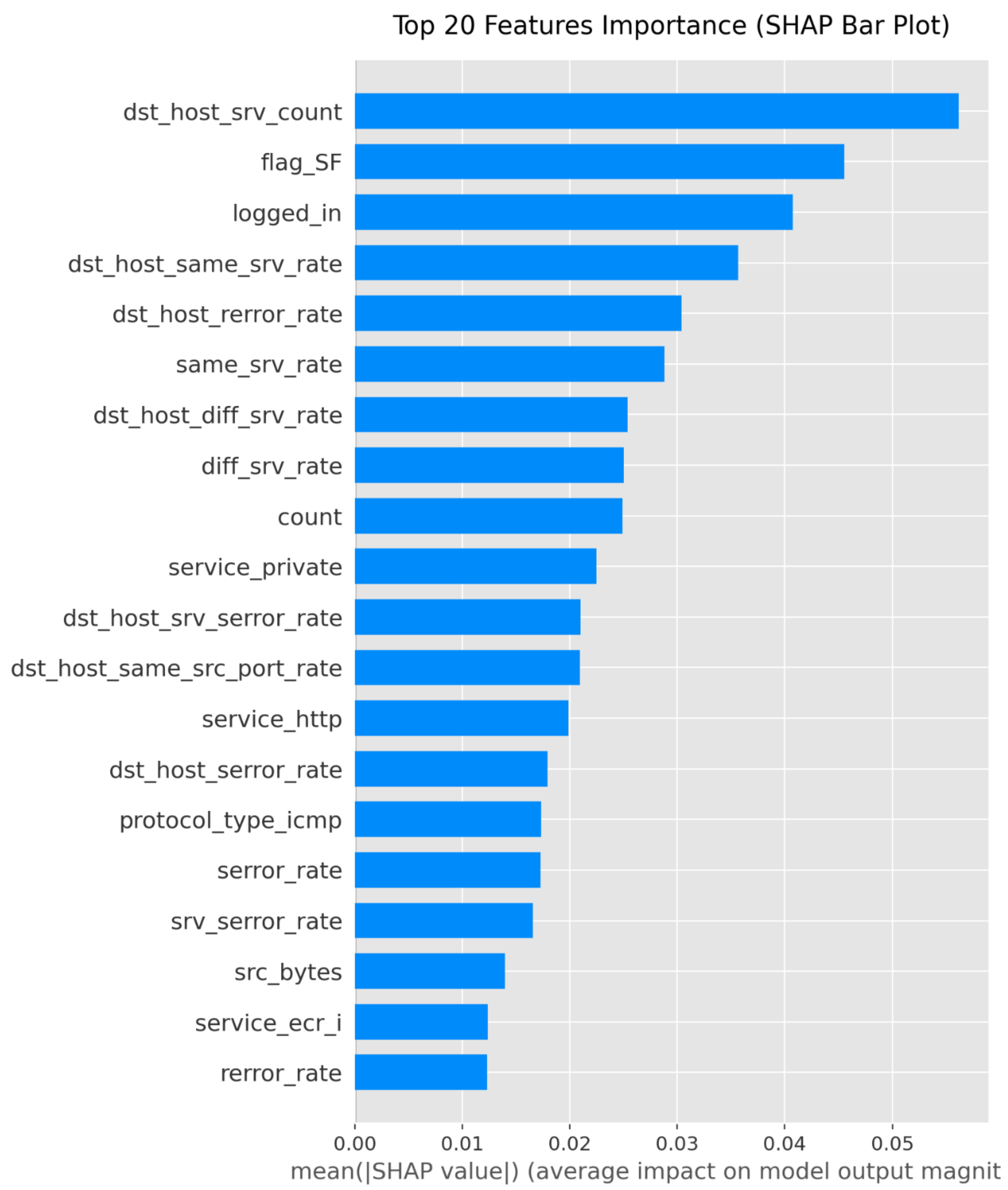
The choice of SHAP explainer depends on the model type:

- **shap.TreeExplainer**: This is an optimized explainer specifically designed for tree-based models (like Random Forest, LightGBM, XGBoost). It's significantly faster because it leverages the tree structure to compute exact Shapley values or highly accurate approximations efficiently.
- **shap.KernelExplainer**: This is a model-agnostic explainer, meaning it can explain any machine learning model. It works by treating the model as a "black box" and approximating Shapley values through a local linear approximation. It is generally slower than TreeExplainer and requires a background dataset for its calculations. The project's code includes a fallback to KernelExplainer if TreeExplainer fails, using a small sample of X_test as the background.

4.5.3. SHAP Visualization Plots

SHAP provides several plot types to visualize feature contributions globally and locally.

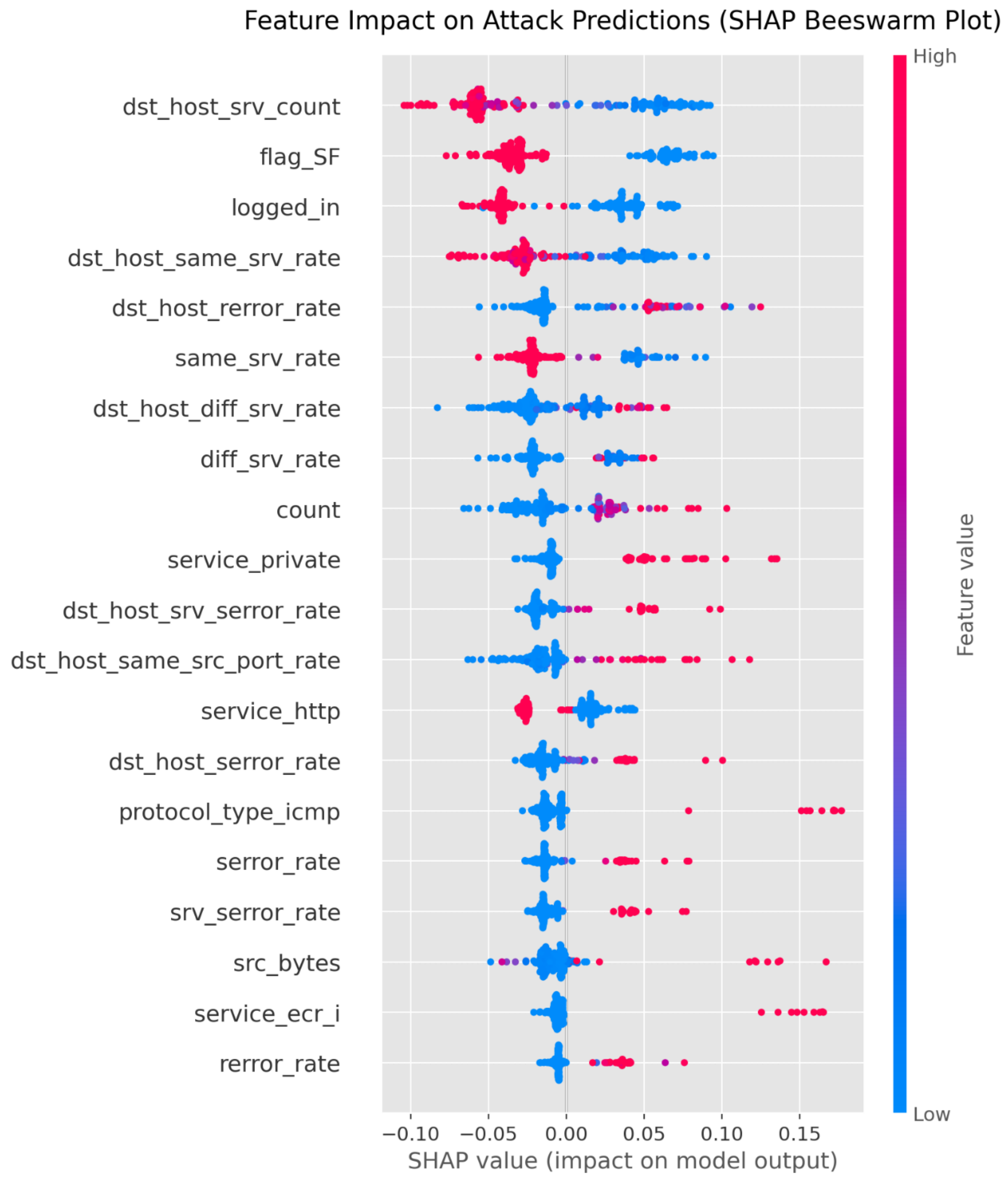
Global Feature Importance (SHAP Bar Plot)



Explanation:

- This plot summarizes the overall importance of features across the entire dataset.
- Each bar represents a feature, and its length indicates the average absolute SHAP value for that feature. A longer bar means the feature has a greater average impact on the model's output (prediction magnitude).
- It provides a global understanding of which features are most influential in predicting network anomalies. For example, src_bytes (source bytes) might be highly influential, indicating that the volume of data sent from the source is a strong indicator of an attack.

Detailed Feature Impact (SHAP Beeswarm Plot)

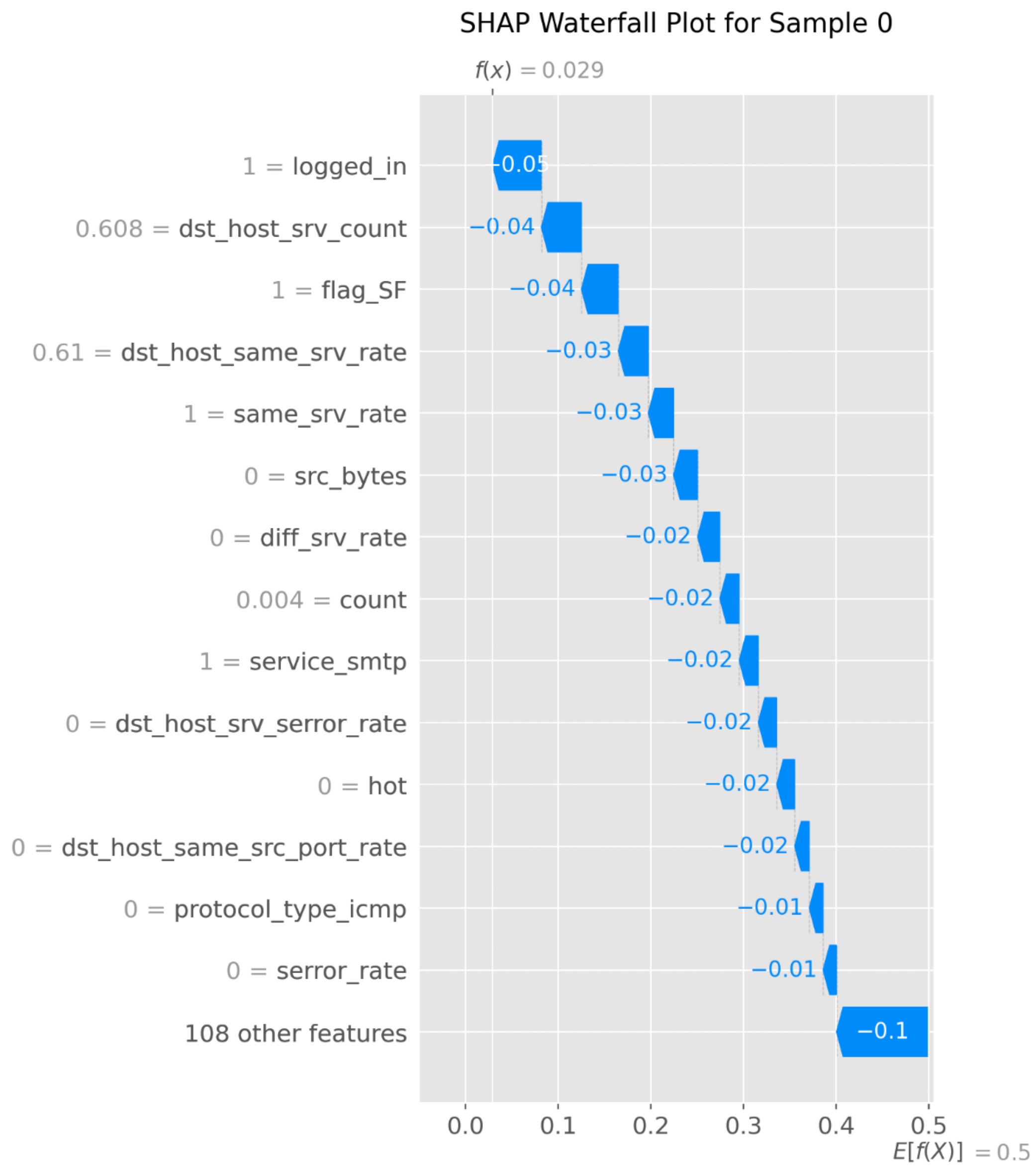


Explanation:

- The Beeswarm plot provides a more detailed view of how each feature affects the model's predictions and shows the distribution of SHAP values for each feature.
- Each dot on the plot represents a single data instance's SHAP value for that feature.
- **Color (Red to Blue):** Represents the feature value for that instance. Red indicates a high feature value, and blue indicates a low feature value.
- **Horizontal Position:** Indicates the SHAP value. Dots to the right push the prediction higher (towards 'attack'), while dots to the left push it lower (towards 'normal').

- This plot allows us to see relationships like: "When duration (connection duration) is high (red dots), it tends to push the prediction towards 'normal' (left side of the plot), suggesting short durations are more indicative of attacks." Or, "When src_bytes (source bytes) is high (red dots), it pushes the prediction towards 'attack' (right side), indicating large data transfers are often malicious."

Individual Prediction Explanation (SHAP Waterfall Plot)



Explanation:

- The Waterfall plot explains a *single prediction* from the model.
- It starts with the expected_value (also known as the base value or the average model output over the dataset) and then shows how each feature's SHAP value adds to or subtracts from this base value to arrive at the final model output for that specific instance.
- **Base Value (E[f(x)]):** The starting point, representing the average prediction for the dataset.
- **Feature Contributions:** Each bar represents a feature.

    - Green bars push the prediction higher (e.g., towards 'attack').
    - Red bars push the prediction lower (e.g., towards 'normal').
- The length of the bar indicates the magnitude of the feature's contribution.
- **f(x):** The final prediction of the model for that specific instance.
- This plot is invaluable for local interpretability, allowing an analyst to understand why a *particular* network connection was classified as an anomaly, by highlighting the specific features that contributed most to that decision. For example, a high src_bytes combined with a low dst_host_diff_srv_rate might be the key factors for a specific attack prediction.

4.6. Artifact Saving

After the model is trained and analyzed, critical components are saved using Python's pickle module. This allows for easy loading and deployment of the model and its associated preprocessing steps without retraining.

```
with open('anomaly_detector.pkl', 'wb') as f:
pickle.dump(model, f)
with open('scaler.pkl', 'wb') as f:
pickle.dump(scaler, f)
with open('features.pkl', 'wb') as f:
pickle.dump(feature_columns, f)
with open('label_encoder.pkl', 'wb') as f:
pickle.dump(label_encoder, f)
with open('model_metrics.pkl', 'wb') as f:
pickle.dump(metrics, f)
if explainer:
with open('shap_explainer.pkl', 'wb') as f:
pickle.dump(explainer, f)
```

Behind the scenes of pickle.dump:

- pickle.dump(object, file_object, protocol): This function serializes a Python object (e.g., the trained RandomForestClassifier model, MinMaxScaler object, or SHAP Explainer) into a byte stream.
- 'wb' mode: Opens the file in "write binary" mode, which is necessary for pickling.
- Serialization: Pickle converts the Python object's structure and data into a format that can be stored on disk. This includes the model's learned weights, decision tree structures, scaler's min/max values, label encoder's mappings, and the SHAP explainer's internal state.
- This allows the saved .pkl files to be loaded later using pickle.load() to reconstruct the objects in memory, enabling the model to be used for predictions in the dashboard without needing to retrain or refit preprocessing steps.

5. Dashboard Implementation (Streamlit)

A web-based dashboard is developed using **Streamlit** to provide a user-friendly interface for real-time anomaly detection and explanation.

5.1. Dashboard Features

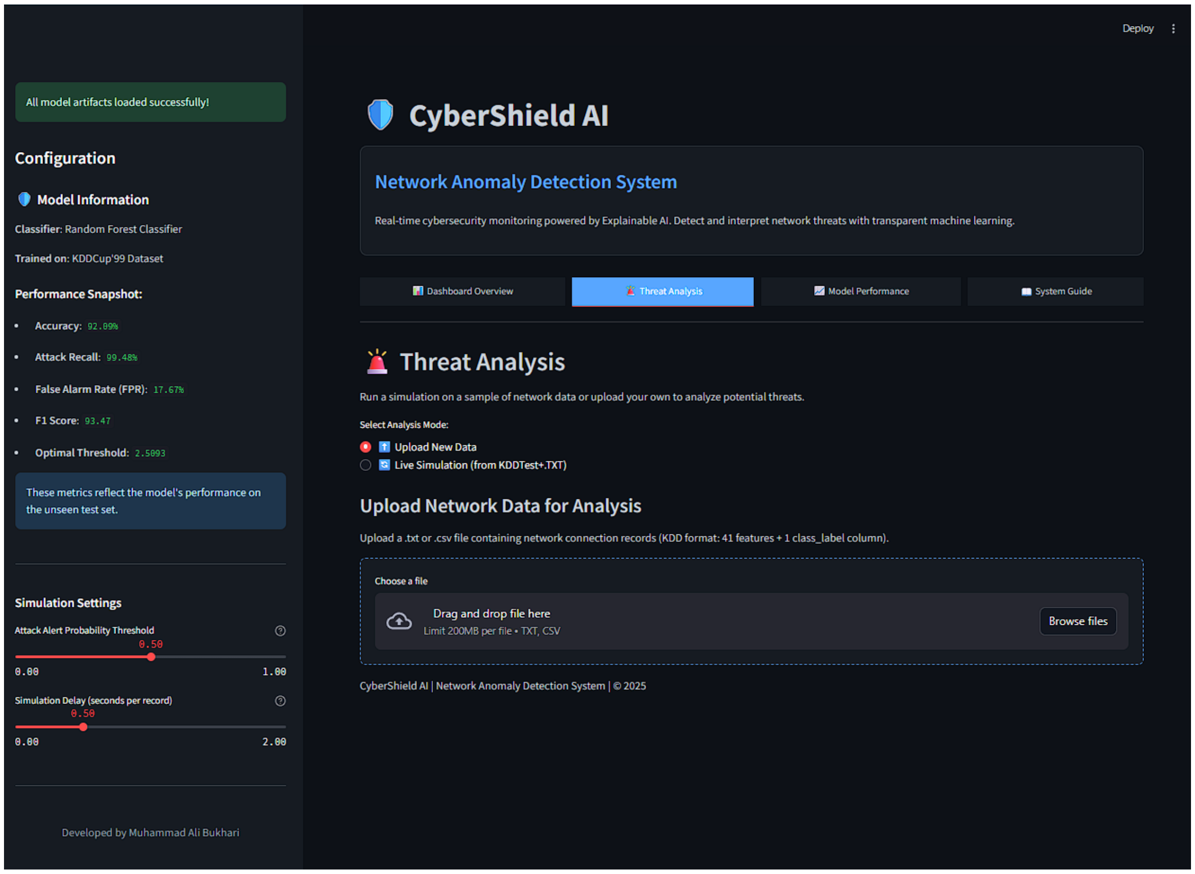
The dashboard provides the following key functionalities:

- **File Upload:** Allows users to upload a CSV file containing network traffic data for analysis.
- **Data Preprocessing Display:** Shows a preview of the uploaded data after preprocessing.

- **Real-time Prediction:** Uses the loaded machine learning model to classify network connections as 'normal' or 'attack'.
- **Interactive Controls:** Users can adjust parameters like the number of rows to display or interact with data.
- **Performance Metrics Display:** Presents the overall model performance metrics (Accuracy, Precision, Recall, F1-Score, FPR).
- **SHAP Explanations:**
    - **Global Feature Importance:** Displays a global view of feature importance using SHAP summary plots.
    - **Local Explanations:** Provides an interactive way to select individual data points and visualize their SHAP waterfall plots, explaining *why* a particular prediction was made.
- **Threat Summary:** Provides statistics on detected anomalies and their protocol/service distribution.

5.2. Real-time Prediction and Explanation

The dashboard integrates the trained model and SHAP explainer to provide real-time insights. When a user uploads a new dataset or input, the data flows through the preprocessing steps (scaling, encoding) using the pre-saved scaler and label_encoder. The anomaly_detector.pkl model then makes predictions. For each prediction, the shap_explainer.pkl is used to generate SHAP values, which are then visualized, providing transparent explanations for the classification.


All model artifacts loaded successfully!

Configuration

Model Information

Classifier: Random Forest Classifier

Trained on: KDDCup'99 Dataset

Performance Snapshot:

- Accuracy: 92.09%
- Attack Recall: 99.48%
- False Alarm Rate (FPR): 17.67%
- F1 Score: 93.47
- Optimal Threshold: 2.5093

These metrics reflect the model's performance on the unseen test set.

Simulation Settings

Attack Alert Probability Threshold: 0.50 (0.00 – 1.00)

Simulation Delay (seconds per record): 0.50 (0.00 – 2.00)

Developed by Muhammad Ali Bukhari

CyberShield AI

Network Anomaly Detection System

Real-time cybersecurity monitoring powered by Explainable AI. Detect and interpret network threats with transparent machine learning.

Threat Analysis

Run a simulation on a sample of network data or upload your own to analyze potential threats.

Select Analysis Mode:

[✓ Upload New Data] Live Simulation (from KDDTest+.TXT)

Upload Network Data for Analysis

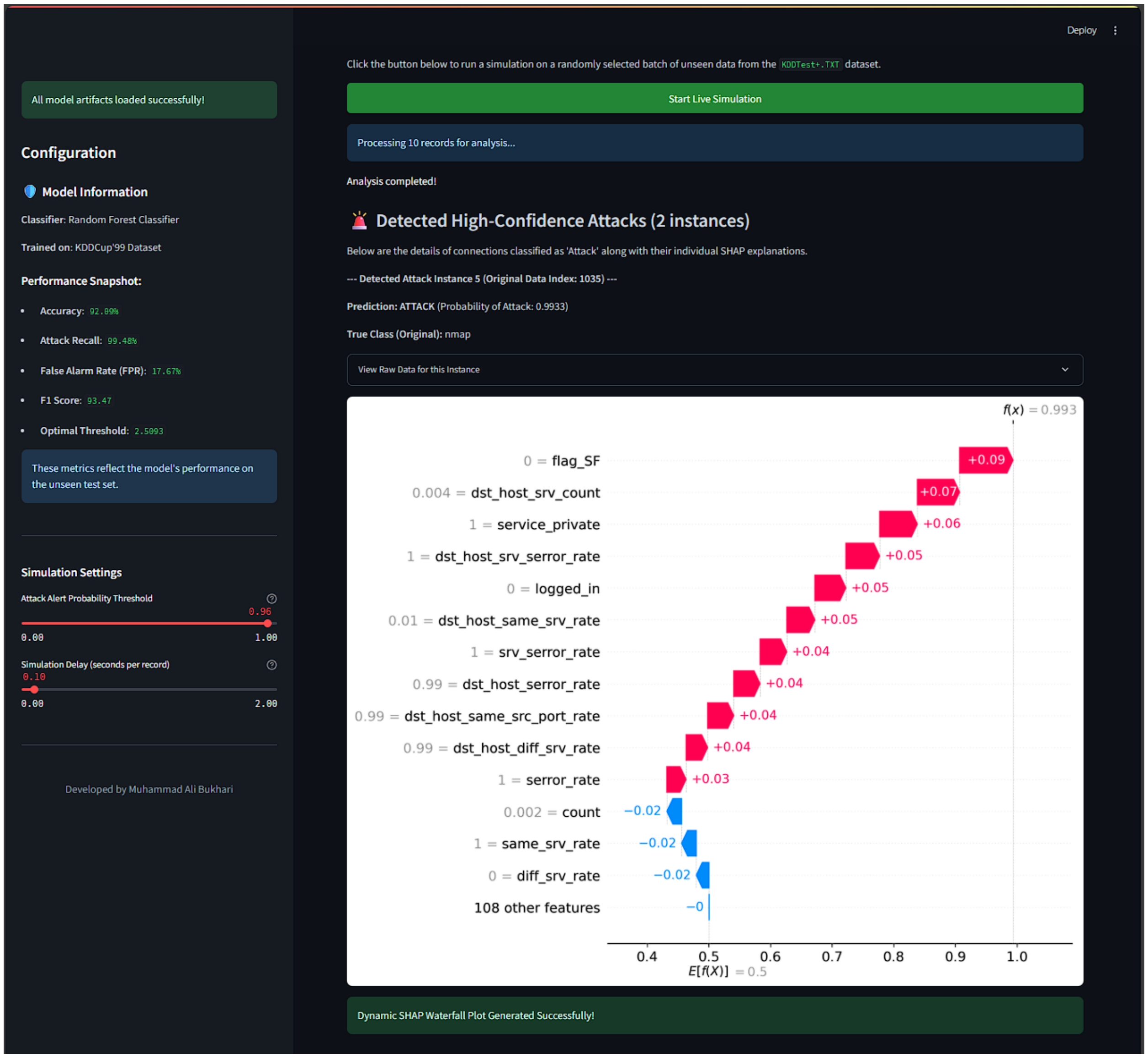
Upload a .txt or .csv file containing network connection records (KDD format: 41 features + 1 class_label column).

Choose a file

Drag and drop file here
Limit 200MB per file • TXT, CSV

CyberShield AI | Network Anomaly Detection System | © 2025

This screenshot illustrates the main interactive area of the "Advanced Network Intrusion Detection Dashboard." It showcases the file upload functionality, the section for displaying processed data, and a summary of recent detections. The user can upload a .csv file, triggering the backend pipeline to preprocess the data, run predictions, and generate explanations. This view would prominently feature the "Anomalies Detected" count and potentially a breakdown by protocol or service, giving an immediate overview of the network's health.



All model artifacts loaded successfully!

Configuration

Model Information

Classifier: Random Forest Classifier

Trained on: KDDCup'99 Dataset

Performance Snapshot:

- **Accuracy:** `92.09%`
- **Attack Recall:** `99.48%`
- **False Alarm Rate (FPR):** `17.67%`
- **F1 Score:** `93.47`
- **Optimal Threshold:** `2.5093`

These metrics reflect the model's performance on the unseen test set.

Simulation Settings

Attack Alert Probability Threshold 0.96 (0.00 – 1.00)

Simulation Delay (seconds per record) 0.10 (0.00 – 2.00)

Developed by Muhammad Ali Bukhari

Click the button below to run a simulation on a randomly selected batch of unseen data from the `KDDTest+.TXT` dataset.

Processing 10 records for analysis...

Analysis completed!

Detected High-Confidence Attacks (2 instances)

Below are the details of connections classified as 'Attack' along with their individual SHAP explanations.

--- Detected Attack Instance 5 (Original Data Index: 1035) ---

Prediction: ATTACK (Probability of Attack: 0.9933)

True Class (Original): nmap

Dynamic SHAP Waterfall Plot Generated Successfully!

This screenshot highlights the Explainable AI (XAI) capabilities of the dashboard, focusing on an individual prediction. It displays a SHAP waterfall plot for a selected network connection. This plot visually demonstrates how each feature (e.g., src_bytes, duration, service_http) contributes to the final prediction (e.g., "Attack"). Green bars indicate features pushing the prediction towards "Attack," while red bars push it towards "Normal." This interactive explanation helps security analysts understand the underlying reasons for a specific anomaly flag, enabling faster and more informed decision-making.

6. Results and Visualizations

The model's performance was evaluated extensively, and its interpretability was enhanced through SHAP visualizations.

6.1. Performance Curves
As shown in section 4.4.3, the **ROC and Precision-Recall curves** demonstrate the model's strong discriminatory power and effectiveness in handling class imbalance.

- **ROC AUC:** Approximately 0.96.
- **Average Precision (PR-AUC):** Approximately 0.96.

The detailed evaluation metrics are:

- **Optimal Threshold:** 2.5093% (probability value of 0.025093)
- **Accuracy:** 92.0910%
- **Precision:** 88.1508%
- **Recall:** 99.4779%
- **F1 Score:** 93.4725%
- **Average Precision (PR-AUC):** 96.8255%
- **False Alarm Rate (FPR):** 17.6707%

The confusion matrix on the test set is:

```
[[ 7995  1716]
 [   67 12766]]
```

Where:

- True Negatives (TN): 7995 (Correctly identified normal connections)
- False Positives (FP): 1716 (Normal connections incorrectly identified as attack - False Alarms)
- False Negatives (FN): 67 (Attack connections incorrectly identified as normal - Missed Detections)
- True Positives (TP): 12766 (Correctly identified attack connections)

The high recall (99.47%) indicates that the model is very effective at catching actual attacks, which is critical in cybersecurity. The precision of 88.15% means that when the model flags an attack, it is correct most of the time. The F1-score of 93.47% represents a good balance between precision and recall. While the FPR (False Alarm Rate) of 17.67% indicates some false alarms, this is a common trade-off in highly sensitive anomaly detection systems, especially when recall is prioritized.

6.2. SHAP Feature Importance (Bar Plot)
This plot provides a global understanding of the most influential features. For example, src_bytes (source bytes), service_http (HTTP service usage), count (number of connections to the same destination host as the current connection in the past two seconds), and dst_bytes (destination bytes) are often among the top features influencing the anomaly detection.

6.3. SHAP Detailed Feature Impact (Beeswarm Plot)
This plot offers deeper insights into how feature values impact predictions. For instance, high values of src_bytes (red dots) typically push predictions towards 'attack' (positive SHAP values), while high values of duration (red dots) might push predictions towards 'normal' (negative SHAP values), indicating that very long connections are less likely to be attacks.

6.4. SHAP Individual Prediction Explanation (Waterfall Plot)

This plot is crucial for understanding individual anomaly classifications. It visually breaks down how each feature contributes to a specific instance's prediction. For example, if a connection is flagged as an attack, the waterfall plot can show that abnormally high src_bytes and unusual flag_SF (normal completion flag) were the primary drivers for that specific detection, relative to the average prediction.

7. Conclusion

The project successfully developed an explainable AI-powered network anomaly detection system. By employing a Random Forest classifier combined with ADASYN for imbalance handling, the system achieves high performance metrics, particularly in recall and F1-score, which are critical for identifying cyber-attacks. The integration of SHAP values provides much-needed transparency, allowing cybersecurity analysts to understand the rationale behind each anomaly detection. The Streamlit dashboard offers an intuitive interface for real-time monitoring and interactive exploration of explanations, significantly improving the actionable intelligence derived from the system. This project demonstrates a robust solution for enhancing cybersecurity defenses through effective and interpretable machine learning.

8. Future Enhancements

- **Zero-Shot Learning Integration:** Explore methods for detecting novel, unseen attack types without explicit prior training, building on the explainability aspects.
- **Multi-Class Classification:** Expand the model to classify specific attack types rather than just binary (Normal/Attack).
- **Real-time Data Streaming:** Integrate with actual network traffic capture tools for true real-time analysis.
- **Alerting Mechanism:** Develop a robust alerting system for detected anomalies.
- **User Feedback Loop:** Allow analysts to provide feedback on predictions to continuously improve model performance.
- **More Advanced Models:** Experiment with deep learning models (e.g., LSTMs for sequential data) for potentially higher accuracy on complex attack patterns.
- **Hardware Acceleration:** Utilize GPUs for faster training and inference if processing large volumes of data.

9. Frequently Asked Questions (FAQs)

Easy Questions:

Q1: What is the main goal of this project? A1: The main goal is to detect unusual activities (anomalies) in network traffic, especially cyber-attacks, and to explain why those detections were made using AI.

Q2: What kind of data does this system analyze? A2: It analyzes network connection data from the NSL-KDD dataset, which contains information about various network sessions and their characteristics.

Q3: How does the system tell if something is an "attack" or "normal"? A3: It uses a machine learning model called Random Forest, which is trained on many examples of normal and attack network traffic to learn patterns and make predictions.

Q4: What is the dashboard for? A4: The dashboard provides an easy-to-use interface where you can upload network data, see predictions in real-time, and get explanations for why the system flagged something as an anomaly.

Q5: What does "Explainable AI (XAI)" mean in this project? A5: It means that the system doesn't just tell you "this is an attack," but also explains *which features* (like data volume or connection duration) were most important in deciding it was an attack. This helps humans understand and trust the AI's decisions.

Hard Questions:

Q1: Why was the NSL-KDD dataset chosen over the original KDD'99 dataset, and what challenges does it specifically address? A1: NSL-KDD was chosen because it addresses crucial issues present in the original KDD'99 dataset, namely redundant records in the training set and duplicate records in the testing set. These redundancies can bias model evaluation, leading to an overestimation of performance and preventing models from learning robustly. NSL-KDD mitigates these problems, providing a more realistic benchmark for intrusion detection systems.

Q2: How does ADASYN specifically help with class imbalance, and why is it preferred over simpler oversampling methods like SMOTE for this project? A2: ADASYN (Adaptive Synthetic Sampling) generates synthetic samples for the minority class, similar to SMOTE. However, ADASYN adaptively shifts the decision boundary by focusing on minority instances that are harder to learn (i.e., those near the decision boundary or surrounded by majority class samples). It generates more synthetic samples for these "difficult" examples, creating a more effective separation between classes. This is a key advantage over SMOTE, which generates synthetic samples without considering the local density of minority samples, potentially leading to overlap or noise.

Q3: Explain the role of the optimal threshold selection in the model evaluation phase. Why is it important, especially in an anomaly detection context, and how is it determined? A3: An optimal threshold is crucial in anomaly detection because the balance between precision (minimizing false alarms) and recall (minimizing missed attacks) is critical. A default 0.5 threshold on predicted probabilities might not be ideal for imbalanced datasets. The optimal threshold is determined by maximizing the F1-score from the Precision-Recall curve. The F1-score is the harmonic mean of precision and recall, providing a balanced metric. By selecting the threshold that yields the highest F1-score, the system aims to achieve the best possible trade-off between accurately identifying attacks and avoiding excessive false alarms, which is essential to prevent alert fatigue in security operations.

Q4: Deep dive into how SHAP values are calculated for a tree-based model like Random Forest, and how shap.TreeExplainer optimizes this process. How does this differ conceptually from shap.KernelExplainer? A4: For tree-based models like Random Forest, shap.TreeExplainer computes exact (or highly accurate approximations of) Shapley values by traversing the tree structure. It efficiently aggregates feature contributions across the paths of individual trees and then across all trees in the ensemble. This is significantly faster than model-agnostic methods because it leverages the underlying decision logic of the trees. Conceptually, shap.TreeExplainer is "white-box" as it understands the internal workings of the tree model. In contrast, shap.KernelExplainer is "black-box" or model-agnostic. It approximates Shapley values by observing how the model's output changes when features are perturbed. It samples many permutations of feature subsets (coalitions), feeds them to the model, and then uses a weighted linear regression to estimate the contribution of each feature. This process is computationally more intensive and requires a background dataset to define the feature space, which is why TreeExplainer is preferred for efficiency when applicable.

Q5: The project lists "Zero-Shot Learning" as a future enhancement. How could the current architecture, particularly the SHAP analysis component, be leveraged to support zero-shot anomaly detection for novel attack types? A5: The SHAP analysis provides a strong foundation for integrating zero-shot learning. While the current model is trained on known attacks, SHAP helps us understand the *characteristics* that lead to an "attack" prediction. For zero-shot learning:

- **Feature-based Anomaly Definition:** By analyzing SHAP values across various known attack types, we can identify a "signature" of feature contributions that define an "attack." For example, consistently high SHAP values for src_bytes and dst_bytes might indicate data exfiltration.
- **Similarity to Known Attack Signatures:** When a *novel* anomaly occurs, even if the model's overall prediction is uncertain or slightly off, its SHAP explanation might reveal feature contributions that are highly similar to known attack signatures (even if the specific attack type is new).
- **Proactive Rule Generation:** The insights from SHAP can be used to manually or semi-automatically create "rules" or "patterns" for previously unseen attack types based on their expected feature impact. For instance, if a new type of attack is described, its characteristics can be mapped to expected SHAP values, allowing a form of detection without direct training.
- **Clustering SHAP Explanations:** Clustering similar SHAP explanations for unclassified anomalies could help identify common underlying attack patterns, even if the specific attack type is unknown. This can lead to new "zero-shot" categories. The explainability provided by SHAP allows for a shift from purely pattern-matching to understanding the underlying *causal features* of an anomaly, which is crucial for generalizing to unseen attack scenarios.