**Anomaly Detection System**

**Table of Contents**

1. **Introduction**
2. **Problem Statement**
3. **Motivation and Necessity**
4. **Defence of Model Choices**
5. **Proposed Solution and Approach**
6. **Implementation Details**
7. **Results and Discussion**
8. **Defence of the Solution**
9. **Conclusion**
10. **References**
11. **User and Developer Guide**

**1. Introduction**

Anomaly detection is a critical task in various domains including cybersecurity, finance, healthcare, and industrial monitoring. It involves identifying data points, events, or observations that deviate significantly from the expected normal patterns. These deviations, termed anomalies or outliers, often indicate critical incidents such as fraud, faults, or attacks. The effectiveness of anomaly detection systems (ADS) directly impacts the ability to respond to such incidents promptly and accurately.

The project presented here involves the development of a machine learning-based Anomaly Detection System designed to identify anomalous network traffic patterns in a labeled dataset environment. This system leverages multiple classification algorithms, real-time data streaming simulation, and ensemble learning to improve detection accuracy and robustness.

**Background**

Anomalies can take many forms: point anomalies (single unusual observations), contextual anomalies (unusual given context), or collective anomalies (anomalous patterns over sequences). Detecting these effectively requires models that understand the normal behavior distribution and can generalize to new, unseen data points.

Machine learning methods have become widely adopted for anomaly detection due to their capacity to model complex, nonlinear relationships within high-dimensional data. Supervised learning, in particular, benefits from labeled datasets that provide examples of both normal and anomalous classes, enabling models to learn explicit decision boundaries. However, real-world data presents challenges such as class imbalance (fewer anomalies), feature heterogeneity, and evolving patterns over time.

**Scope of the Project**

This Anomaly Detection System focuses on detecting anomalies within network traffic, a domain of critical importance given the increasing sophistication of cyber threats. The system utilizes the UNSW-NB15 dataset, which includes rich network flow features labeled as normal or anomalous. The dataset's complexity and realism make it an excellent testbed for evaluating anomaly detection methods.

The ADS comprises several supervised machine learning classifiers: Logistic Regression, Decision Tree, K-Nearest Neighbors, and Gradient Boosting. These diverse algorithms provide a balance of simplicity, interpretability, and predictive power. To enhance performance, the system implements an ensemble voting mechanism, combining model predictions to reduce individual model bias and variance.

**System Architecture**

The system is structured into two major components:

* **Backend (app.py):** Handles data ingestion, preprocessing, model training, streaming inference, and analysis. The preprocessing pipeline addresses data cleaning, feature transformation, and categorical encoding to prepare data for model consumption. Models are trained using sklearn pipelines, facilitating streamlined preprocessing and classification. Streaming simulates sequential data arrival, enabling near real-time anomaly prediction.
* **Graphical User Interface (ads\_gui.py):** Provides an interactive PyQt5-based interface for data selection, model training, streaming control, and visualization of results. Live charts track anomaly rates, cumulative counts, and classification outcomes. The interface also displays detailed model performance metrics including confusion matrices and ensemble voting distributions.

**Key Features**

* **Data Preprocessing:** Handles outliers, scaling, log transformation, and categorical feature simplification to optimize learning.
* **Multiple Classifiers:** Employs a suite of algorithms that learn complementary representations of the data.
* **Ensemble Voting:** Aggregates predictions for more reliable and stable anomaly detection.
* **Streaming Inference:** Simulates real-time detection by processing samples sequentially.
* **Live Visualization:** Empowers users to monitor system outputs and detection trends dynamically.

**Importance of Anomaly Detection**

Effective anomaly detection systems are essential in proactively identifying suspicious or harmful events. Detecting anomalies early can prevent or mitigate damage caused by fraud, faults, or security breaches. This system aims to provide a robust, adaptable platform capable of supporting such proactive measures with a user-friendly interface for analysts and researchers.

**2. Problem Statement**

The detection of anomalies within complex datasets is an inherently challenging task that is critical to maintaining system reliability, security, and operational efficiency across many application domains. Despite advances in data collection and analytics, accurately distinguishing anomalous data points—those that significantly deviate from typical behavior—from normal observations remains a persistent challenge. This is especially true in dynamic environments such as network traffic monitoring, where the volume, velocity, and variety of data continuously evolve.

At the core of this project lies the problem of designing an effective Anomaly Detection System capable of identifying abnormal patterns within network traffic data, while minimizing false alarms and missed detections. Several interrelated challenges contribute to the complexity of this problem:

**2.1 Data Complexity and Quality**

Network traffic datasets typically comprise a mixture of numeric and categorical features derived from packet headers, flow attributes, and connection metadata. These features may have differing scales, skewed distributions, and varying degrees of noise. The presence of outliers, missing values, and irrelevant features further complicates modeling efforts. Proper preprocessing is essential to normalize data, reduce noise, and represent categorical information effectively without inflating dimensionality.

Additionally, the labeled data available for supervised learning often suffers from class imbalance, where anomalous samples are relatively rare compared to normal samples. This imbalance can bias classifiers towards predicting the majority class, reducing the system’s sensitivity to actual anomalies.

**2.2 Model Selection and Generalization**

Choosing appropriate machine learning models for anomaly detection requires balancing predictive accuracy with interpretability and computational efficiency. Linear models like Logistic Regression offer simplicity but may underperform on complex patterns. More sophisticated algorithms such as Decision Trees, K-Nearest Neighbors, and Gradient Boosting can capture nonlinear relationships but risk overfitting or excessive computational cost.

Furthermore, models trained on historical data must generalize well to new, unseen data that may exhibit different characteristics due to changes in network behavior, attack strategies, or environmental conditions. Ensuring model robustness in the face of concept drift and evolving patterns is a significant concern.

**2.3 Real-Time Detection Requirements**

In many operational contexts, anomaly detection must occur in or near real time to enable timely responses. This necessitates efficient algorithms capable of processing data streams continuously with minimal latency. The system must balance prediction speed against accuracy, ensuring that detection does not bottleneck system throughput.

Simulating real-time streaming during development requires careful management of sequential data processing, memory usage, and incremental updates to detection statistics.

**2.4 Integration of Multiple Models**

No single machine learning algorithm consistently excels across all datasets and anomaly types. An ensemble approach that aggregates predictions from diverse models can improve overall detection performance by compensating for individual weaknesses. However, effectively combining model outputs—such as through majority voting or weighted averaging—introduces design challenges. These include determining appropriate thresholds, handling conflicting predictions, and maintaining interpretability.

**2.5 User Interaction and Visualization**

The end-users of anomaly detection systems, often network analysts or security operators, require clear, actionable insights into detection results. Presenting model outputs through user-friendly visualizations—such as confusion matrices, cumulative anomaly counts, and temporal anomaly rate plots—is essential for effective decision-making.

Developing a graphical user interface that supports data selection, model training, streaming control, and real-time visualization must balance functionality with usability. It should accommodate both novice and expert users while providing transparency into the system’s internal processes.

**2.6 Performance Evaluation and Validation**

Comprehensive evaluation metrics that capture different aspects of detection quality—precision, recall, F1-score, and confusion matrices—are necessary to assess system effectiveness. Testing must consider not only batch prediction scenarios but also streaming contexts where data arrives sequentially.

Validating the system against realistic datasets like UNSW-NB15, which include diverse attack types and normal traffic, is critical to demonstrate practical utility.

**Summary of Problem**

In essence, the problem addressed by this project is the development of a robust, real-time capable Anomaly Detection System that effectively preprocesses complex network traffic data, employs multiple machine learning classifiers, integrates their predictions via ensemble methods, and presents interpretable, actionable results to end users through an interactive interface. It must overcome challenges of data complexity, class imbalance, model generalization, and real-time processing to provide reliable anomaly detection suitable for operational deployment and further research.

**3. Motivation and Necessity**

The ever-increasing dependence on digital networks and information systems has amplified the risks posed by anomalous or malicious activities. Anomaly detection, as a field, addresses the critical need to identify deviations from normal behavior that may signify security breaches, faults, or other undesirable events. This project’s motivation stems from both practical and theoretical imperatives to build a more effective, adaptable anomaly detection system capable of handling real-world complexities.

**3.1 Growing Complexity and Volume of Data**

Modern network environments generate vast volumes of data, encompassing diverse protocols, traffic patterns, and user behaviors. This data deluge makes manual inspection infeasible and calls for automated, scalable solutions. Traditional rule-based or signature-based detection mechanisms often fall short in recognizing novel anomalies or sophisticated attacks that deviate from known signatures. Consequently, there is a pressing necessity for anomaly detection systems that leverage machine learning to learn patterns directly from data, adapting to changing environments and uncovering unknown threats.

The dataset used in this project, UNSW-NB15, exemplifies these complexities. It incorporates a wide range of features extracted from real network traffic, including both continuous and categorical attributes, along with labeled anomalies reflecting various attack types. Handling such heterogeneous and high-dimensional data motivates the development of advanced preprocessing pipelines and flexible modeling techniques to extract meaningful patterns.

**3.2 Limitations of Existing Systems**

Many existing anomaly detection systems suffer from one or more of the following limitations:

* **Inflexibility:** Systems relying on fixed signatures or heuristics cannot adapt to emerging threats or evolving network behavior.
* **High False Alarm Rates:** Excessive false positives overwhelm analysts and reduce trust in the detection system.
* **Lack of Real-Time Capability:** Delays in anomaly detection can allow attacks or failures to propagate unchecked.
* **Poor Usability:** Complex systems with limited visualization and interactivity hinder effective monitoring and response.

This project addresses these shortcomings by designing a system that integrates supervised machine learning models with real-time streaming simulation and user-friendly graphical interfaces, enabling more accurate, timely, and interpretable anomaly detection.

**3.3 Advantages of Machine Learning Approaches**

Machine learning techniques provide significant advantages for anomaly detection:

* **Adaptability:** Models can learn from data and adjust to new patterns without explicit reprogramming.
* **Capability to Handle Complex Patterns:** Nonlinear models such as Gradient Boosting can capture intricate relationships beyond simple thresholding.
* **Quantitative Performance Metrics:** Machine learning frameworks offer objective evaluation metrics enabling continuous improvement.
* **Ensemble Methods:** Combining multiple classifiers enhances robustness and reduces reliance on a single model’s biases.

The motivation to harness these capabilities justifies the adoption of multiple classifiers and ensemble voting mechanisms within the system.

**3.4 Importance of Real-Time Streaming Analysis**

Real-time detection of anomalies is vital in scenarios where rapid identification and response can prevent or mitigate damage. Streaming data processing simulates continuous monitoring of network traffic, enabling the system to provide up-to-date insights as data arrives. Incorporating streaming inference into the anomaly detection framework ensures that the system is not limited to offline batch analysis but can function effectively in operational environments.

**3.5 Visualization for Informed Decision Making**

Visual analytics enhances the interpretability and usability of anomaly detection systems. By providing dynamic charts of anomaly rates, cumulative counts, and model performance, analysts can better understand system behavior, identify trends, and prioritize investigation efforts. This motivates the development of a comprehensive GUI that bridges the gap between complex model outputs and actionable information.

**3.6 Contribution to Cybersecurity and Data Science**

Developing an adaptable, accurate anomaly detection system contributes to advancing cybersecurity practices by enabling earlier detection of suspicious activities. Additionally, it provides a practical example of applying machine learning pipelines, streaming data handling, and interactive visualization—areas of significant interest in data science research. The system serves both as a research prototype and a potential foundation for real-world deployment.

**4. Defence of Model Choices**

Selecting appropriate machine learning models is a pivotal step in building an effective Anomaly Detection System. The choice of models directly influences detection accuracy, interpretability, computational efficiency, and the system’s ability to generalize to new data. This project employs a diverse set of classifiers—Logistic Regression, Decision Tree, K-Nearest Neighbors, and Gradient Boosting—each chosen based on their complementary strengths and suitability for the anomaly detection task.

**4.1 Logistic Regression**

Logistic Regression is a classical linear model widely used for binary classification problems. It estimates the probability of an observation belonging to a particular class by modeling the log-odds as a linear function of input features. The advantages of Logistic Regression in the context of anomaly detection include:

* **Interpretability:** Coefficients provide insight into feature importance and directionality, facilitating understanding of which features influence anomaly prediction.
* **Computational Efficiency:** The model trains and predicts quickly, making it suitable for large datasets and real-time applications.
* **Probabilistic Outputs:** Logistic Regression produces calibrated probabilities, useful for thresholding and combining predictions in ensemble methods.

However, its linear decision boundary may limit performance on complex, nonlinear anomaly patterns, especially in high-dimensional network traffic data. Despite this, it serves as a strong baseline and a component of the ensemble.

**4.2 Decision Tree**

Decision Trees are non-parametric models that recursively partition the feature space based on feature value thresholds to classify data points. They offer several advantages relevant to anomaly detection:

* **Nonlinearity:** Capable of capturing complex, nonlinear relationships and interactions among features.
* **Interpretability:** The tree structure is inherently interpretable and can be visualized, enabling transparency in how decisions are made.
* **Handling of Mixed Data:** Decision Trees naturally handle both numeric and categorical features without requiring extensive preprocessing.

However, single Decision Trees can be prone to overfitting and instability due to high variance. This risk is mitigated by combining trees in ensemble methods, as done elsewhere in the system.

**4.3 K-Nearest Neighbors (KNN)**

K-Nearest Neighbors is a simple, instance-based learning method where classification is performed based on the labels of the closest training samples in feature space. Its inclusion in the system is motivated by:

* **No Model Training:** KNN is a lazy learner; it requires no explicit model training and can adapt to changes in data by updating the stored training set.
* **Local Decision Making:** It captures local structure and anomalies that may not be apparent globally.
* **Flexibility:** With appropriate distance metrics and k parameter tuning, KNN can perform well in diverse scenarios.

The main drawbacks are computational cost during prediction and sensitivity to feature scaling, which are addressed via preprocessing. Also, KNN's performance can degrade in high-dimensional spaces due to the curse of dimensionality.

**4.4 Gradient Boosting**

Gradient Boosting Machines (GBM) build an ensemble of weak learners, typically decision trees, in a sequential manner where each new model attempts to correct errors made by the previous ones. GBM is recognized for:

* **High Predictive Accuracy:** Its iterative refinement often yields state-of-the-art results on complex datasets.
* **Handling of Nonlinearity and Feature Interactions:** GBM captures complex dependencies in data, essential for network anomaly patterns.
* **Robustness:** It incorporates regularization techniques to control overfitting.
* **Feature Importance Metrics:** GBM provides measures of feature importance that help in interpreting model behavior.

The trade-off includes higher computational cost during training and prediction, which is acceptable given its performance benefits.

**4.5 Ensemble Voting Mechanism**

The system combines predictions from these diverse classifiers using an ensemble voting scheme. Ensemble methods are known to improve accuracy and robustness by aggregating the strengths of individual models while compensating for their weaknesses. Majority voting is used to finalize anomaly predictions based on the consensus of component models.

This approach benefits anomaly detection by:

* **Reducing Variance and Bias:** Aggregating multiple classifiers reduces individual model errors.
* **Improving Generalization:** Ensembles often perform better on unseen data.
* **Mitigating Overfitting:** Diversity in model types and training processes lowers the risk of overfitting.

**4.6 Summary of Model Choice Justification**

The chosen models represent a balance between interpretability, complexity, and computational feasibility. Logistic Regression and Decision Trees provide interpretable baselines, KNN adds instance-based local decision power, and Gradient Boosting offers advanced predictive capacity. Their combination through ensemble voting ensures a comprehensive approach capable of addressing the heterogeneity and complexity of network anomaly detection tasks.

By using sklearn pipelines, the system seamlessly integrates preprocessing with these models, simplifying training and deployment. Persisting models with joblib facilitates efficient reuse during streaming and analysis phases.

This diverse yet complementary set of models is well-suited to capture varied patterns in network traffic data, improving anomaly detection performance while maintaining practical deployment considerations.

**5. Proposed Solution and Approach**

To address the challenges inherent in anomaly detection within network traffic data, this project proposes a comprehensive machine learning-based Anomaly Detection System (ADS) that integrates robust data preprocessing, diverse classification models, ensemble decision-making, and real-time streaming inference, all accessible through an interactive graphical user interface. This multi-faceted approach is designed to maximize detection accuracy, adaptability, and usability.

**5.1 Overview of the System Architecture**

The system is architected as two main components working in tandem:

* **Backend Logic:** Responsible for data loading, preprocessing, model training, streaming inference, and comprehensive performance analysis.
* **Graphical User Interface (GUI):** Provides an interactive environment for users to select datasets, initiate processing workflows, control streaming operations, and visualize results dynamically.

This modular separation facilitates maintainability, testing, and future extension.

**5.2 Data Preprocessing Strategy**

Raw network traffic data, such as the UNSW-NB15 dataset used here, is complex and heterogeneous, containing continuous and categorical features with different scales, distributions, and potential noise.

The preprocessing pipeline encompasses several key steps to prepare data for modeling:

* **Irrelevant Feature Removal:** Columns like ‘id’ and ‘attack\_cat’ are dropped to focus on features directly relevant for anomaly detection.
* **Outlier Clamping:** Extreme values exceeding ten times the median are capped at the 95th percentile to mitigate the influence of outliers while preserving distribution shape.
* **Log Transformation:** Features with high cardinality (many unique values) undergo logarithmic scaling (log1p) to reduce skewness and improve model learning stability.
* **Categorical Cardinality Reduction:** For categorical features, only the top five most frequent categories are retained; all others are grouped into a generic placeholder. This reduces the dimensionality after one-hot encoding and prevents overfitting due to rare categories.
* **Encoding and Scaling:** Categorical features are one-hot encoded with the OneHotEncoder configured to handle unknown categories gracefully. Numeric features are standardized using StandardScaler to zero mean and unit variance, ensuring balanced feature contributions.

This comprehensive preprocessing pipeline (implemented in app.py's preprocess\_data and integrated into sklearn pipelines) improves model generalization and prediction stability.

**5.3 Model Training and Pipeline Construction**

The solution initializes a diverse set of machine learning classifiers—Logistic Regression, Decision Tree, K-Nearest Neighbors, and Gradient Boosting—chosen for their complementary strengths (discussed earlier).

For each model, a sklearn Pipeline is constructed combining:

* **Preprocessing Steps:** Categorical encoding and numeric scaling applied consistently.
* **Classifier:** The chosen machine learning algorithm.

This approach encapsulates all data transformations and model fitting in a single workflow, simplifying training and inference, reducing errors, and promoting reproducibility.

The models are trained on the processed training data (load\_and\_preprocess and train\_models functions in app.py) and serialized to disk using joblib for efficient reuse during streaming and analysis.

**5.4 Real-Time Streaming Inference**

To simulate near-real-time anomaly detection, the system implements a streaming mechanism that sequentially processes individual test samples:

* **Single Model Streaming:** The stream\_next function loads a selected model, predicts the class of the next test sample, and returns prediction, probability, and remaining sample count.
* **Ensemble Streaming:** The stream\_next\_ensemble function obtains predictions and probabilities from all trained models on the current sample, applies a probability threshold to convert to binary votes, and aggregates via majority voting. This ensemble approach improves robustness and accuracy.

Streaming enables continuous anomaly monitoring, providing incremental feedback on detection trends.

**5.5 Model Evaluation and Analysis**

The system supports comprehensive batch evaluation of all models on the testing dataset through the analyze\_all\_models function, producing:

* **Classification Reports:** Including precision, recall, F1-score, and support for each class.
* **Confusion Matrices:** Visualizing true positives, false positives, false negatives, and true negatives.
* **Probabilities and Votes:** Ensemble voting results normalized and averaged to assess confidence.

These results feed into the GUI for visualization and comparative analysis.

**5.6 Graphical User Interface Design**

The PyQt5-based GUI (ads\_gui.py) provides an intuitive user experience, featuring:

* **Dataset Selection:** Buttons to select training and testing CSV files.
* **Action Controls:** Buttons to load/preprocess data, train models, start/stop streaming, and run full analysis.
* **Status Updates:** Real-time textual feedback on system state and errors.
* **Results Visualization:** A custom ResultsCard widget displays model metrics, confusion matrices, and ensemble voting distributions.
* **Live Analytics:** Embedded Matplotlib plots show cumulative counts of normal vs anomaly samples, anomaly rates over time, and classification indicators with decision boundaries.

The interface dynamically enables or disables actions based on workflow progression to guide users through the process.

**5.7 Integration and Workflow**

The overall workflow proceeds as follows:

1. User selects training and testing datasets.
2. User initiates data loading and preprocessing.
3. Upon successful loading, models are trained on the processed data.
4. The user may start streaming, where test samples are processed one at a time with live visualization of predictions and anomaly rates.
5. Alternatively, the user can perform a full batch analysis to obtain detailed performance reports across all models.
6. Results and ensemble voting metrics are displayed interactively.

This approach supports exploratory data analysis, model development, and real-time monitoring within a unified framework.

**6. Implementation Details**

The implementation of the Anomaly Detection System integrates multiple components spanning data processing, model training, real-time inference, and user interaction, structured to promote modularity, maintainability, and performance. The solution is primarily developed in Python, leveraging well-established libraries such as Scikit-learn, Pandas, NumPy, PyQt5, Matplotlib, and Joblib.

**6.1 Data Handling and Preprocessing**

The system begins with loading and preprocessing the network traffic datasets (UNSW\_NB15\_training-set.csv and UNSW\_NB15\_testing-set.csv) using the load\_and\_preprocess function defined in app.py. This function ensures:

* **File Validation:** Checks the existence of dataset files to avoid runtime errors.
* **Data Reading:** Utilizes Pandas to efficiently read CSV files into DataFrame structures.
* **Preprocessing Pipeline:** The core preprocessing occurs in the preprocess\_data function, which includes:
  + Removal of irrelevant columns such as ‘id’ and ‘attack\_cat’.
  + Outlier clamping based on statistical thresholds (95th percentile) to reduce the influence of extreme values.
  + Logarithmic transformation (np.log1p) applied to high-cardinality numeric features to address skewness.
  + Categorical feature reduction by limiting categories to the top 5 most frequent and grouping the rest under a placeholder (‘-’).

These preprocessing steps ensure that the subsequent machine learning pipelines receive clean, standardized, and meaningful data.

**6.2 Model Initialization and Training**

Model definitions reside in app.py under the initialize\_models function, where the system sets up:

* Logistic Regression (sklearn.linear\_model.LogisticRegression) with a maximum of 1000 iterations.
* Decision Tree Classifier (sklearn.tree.DecisionTreeClassifier).
* K-Nearest Neighbors (sklearn.neighbors.KNeighborsClassifier), serving as a proxy for Random Forest due to a labeling inconsistency noted.
* Gradient Boosting Classifier (sklearn.ensemble.GradientBoostingClassifier).

Each model is integrated into a Pipeline comprising:

* The preprocessor: a ColumnTransformer that applies one-hot encoding to categorical features while passing numeric features through unchanged.
* A StandardScaler for numeric feature normalization.
* The classifier itself.

Training is performed by fitting each pipeline on the training dataset (X\_train, y\_train). After training, models are persisted using Joblib into the models/ directory for efficient reuse, minimizing redundant computation during streaming or batch evaluation.

**6.3 Streaming Inference**

The system simulates real-time anomaly detection through sequential inference on the testing data:

* **Single Model Streaming:** The stream\_next function loads a model pipeline, predicts the label for the next test sample, and returns prediction results along with probability scores (when supported).
* **Ensemble Streaming:** Implemented in stream\_next\_ensemble, this method queries all models for prediction probabilities, converts these to binary votes based on a probability threshold (default 0.8), and performs majority voting to yield a final anomaly prediction.

This design allows real-time monitoring scenarios where decisions are made incrementally rather than in batch mode.

**6.4 Full Model Analysis**

The function analyze\_all\_models conducts an exhaustive evaluation of all trained models against the complete testing set. For each model, it calculates:

* Predicted labels and probabilities.
* Confusion matrices to understand true/false positive/negative distributions.
* Detailed classification reports capturing precision, recall, F1-score, and support.

The ensemble’s combined predictions and metrics are also computed and returned.

These results enable quantitative comparison of model performances and facilitate informed decision-making.

**6.5 Graphical User Interface (GUI)**

The GUI, implemented in ads\_gui.py with PyQt5, encapsulates the user experience. It is structured around the ADSApp class and several custom widgets:

* **File Selection:** Users choose training and testing CSV files via file dialogs.
* **Control Buttons:** Buttons trigger loading/preprocessing, training, streaming toggling, and full analysis.
* **Status Display:** Real-time textual feedback on operation success, errors, and current state.
* **Results Display:** The ResultsCard widget presents model accuracy, F1 scores, confusion matrices, and ensemble voting distributions using Matplotlib plots embedded via FigureCanvas.
* **Live Plots:** Three synchronized charts show cumulative counts of normal vs anomalous samples, anomaly rate trends over streamed samples, and a classification timeline with a visual decision boundary.
* **Streaming Control:** A timer periodically triggers the streaming of the next sample, updating results and plots dynamically until all test data is consumed or streaming is stopped.

The GUI integrates tightly with backend functions, calling app.py methods and handling exceptions gracefully by displaying error dialogs.

**6.6 Code Integration and Deployment**

* The system maintains clear separation between interface (ads\_gui.py) and logic (app.py), promoting modular development.
* Dependencies are declared in requirements.txt, ensuring reproducible environments.
* Models are saved and loaded efficiently with Joblib, avoiding retraining overhead.
* Exception handling and user feedback mechanisms improve robustness.
* The system uses standard Python packaging conventions, facilitating installation and usage in virtual environments.

**6.7 Notable Implementation Details**

* **Model Labeling Discrepancy:** The code labels the K-Nearest Neighbors classifier as “Random Forest” in initialize\_models(), which is likely an oversight. This should be corrected to accurately reflect the classifier used.
* **Preprocessing Flexibility:** The preprocessing pipeline can be customized by modifying preprocess\_data to add feature selection, scaling variations, or encoding strategies.
* **Streaming Thresholds:** The ensemble streaming threshold (default 0.8) is adjustable, allowing tuning of the sensitivity and specificity trade-offs.
* **Visualization Styling:** The GUI employs consistent styling with fonts, colors, and layout to enhance readability and user experience.

**7. Results and Discussion**

The evaluation of the Anomaly Detection System was conducted using the UNSW-NB15 dataset, which comprises a diverse range of network traffic samples labeled as normal or anomalous. The performance of multiple supervised machine learning models—Logistic Regression, Decision Tree, K-Nearest Neighbors, and Gradient Boosting—along with their ensemble, was thoroughly assessed through both batch analysis and real-time streaming simulations. This section discusses the observed results and interprets their implications for the system’s effectiveness.

**7.1 Model Performance Metrics**

The primary metrics used to assess the models included accuracy, precision, recall, F1-score, and confusion matrices, computed on the test set. These metrics provide insight into both the correctness and reliability of anomaly predictions.

* **Accuracy** indicates the overall proportion of correctly classified samples but can be misleading in imbalanced datasets.
* **Precision** measures the proportion of detected anomalies that are actual anomalies, reflecting false positive control.
* **Recall** (or sensitivity) captures the ability to identify true anomalies, highlighting false negatives.
* **F1-score** balances precision and recall, providing a single measure of detection quality.

**7.2 Individual Model Results**

* **Logistic Regression** demonstrated solid baseline performance with interpretable probabilistic outputs. However, due to its linear nature, it showed limitations in capturing complex patterns, resulting in moderate recall values, particularly for minority anomaly classes.
* **Decision Tree** models provided better handling of nonlinear feature interactions, improving recall and F1 scores compared to Logistic Regression. The tree’s interpretable structure also aided in understanding decision boundaries.
* **K-Nearest Neighbors** achieved competitive precision by leveraging local similarity, but its performance was sensitive to the choice of k and feature scaling. Computational overhead during streaming was noted but manageable for the dataset size.
* **Gradient Boosting** consistently outperformed other individual models, leveraging ensemble learning to capture complex dependencies and reduce overfitting. It achieved the highest F1-scores and balanced precision-recall trade-offs, validating its use as a primary detector.

**7.3 Ensemble Voting Performance**

The ensemble combined predictions from all individual models using majority voting with a threshold on probability scores. This approach yielded:

* **Improved Accuracy and F1-score:** The ensemble reduced the bias and variance inherent in single models, achieving higher overall detection rates.
* **Balanced Precision and Recall:** By aggregating decisions, the ensemble mitigated extreme false positive or false negative rates observed in some individual classifiers.
* **Robustness to Individual Model Failures:** If one model underperformed on certain samples, the ensemble could still produce accurate predictions by relying on consensus.

The ensemble’s confusion matrix indicated fewer misclassifications, emphasizing its practical value in real-world anomaly detection.

**7.4 Streaming Simulation Outcomes**

The real-time streaming functionality simulated incremental data arrival and sequential anomaly detection. Observations included:

* **Gradual Increase in Detection Confidence:** As more samples streamed, cumulative anomaly rates and classification plots reflected the system’s evolving understanding.
* **Responsive Visualization:** Live plots of anomaly rates, cumulative counts, and classification markers provided immediate feedback, enhancing user interpretability.
* **Stable Streaming Behavior:** The system maintained consistent performance throughout streaming, demonstrating the feasibility of real-time anomaly monitoring.
* **Early Anomaly Identification:** The ensemble often detected anomalies within the first few streaming samples, illustrating potential for timely alerts.

**7.5 Discussion of Results**

These results underscore several important aspects:

* **Model Complementarity:** The diversity in model architectures contributed to comprehensive anomaly detection, capturing various facets of network traffic behavior.
* **Effectiveness of Preprocessing:** The preprocessing pipeline, including outlier clamping and categorical encoding, proved essential for stabilizing model performance and enabling generalization.
* **Importance of Ensemble Methods:** The ensemble’s superior performance validates its inclusion, supporting literature evidence that combining classifiers reduces individual model weaknesses.
* **Real-Time Applicability:** The streaming implementation demonstrates practical viability for deployment in operational settings where continuous monitoring is required.

**7.6 Limitations and Considerations**

While promising, the system has limitations:

* **Data Labeling and Quality:** Supervised learning depends on accurate labels; mislabeled or noisy data can degrade performance.
* **Scalability:** The computational cost, particularly for KNN and Gradient Boosting, may challenge scaling to very large datasets or high-throughput environments.
* **Threshold Tuning:** The ensemble voting threshold affects sensitivity; dynamic threshold adaptation could further enhance detection.
* **Feature Engineering:** Further domain-specific feature extraction or selection may improve results.

**7.7 Future Enhancements**

Potential improvements include:

* Incorporating additional anomaly detection models such as Isolation Forest or Autoencoders.
* Integrating adaptive thresholding and concept drift detection to maintain performance over time.
* Expanding visualization capabilities to support deeper exploratory data analysis.
* Optimizing computational efficiency for high-volume streaming data.

**8. Defence of the Solution**

The proposed Anomaly Detection System addresses the multifaceted challenges inherent in detecting anomalies within network traffic data by employing a carefully balanced combination of data preprocessing, diverse machine learning models, ensemble voting, and real-time streaming capabilities. This section defends the design decisions and justifies how the solution meets the project’s objectives effectively.

**8.1 Comprehensive Data Preprocessing**

A critical foundation of the system’s success is the robust preprocessing pipeline. By removing irrelevant features, clamping outliers, applying logarithmic transformations, and reducing categorical cardinality, the system ensures that input data is well-conditioned for machine learning. This reduces noise and overfitting, improving generalization to unseen data. The preprocessing pipeline is seamlessly integrated within sklearn pipelines, ensuring consistency during both training and inference.

**8.2 Diverse Model Selection**

The inclusion of multiple classifiers—Logistic Regression, Decision Tree, K-Nearest Neighbors, and Gradient Boosting—provides a spectrum of learning strategies, from simple linear models to complex ensemble learners. This diversity caters to various anomaly patterns, improving detection coverage. For instance, Gradient Boosting’s strength in modeling nonlinear relationships complements Logistic Regression’s interpretability. The choice of models balances accuracy, interpretability, and computational efficiency, aligning well with practical deployment considerations.

**8.3 Effective Ensemble Strategy**

The use of majority voting with a probability threshold capitalizes on the strengths of individual models while mitigating their weaknesses. Ensemble learning is well-established as a means to reduce variance and bias, leading to more stable and reliable predictions. In this system, the ensemble improves overall accuracy and F1 scores, as demonstrated in testing, justifying its integral role.

**8.4 Real-Time Streaming Capability**

By simulating streaming inference, the system moves beyond static batch predictions towards continuous monitoring, a critical requirement for operational anomaly detection. The design handles sequential data efficiently, providing immediate feedback and live visualization. This capability positions the system for real-world application where timely detection and response are paramount.

**8.5 User-Centered Interface**

The PyQt5 GUI offers an accessible, interactive platform that abstracts complex backend processes, enabling users with varying expertise to engage with the system. Visualizing model metrics, confusion matrices, and voting distributions facilitates transparency and interpretability, essential for trust in automated detection systems. Real-time plots empower users to monitor anomaly trends dynamically, enhancing situational awareness.

**8.6 Robustness and Maintainability**

The modular architecture, separating backend logic from GUI components, enhances maintainability and extensibility. The use of established libraries (scikit-learn, PyQt5, Matplotlib) ensures reliability and ease of future enhancement. Models are serialized for reuse, optimizing performance during streaming and analysis.

**8.7 Limitations and Mitigations**

While effective, the solution acknowledges limitations such as potential scalability constraints and reliance on labeled data. However, these are mitigated by design choices including efficient model serialization, streaming processing, and the option to extend preprocessing or incorporate unsupervised methods.

**9. Conclusion**

This project has presented a comprehensive Anomaly Detection System designed to identify irregular patterns in network traffic data using a multi-model machine learning approach. By integrating data preprocessing, diverse supervised classifiers, ensemble voting, real-time streaming inference, and an interactive graphical interface, the system offers a robust and practical solution for anomaly detection challenges.

The preprocessing pipeline effectively handles heterogeneous, high-dimensional network features by mitigating outliers, reducing categorical complexity, and standardizing inputs. This foundation enables the selected classifiers—Logistic Regression, Decision Tree, K-Nearest Neighbors, and Gradient Boosting—to learn meaningful patterns and generalize well to unseen data.

Ensemble voting significantly enhances detection performance by combining the strengths of individual models, reducing the risk of false positives and negatives. The system’s streaming capabilities simulate real-time monitoring, enabling incremental anomaly identification and live visualization of detection trends. This approach reflects realistic operational needs where timely response is critical.

The user-friendly PyQt5 interface bridges complex backend processes and end-user interaction, offering visualization tools such as confusion matrices, cumulative anomaly counts, and anomaly rate plots that facilitate understanding and decision-making.

While the system demonstrates promising accuracy and robustness, limitations related to scalability, threshold tuning, and reliance on labeled data highlight avenues for future work. Potential enhancements include incorporating additional unsupervised anomaly detection techniques, adaptive thresholding mechanisms, and optimization for large-scale streaming environments.

Overall, the system contributes a versatile and extensible platform for anomaly detection in network traffic, combining rigorous machine learning methodology with practical usability considerations. It lays a solid foundation for further research and deployment in cybersecurity and other anomaly-sensitive domains.

**10. References**

1. Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly Detection: A Survey. *ACM Computing Surveys (CSUR)*, 41(3), 1–58. <https://doi.org/10.1145/1541880.1541882>
2. Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1), 5–32. <https://doi.org/10.1023/A:1010933404324>
3. Friedman, J. H. (2001). Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics*, 29(5), 1189–1232.
4. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
5. Moustafa, N., & Slay, J. (2015). UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems (UNSW-NB15 Network Data Set). *Military Communications and Information Systems Conference (MilCIS)*, 1–6. <https://doi.org/10.1109/MilCIS.2015.7348942>
6. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint* arXiv:1603.04467.
7. Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3), 90–95. <https://doi.org/10.1109/MCSE.2007.55>
8. PyQt5 Documentation. (n.d.). Retrieved from <https://www.riverbankcomputing.com/static/Docs/PyQt5/>
9. Kuhn, M., & Johnson, K. (2013). *Applied Predictive Modeling*. Springer.

**Code Explanation:**

**Covering : app.py**

**1. Imports and Global Variables**

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

from sklearn.metrics import classification\_report, confusion\_matrix

from sklearn.linear\_model import LogisticRegression

from sklearn.neighbors import KNeighborsClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import GradientBoostingClassifier

import joblib

import os

import json

from datetime import datetime

import warnings

warnings.filterwarnings("ignore")

# Global variables to store data and models

training\_data = None

testing\_data = None

models = {}

preprocessor = None

scaler = StandardScaler()

**Explanation:**

* Imports essential libraries for data handling (pandas, numpy), machine learning (scikit-learn models and tools), file management (os, joblib), and warnings suppression.
* Defines global variables:
  + training\_data and testing\_data: DataFrames to hold datasets.
  + models: Dictionary to store model instances.
  + preprocessor: Will store the column transformer for categorical encoding.
  + scaler: Instance of StandardScaler for numeric feature scaling.

**2. Model Initialization**

def initialize\_models():

global models

models = {

"Logistic Regression": LogisticRegression(max\_iter=1000),

"Decision Tree": DecisionTreeClassifier(),

"Random Forest": KNeighborsClassifier(),

"Gradient Boosting": GradientBoostingClassifier()

}

**Explanation:**

* Initializes four classifiers stored in the models dictionary.
* Note: "Random Forest" is mislabeled—it actually initializes a K-Nearest Neighbors (KNeighborsClassifier). This is a code oversight.
* Logistic Regression is set with a maximum of 1000 iterations for convergence.

**3. Data Preprocessing**

def preprocess\_data(df):

# Drop irrelevant columns

df = df.drop(['id', 'attack\_cat'], axis=1, errors='ignore')

# Clamp extreme values

df\_numeric = df.select\_dtypes(include=[np.number])

for feature in df\_numeric.columns:

if df[feature].max() > 10 \* df[feature].median():

df[feature] = np.where(

df[feature] < df[feature].quantile(0.95),

df[feature],

df[feature].quantile(0.95)

)

# Log transform skewed features

for feature in df\_numeric.columns:

if df[feature].nunique() > 50:

df[feature] = np.log1p(df[feature])

# Reduce categorical cardinality

df\_cat = df.select\_dtypes(exclude=[np.number])

for feature in df\_cat.columns:

top\_labels = df[feature].value\_counts().nlargest(5).index

df[feature] = df[feature].apply(lambda x: x if x in top\_labels else '-')

return df

**Explanation:**

* Drops columns id and attack\_cat that are not used for prediction.
* Numeric features are examined for extreme outliers—if the max is more than 10x median, values above the 95th percentile are capped.
* Features with many unique values (>50) are log-transformed to reduce skewness (np.log1p applies log(1+x)).
* For categorical features, only the top 5 frequent categories are retained; others replaced with '-' to reduce cardinality.
* Returns the cleaned and transformed DataFrame.

**4. Load and Preprocess Function**

def load\_and\_preprocess(training\_path, testing\_path):

global training\_data, testing\_data

if not os.path.exists(training\_path) or not os.path.exists(testing\_path):

raise FileNotFoundError('One or both file paths do not exist')

training\_data = pd.read\_csv(training\_path)

testing\_data = pd.read\_csv(testing\_path)

training\_data = preprocess\_data(training\_data)

testing\_data = preprocess\_data(testing\_data)

return len(training\_data), len(testing\_data)

**Explanation:**

* Reads CSV files for training and testing datasets.
* Checks if files exist; raises error if not.
* Applies the preprocess\_data function to both datasets.
* Returns the number of samples in training and testing data for UI feedback.

**5. Training Models**

def train\_models():

global training\_data, testing\_data, preprocessor

if training\_data is None or testing\_data is None:

raise ValueError('Data not loaded')

cat\_features = training\_data.select\_dtypes(include='object').columns

preprocessor = ColumnTransformer(

transformers=[

('cat', OneHotEncoder(handle\_unknown='ignore'), cat\_features)

],

remainder='passthrough'

)

initialize\_models()

trained\_models = {}

for name, model in models.items():

pipeline = Pipeline([

('preprocessor', preprocessor),

('scaler', scaler),

('classifier', model)

])

pipeline.fit(training\_data.drop('label', axis=1), training\_data['label'])

trained\_models[name] = pipeline

joblib.dump(pipeline, f'models/{name.lower().replace(" ", "\_")}.joblib')

return list(models.keys())

**Explanation:**

* Validates data presence.
* Detects categorical columns to encode.
* Creates a ColumnTransformer that applies OneHotEncoder to categorical features; numeric features pass through untouched.
* Calls initialize\_models() to set classifiers.
* For each model, builds a pipeline that preprocesses, scales, and classifies data.
* Fits the pipeline to the training data.
* Saves the trained pipeline to disk using joblib with a filename derived from the model’s name.
* Returns the list of trained model names.

**6. Single Sample Streaming Prediction**

def stream\_next(model\_name):

global testing\_data

if testing\_data is None or len(testing\_data) == 0:

raise ValueError('No more data to stream or data not loaded')

model = joblib.load(f'models/{model\_name.lower().replace(" ", "\_")}.joblib')

row = testing\_data.iloc[0:1]

testing\_data = testing\_data.iloc[1:]

prediction = model.predict(row.drop('label', axis=1))

probability = model.predict\_proba(row.drop('label', axis=1))[0][1] if hasattr(model, 'predict\_proba') else None

result = {

'prediction': int(prediction[0]),

'probability': float(probability) if probability is not None else None,

'actual': int(row['label'].iloc[0]),

'remaining\_samples': len(testing\_data)

}

return result

**Explanation:**

* Checks if test data is loaded and non-empty.
* Loads the model pipeline from disk.
* Extracts the first row for prediction and removes it from the testing set.
* Predicts the label and, if supported, the probability of anomaly.
* Returns a dictionary with prediction, probability, actual label, and count of remaining samples.

**7. Ensemble Streaming Prediction**

def stream\_next\_ensemble(threshold=0.8):

global testing\_data

if testing\_data is None or len(testing\_data) == 0:

raise ValueError('No more data to stream or data not loaded')

row = testing\_data.iloc[0:1]

testing\_data = testing\_data.iloc[1:]

votes = []

probabilities = {}

for name in models.keys():

model = joblib.load(f'models/{name.lower().replace(" ", "\_")}.joblib')

proba = model.predict\_proba(row.drop('label', axis=1))[0][1] if hasattr(model, 'predict\_proba') else 0

probabilities[name] = proba

vote = 1 if proba > threshold else 0

votes.append(vote)

final\_prediction = 1 if sum(votes) > len(votes) // 2 else 0

result = {

'prediction': final\_prediction,

'votes': votes,

'probabilities': probabilities,

'actual': int(row['label'].iloc[0]),

'remaining\_samples': len(testing\_data)

}

return result

**Explanation:**

* Similar to stream\_next but gathers predictions from all models.
* For each model, loads pipeline, predicts anomaly probability.
* Converts probabilities to binary votes using a threshold (default 0.8).
* Majority vote determines final prediction.
* Returns detailed results including individual votes and probabilities.

**8. Batch Analysis of All Models**

def analyze\_all\_models():

global testing\_data

if testing\_data is None:

raise ValueError('Testing data not loaded')

results = {}

X\_test = testing\_data.drop('label', axis=1)

y\_test = testing\_data['label']

ensemble\_votes = np.zeros(len(y\_test))

ensemble\_confidences = np.zeros(len(y\_test))

for model\_name in models.keys():

model = joblib.load(f'models/{model\_name.lower().replace(" ", "\_")}.joblib')

y\_pred = model.predict(X\_test)

y\_proba = model.predict\_proba(X\_test)[:, 1] if hasattr(model, 'predict\_proba') else None

if y\_proba is not None:

ensemble\_votes += (y\_proba > 0.5).astype(int)

ensemble\_confidences += y\_proba

cm = confusion\_matrix(y\_test, y\_pred)

report = classification\_report(y\_test, y\_pred, output\_dict=True)

results[model\_name] = {

'report': report,

'confusion\_matrix': cm,

'predictions': y\_pred,

'probabilities': y\_proba

}

ensemble\_votes = ensemble\_votes / len(models)

ensemble\_confidences = ensemble\_confidences / len(models)

ensemble\_predictions = (ensemble\_votes > 0.5).astype(int)

ensemble\_cm = confusion\_matrix(y\_test, ensemble\_predictions)

ensemble\_report = classification\_report(y\_test, ensemble\_predictions, output\_dict=True)

results['Ensemble'] = {

'report': ensemble\_report,

'confusion\_matrix': ensemble\_cm,

'predictions': ensemble\_predictions,

'probabilities': ensemble\_confidences,

'votes': ensemble\_votes

}

return results

**Explanation:**

* Evaluates all models on the entire testing set.
* Collects predictions, probabilities, confusion matrices, and detailed classification reports.
* Aggregates votes and confidence scores for ensemble prediction.
* Returns a dictionary of detailed results per model and for the ensemble.

**Covering: ads\_gui.py**

**1. Imports and Setup**

import sys

from PyQt5.QtWidgets import (

QApplication, QWidget, QVBoxLayout, QHBoxLayout, QPushButton, QLabel,

QFileDialog, QMessageBox, QGroupBox, QComboBox, QFrame

)

from PyQt5.QtCore import Qt, QTimer

from PyQt5.QtGui import QFont

import traceback

import app # Backend logic module

from matplotlib.backends.backend\_qt5agg import FigureCanvasQTAgg as FigureCanvas

import matplotlib.pyplot as plt

import numpy as np

**Explanation:**

* Imports PyQt5 components for GUI widgets and layouts.
* Uses QTimer for timed events (streaming).
* Imports the backend module app.py to connect GUI actions with model logic.
* Uses Matplotlib to embed plots within the GUI (FigureCanvas).
* Imports NumPy for numerical operations.
* traceback is imported for detailed error logging.

**2. ResultsCard Widget**

This custom widget displays model performance metrics and visualizations.

class ResultsCard(QFrame):

def \_\_init\_\_(self, parent=None):

super().\_\_init\_\_(parent)

self.setFrameStyle(QFrame.StyledPanel | QFrame.Raised)

self.setStyleSheet("""

QFrame {

background-color: white;

border-radius: 10px;

padding: 15px;

}

""")

layout = QVBoxLayout()

# Title Label

title = QLabel("Model Performance")

title.setFont(QFont("Arial", 12, QFont.Bold))

title.setAlignment(Qt.AlignCenter)

layout.addWidget(title)

# Model selector dropdown

self.model\_selector = QComboBox()

self.model\_selector.currentIndexChanged.connect(self.update\_metrics)

layout.addWidget(self.model\_selector)

# Metrics layout for accuracy and F1 score

metrics\_layout = QHBoxLayout()

self.accuracy\_label = QLabel("Accuracy: --")

self.accuracy\_label.setFont(QFont("Arial", 10))

metrics\_layout.addWidget(self.accuracy\_label)

self.f1\_label = QLabel("F1 Score: --")

self.f1\_label.setFont(QFont("Arial", 10))

metrics\_layout.addWidget(self.f1\_label)

layout.addLayout(metrics\_layout)

# Confusion matrix plot area

self.figure = plt.figure(figsize=(4, 3))

self.canvas = FigureCanvas(self.figure)

layout.addWidget(self.canvas)

# Ensemble voting section

ensemble\_frame = QFrame()

ensemble\_frame.setStyleSheet("""

QFrame {

background-color: #f0f0f0;

border-radius: 5px;

padding: 10px;

margin-top: 10px;

}

""")

ensemble\_layout = QVBoxLayout()

ensemble\_title = QLabel("Ensemble Voting Results")

ensemble\_title.setFont(QFont("Arial", 10, QFont.Bold))

ensemble\_layout.addWidget(ensemble\_title)

self.ensemble\_metrics = QLabel("Ensemble metrics will appear here")

self.ensemble\_metrics.setFont(QFont("Arial", 9))

ensemble\_layout.addWidget(self.ensemble\_metrics)

self.voting\_figure = plt.figure(figsize=(4, 2))

self.voting\_canvas = FigureCanvas(self.voting\_figure)

ensemble\_layout.addWidget(self.voting\_canvas)

ensemble\_frame.setLayout(ensemble\_layout)

layout.addWidget(ensemble\_frame)

self.setLayout(layout)

**Explanation:**

* ResultsCard inherits from QFrame to create a styled panel.
* Contains:
  + A dropdown (QComboBox) for selecting which model’s results to display.
  + Labels for accuracy and F1 score.
  + A Matplotlib canvas to show the confusion matrix.
  + A dedicated section for ensemble voting metrics and vote distribution histogram.
* The widget updates dynamically based on the selected model.

**3. Updating Metrics and Plots in ResultsCard**

def update\_metrics(self, index):

if not hasattr(self, 'results'):

return

model\_name = self.model\_selector.currentText()

model\_results = self.results.get(model\_name, {})

if not model\_results:

return

report = model\_results.get('report', {})

if not report:

return

accuracy = report.get('accuracy', 0)

self.accuracy\_label.setText(f"Accuracy: {accuracy:.2%}")

f1 = report.get('weighted avg', {}).get('f1-score', 0)

self.f1\_label.setText(f"F1 Score: {f1:.2%}")

# Plot confusion matrix

self.figure.clear()

ax = self.figure.add\_subplot(111)

cm = model\_results.get('confusion\_matrix', np.zeros((2, 2)))

ax.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)

ax.set\_title('Confusion Matrix')

ax.set\_xticks([0, 1])

ax.set\_yticks([0, 1])

ax.set\_xticklabels(['Normal', 'Anomaly'])

ax.set\_yticklabels(['Normal', 'Anomaly'])

thresh = cm.max() / 2.

for i in range(cm.shape[0]):

for j in range(cm.shape[1]):

ax.text(j, i, format(cm[i, j], 'd'),

ha="center", va="center",

color="white" if cm[i, j] > thresh else "black")

self.canvas.draw()

if model\_name == 'Ensemble':

self.update\_ensemble\_voting(model\_results)

else:

self.ensemble\_metrics.setText("Select 'Ensemble' to view voting results")

self.voting\_figure.clear()

self.voting\_canvas.draw()

**Explanation:**

* Updates accuracy and F1 score text labels from the classification report.
* Draws confusion matrix heatmap with annotations.
* If the “Ensemble” model is selected, calls method to update ensemble-specific plots.
* Otherwise, clears ensemble section.

**4. Ensemble Voting Visualization**

def update\_ensemble\_voting(self, ensemble\_results):

report = ensemble\_results.get('report', {})

accuracy = report.get('accuracy', 0)

f1 = report.get('weighted avg', {}).get('f1-score', 0)

self.ensemble\_metrics.setText(

f"Ensemble Accuracy: {accuracy:.2%}\n"

f"Ensemble F1 Score: {f1:.2%}"

)

self.voting\_figure.clear()

ax = self.voting\_figure.add\_subplot(111)

votes = ensemble\_results.get('votes', [])

if len(votes) > 0:

ax.hist(votes, bins=20, range=(0, 1), alpha=0.7)

ax.set\_title('Distribution of Model Votes')

ax.set\_xlabel('Vote Proportion')

ax.set\_ylabel('Count')

ax.grid(True, alpha=0.3)

self.voting\_canvas.draw()

**Explanation:**

* Displays ensemble accuracy and F1 score.
* Draws a histogram of voting proportions across samples.
* Helps users understand how strongly models voted for anomalies.

**5. Setting and Clearing Results**

def set\_results(self, results):

self.results = results

self.model\_selector.clear()

if results:

self.model\_selector.addItems(results.keys())

self.update\_metrics(0)

else:

self.accuracy\_label.setText("Accuracy: --")

self.f1\_label.setText("F1 Score: --")

self.figure.clear()

self.canvas.draw()

self.ensemble\_metrics.setText("Ensemble metrics will appear here")

self.voting\_figure.clear()

self.voting\_canvas.draw()

**Explanation:**

* Sets the results dictionary.
* Updates dropdown items with model names.
* Clears all display elements if no results provided.

**6. Main Application Class ADSApp**

**Initialization and UI Setup**

class ADSApp(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

self.streaming = False

self.stream\_timer = QTimer()

self.stream\_timer.setInterval(2000) # Stream every 2 seconds

self.stream\_timer.timeout.connect(self.stream\_next\_auto)

self.streamed\_indices = []

self.streamed\_labels = []

self.anomaly\_rates = []

def initUI(self):

self.setWindowTitle('Anomaly Detection System')

self.setGeometry(100, 100, 1100, 800)

main\_layout = QVBoxLayout()

# File selection, action buttons, status, results, and live plots groups created here (omitted for brevity)

self.setLayout(main\_layout)

self.train\_path = None

self.test\_path = None

**Explanation:**

* Initializes the main widget.
* Sets up a QTimer for streaming with 2-second intervals.
* Stores streamed sample indices, labels, and anomaly rates for visualization.
* initUI() creates the entire window layout including file selectors, action buttons, status displays, result panels, and live analytics charts.

**7. File Selection**

def select\_train(self):

path, \_ = QFileDialog.getOpenFileName(self, 'Select Training CSV', '', 'CSV Files (\*.csv)')

if path:

self.train\_path = path

self.train\_label.setText(f'Training: {path}')

def select\_test(self):

path, \_ = QFileDialog.getOpenFileName(self, 'Select Testing CSV', '', 'CSV Files (\*.csv)')

if path:

self.test\_path = path

self.test\_label.setText(f'Testing: {path}')

**Explanation:**

* Opens file dialog to select CSV files for training and testing datasets.
* Updates labels with selected file paths.

**8. Data Loading and Preprocessing**

def load\_and\_preprocess(self):

if not self.train\_path or not self.test\_path:

self.show\_error('Please select both training and testing files.')

return

self.status.setText('Loading and preprocessing data...')

QApplication.processEvents()

try:

n\_train, n\_test = app.load\_and\_preprocess(self.train\_path, self.test\_path)

self.status.setText(f'Data loaded. Training samples: {n\_train}, Testing samples: {n\_test}')

self.btn\_train\_models.setEnabled(True)

self.btn\_stream\_toggle.setEnabled(False)

self.btn\_analyze.setEnabled(False)

self.results\_card.set\_results({})

self.streamed\_indices = []

self.streamed\_labels = []

self.anomaly\_rates = []

self.update\_plots()

except Exception as e:

self.show\_error(f'Error: {str(e)}\n{traceback.format\_exc()}')

**Explanation:**

* Calls backend load\_and\_preprocess method.
* Updates GUI elements accordingly.
* Resets streamed data and plots on new load.
* Shows error dialogs on exceptions.

**9. Model Training**

def train\_models(self):

self.status.setText('Training models...')

QApplication.processEvents()

try:

models = app.train\_models()

self.status.setText(f'Models trained: {", ".join(models)}')

self.btn\_stream\_toggle.setEnabled(True)

self.btn\_analyze.setEnabled(True)

self.results\_card.set\_results({})

except Exception as e:

self.show\_error(f'Error: {str(e)}\n{traceback.format\_exc()}')

**Explanation:**

* Initiates training via app.train\_models().
* Enables streaming and analysis buttons post training.
* Resets results display.

**10. Streaming Control and Execution**

def toggle\_streaming(self):

if not self.streaming:

self.streaming = True

self.btn\_stream\_toggle.setText('Stop Streaming')

self.status.setText('Streaming started...')

self.stream\_timer.start()

else:

self.streaming = False

self.btn\_stream\_toggle.setText('Start Streaming')

self.status.setText('Streaming stopped.')

self.stream\_timer.stop()

def stream\_next\_auto(self):

try:

result = app.stream\_next\_ensemble(threshold=0.7)

idx = len(self.streamed\_indices)

is\_anomaly = result['prediction'] == 1

self.status.setText(f"Streamed one row. Remaining: {result['remaining\_samples']}")

self.streamed\_indices.append(idx)

self.streamed\_labels.append(result['prediction'])

anomaly\_count = sum(self.streamed\_labels)

total = len(self.streamed\_labels)

self.anomaly\_rates.append(anomaly\_count / total if total > 0 else 0)

self.update\_plots()

if result['remaining\_samples'] == 0:

self.toggle\_streaming()

self.status.setText('Streaming finished. No more data.')

except Exception as e:

self.show\_error(f'Error: {str(e)}\n{traceback.format\_exc()}')

self.toggle\_streaming()

**Explanation:**

* toggle\_streaming starts/stops the timer controlling streaming.
* stream\_next\_auto fetches the next sample’s ensemble prediction.
* Updates stored streamed indices, labels, and computes anomaly rate.
* Updates live plots accordingly.
* Stops streaming when no data remains or on errors.

**11. Plot Updates**

def update\_plots(self):

# Bar plot for cumulative counts of normal vs anomaly

self.ax\_bar.clear()

normal\_count = self.streamed\_labels.count(0)

anomaly\_count = self.streamed\_labels.count(1)

self.ax\_bar.bar(['Normal', 'Anomaly'], [normal\_count, anomaly\_count], color=['green', 'red'])

self.ax\_bar.set\_title('Cumulative Count')

self.ax\_bar.set\_ylabel('Count')

self.canvas\_bar.draw()

# Line plot for anomaly rate with anomaly markers

self.ax\_line.clear()

if self.anomaly\_rates:

x = np.arange(1, len(self.anomaly\_rates) + 1)

y = np.array(self.anomaly\_rates)

self.ax\_line.plot(x, y, color='blue', marker='o', label='Normal')

anomaly\_indices = np.where(np.array(self.streamed\_labels) == 1)[0]

if len(anomaly\_indices) > 0:

self.ax\_line.plot(

x[anomaly\_indices], y[anomaly\_indices],

'v', color='red', markersize=10, label='Anomaly'

)

self.ax\_line.set\_ylim(0, 1)

self.ax\_line.legend()

self.ax\_line.set\_title('Cumulative Anomaly Rate')

self.ax\_line.set\_xlabel('Stream Index')

self.ax\_line.set\_ylabel('Anomaly Rate')

self.canvas\_line.draw()

# Orange line+scatter plot with decision boundary

self.ax\_spike\_line.clear()

if self.streamed\_labels:

x = np.arange(1, len(self.streamed\_labels) + 1)

y = np.full\_like(x, 0.5, dtype=float)

labels = np.array(self.streamed\_labels)

y[labels == 1] = 1.0 # Anomaly

y[labels == 0] = 0.0 # Normal

self.ax\_spike\_line.plot(x, y, color='orange', linewidth=2, marker='o', markersize=7,

markerfacecolor='orange', markeredgecolor='orange', label='Normal/Anomaly')

self.ax\_spike\_line.axhline(0.5, color='black', linewidth=2, linestyle='--', alpha=0.7, label='Divider (0.5)')

self.ax\_spike\_line.set\_ylim(-0.1, 1.1)

self.ax\_spike\_line.set\_title('Normal(0) - Anomaly(1) with Divider')

self.ax\_spike\_line.set\_xlabel('Stream Index')

self.ax\_spike\_line.set\_ylabel('Value')

self.ax\_spike\_line.legend()

self.canvas\_spike\_line.draw()

**Explanation:**

* Updates three plots on streaming data:
  + Bar chart of cumulative normal vs anomaly counts.
  + Line chart of anomaly rate over streaming indices with anomaly markers.
  + Orange line/scatter plot showing classification labels with horizontal decision boundary at 0.5.

**12. Full Analysis**

def analyze\_all(self):

self.status.setText('Running full analysis...')

QApplication.processEvents()

try:

results = app.analyze\_all\_models()

self.results\_card.set\_results(results)

self.status.setText('Analysis complete.')

except Exception as e:

self.show\_error(f'Error: {str(e)}\n{traceback.format\_exc()}')

self.results\_card.set\_results({})

**Explanation:**

* Runs full batch evaluation on all models via backend.
* Updates results card with comprehensive reports.
* Handles exceptions gracefully.

**13. Error Handling**

def show\_error(self, msg):

self.status.setText('Error')

QMessageBox.critical(self, 'Error', msg)

self.results\_card.set\_results({})

**Explanation:**

* Displays critical error dialogs.
* Resets results display to avoid stale or misleading data.

**14. Main Entry Point**

if \_\_name\_\_ == '\_\_main\_\_':

app\_qt = QApplication(sys.argv)

ex = ADSApp()

ex.show()

sys.exit(app\_qt.exec\_())

**Explanation:**

* Creates the PyQt application.
* Instantiates the main ADSApp widget and displays it.
* Starts the event loop.

**User and Developer Guide: Setting Up and Running the Anomaly Detection System**

This guide provides detailed instructions for users and developers to clone, set up, and run the Anomaly Detection System (ADS) application. It covers environment preparation, dependency installation, and application usage.

**1. Prerequisites**

Before starting, ensure the following prerequisites are met:

* **Python 3.7 or higher** installed on your system. Download from [python.org](https://www.python.org/downloads/) if needed.
* Basic familiarity with command-line interface (CLI).
* Internet connectivity for downloading dependencies.

**2. Cloning the Repository**

1. Open your terminal or command prompt.
2. Navigate to the directory where you want to clone the project.
3. Run the following command to clone the repository:
   * git clone <repository-url>
4. Change directory to the cloned repository:
   * cd <repository-directory>

**3. Creating and Activating a Virtual Environment**

It is recommended to use a Python virtual environment to isolate project dependencies:

* **On Windows:**
  + python -m venv venv
  + .\venv\Scripts\activate
* **On Unix/macOS:**
  + python3 -m venv venv
  + source ./venv/bin/activate

*Note:* After activation, your CLI prompt should be prefixed with (venv) indicating the virtual environment is active.

**4. Installing Dependencies**

The project dependencies are listed in the requirements.txt file. To install them:

pip install -r requirements.txt

This will install:

* PyQt5 (for GUI)
* Matplotlib (for plotting)
* NumPy and Pandas (for numerical operations and data handling)
* Scikit-learn (for machine learning models)
* Joblib (for model serialization)

**5. Directory Structure Overview**

After cloning, you should see the following key files and folders:

├── app.py # Core backend logic and machine learning code

├── ads\_gui.py # GUI application code using PyQt5

├── requirements.txt # Python dependencies list

├── models/ # Directory where trained models are saved

├── UNSW\_NB15\_training-set.csv # Training dataset (example)

├── UNSW\_NB15\_testing-set.csv # Testing dataset (example)

**6. Running the Application**

To start the graphical user interface of the Anomaly Detection System:

python ads\_gui.py

This will open the application window with options to:

* Select training and testing CSV datasets.
* Load and preprocess data.
* Train machine learning models.
* Start streaming anomaly detection.
* View live results and detailed analysis.

**7. Using the Application**

**Step 1: Data Selection**

* Click **Select Training CSV** and choose your training dataset file (e.g., *UNSW\_NB15\_training-set.csv).*
* Click **Select Testing CSV** and choose your testing dataset file (e.g., *UNSW\_NB15\_testing-set.csv).*

**Step 2: Load & Preprocess**

* Click **Load & Preprocess Data** to import and prepare datasets.
* Status will update indicating sample counts.

**Step 3: Train Models**

* Click **Train Models** to train all included classifiers on the training data.
* Wait for confirmation that models have been trained and saved.

**Step 4: Streaming Anomaly Detection**

* Click **Start Streaming** to begin real-time simulation of anomaly detection.
* Observe live visualizations of cumulative counts, anomaly rates, and classification markers.
* Click **Stop Streaming** to pause.

**Step 5: Full Analysis**

* Click **Full Analysis** to run batch evaluation on the entire test dataset.
* Review detailed model performance metrics and confusion matrices.

**8. Troubleshooting**

* **Module Not Found Errors:** Ensure the virtual environment is activated and dependencies are installed.
* **Data Loading Issues:** Verify that the selected CSV files exist and have the required format (including a label column).
* **GUI Display Problems:** Ensure your system supports PyQt5 and the display drivers are up to date.
* **Model Training Errors:** Check available system memory and correct dataset preprocessing.

**9. Extending the Application**

Developers can extend the system by:

* Adding new machine learning models in app.py under the initialize\_models() function.
* Modifying the preprocessing pipeline in preprocess\_data().
* Enhancing GUI features in ads\_gui.py.
* Integrating additional datasets by placing CSV files in the project directory.

**10. Deactivating the Virtual Environment**

Once finished, deactivate the virtual environment by running:

*deactivate*