AI Project Report

1. Project Setup and Data Analysis

Dataset Overview

- **Understanding the Data:** The first step involved analyzing the dataset's distribution and functionality. This was done by examining the features.csv file, which details 49 columns representing every possible type of access.
- **Tools & Environment:** I used VSCode and Python as the development environment and language to generate the necessary code for iterating through the CSVs. This setup also facilitates linking with GitHub, saving time on project export and version control.

Data Integrity Issue

- **Correction:** A critical issue was identified during the initial review: the file names for training-set and testing-set were swapped in the source files. I realized this by comparing the row counts with the information provided on the original download page. Additionally, when training the algorithm with the files as they were, the results were significantly poorer because the "testing" file (actually the training set) was smaller and had roughly half the attacks. Swapping the names resolved this problem.

2. Methodology and Baseline Strategy

Model Selection

- **Baseline:** Logistic Regression was chosen for its simplicity and interpretability, with the option to change it later for efficiency.
- **Advanced Model:** Random Forest was selected as the advanced model for similar reasons.

Preprocessing

- **Configuration (config.py):** Established RANDOM_SEED = 42 to ensure reproducibility and defined TARGET_COL = 'Label' (0=Normal, 1=Attack).
- **Data Cleaning:**
    1. Transformed the target column (Label) and removed it from the training set to prevent data leakage.
    2. Handled null values in service, state, and proto columns.
    3. Used fit() + transform() to standardize scales across all columns.
- **Classification Strategy:** Initially, I decided to focus on binary classification (Normal vs. Attack) because early attempts at multi-class classification yielded very low results (35-40% F1-score).

3. Implementation and Notebook Structure

After achieving a decent baseline (60%) through trial and error, I structured the project into organized notebooks for better clarity and shareability.

Notebook 1: Baseline (Logistic Regression)

- **Optimization:** Utilized libraries like joblib to save results in .pkl format (stored in a processed folder) for efficiency.
- **Implementation:** Used sklearn.linear_model.LogisticRegression. This streamlined the code and slightly improved precision.
- **Results:** Achieved a final **F1-score of 0.6473**, a solid baseline.
- **Evaluation:** Included a Confusion Matrix (to visualize False Positives/Negatives), ROC Curve, and script to export results.

Notebook 2: Feature Engineering

- **Experimentation:**
    - *Attempt 1:* Tested random ratios (failed, results dropped by 10%).
    - *Attempt 2:* Added specific features mentioned in the prompt (rates, averages, total_bytes).
- **Observation:** The results worsened by 4%. Given the complex and "tricky" nature of the dataset, simple algorithms (and even advanced ones) struggled with the added noise, leading to overfitting.
- **PCA Analysis:** Applied PCA to reduce features to 3 (retaining 95% variance). The score dropped further from 0.6450 to **0.6194**.
- **Conclusion:** The dataset is highly sensitive; adding features creates noise, while reducing them leads to information loss.

Notebook 3: Advanced Models

- **Selection:** Discarded Neural Networks due to inefficiency with high feature counts. Chose Random Forest.
- **Initial Struggle:** Early implementations using libraries yielded poor results (< 0.5). Attempted Gradient Boosting and XGBoost with similar low performance.
- **Refinement:** Manual coding/tuning improved the score to ~0.75.
- **Key Insight:** Feedback from the professor highlighted that the number of trees (100-200) was too high for this dataset, causing poor performance.

- **Tuning Results:**
    1. **Random Forest V1:** n_estimators=26, max_depth=4, n_jobs=-1. **F1-score = 0.7115**.
    2. **Random Forest V2:** Added min_samples_split=5, min_samples_leaf=2, max_features=0.1. **F1-score = 0.7698**.
    3. **XGBoost (Final):** Implemented with optimized parameters. **Best Score = 0.8069**.

Notebook 4: Multi-Class Classification

- **Setup:** Re-imported non-scaled datasets and fixed attack_cat labels (removing nulls and correcting types).
- **Results:** As expected, performance dropped compared to binary classification (**F1-score = 0.69**).
- **Class Imbalance:** Rare classes (<50 instances) were barely detected.
- **Solution:** Implemented compute_class_weight to balance class importance. This improved detection for "Generic", "Exploits", and "Reconnaissance" attacks.
- **Trade-off:** General accuracy decreased. Common attacks like "Backdoor" and "Shellcode" saw performance drops, likely because balancing weights reduced their relative importance.

4. Scalability Analysis & Conclusions

- **Dataset Size Analysis:** I included a graph showing the impact of dataset size on XGBoost performance. The performance drops significantly after using 70% of the data. This demonstrates that as the quantity of classes and the disparity in their frequency increases, the algorithm struggles to maintain reliable real-world performance.
- **Final Learnings:**
    - The environment and tools are crucial.
    - Feature scaling, algorithm selection, and iterative testing are fundamental.
    - Most importantly, deeply understanding the nature of the information/data is essential for achieving good results.