CSAI-253 | Machine Learning
Project Report - Phase II

Team:
Machine Not Learning

Team Email:
Amr Yasser 202301043

NAME	ID	SECTION
Amr Yasser	202301043	5
Amr Mahmoud	202300491	2
Momen Mahmoud	202300971	2
Aya Ayoub	202300348	2

CSAI-253 | Machine Learning
Project Report - Phase II
Tasks Workload

TASK	DONE BY
Data Loading & Cleaning	Amr Yasser
EDA: Visualize data distributions using histograms, box plots, etc	Aya
FE: Handle categorical variables using appropriate encoding techniques.	Amr Yasser
FE: Scale numerical features if required.	Aya
MI: XGBoost & evaluate its performance & compare it with different models.	Amr Mahmoud
MI: Random Forest & evaluate its performance & compare it with different models.	Amr Yasser
MI: LightGBM & evaluate its performance & compare it with different models.	Momen
MI: Decision Tree & evaluate its performance & compare it with different models.	Amr Mahmoud
Stacking Ensembling technique	Momen

]

A. Problem Statement

- **Objective:** In this project, we aim to build a machine learning classification model that can accurately distinguish between different types of cyber-attacks (e.g., DDoS, DoS, Mirai, Recon, MITM) and benign traffic, based on per-flow network traffic features. By analyzing features such as flow durations, packet rates, flag counts, and protocol indicators, the model will learn characteristic patterns of each attack type to enable real-time detection and response.

- **Significance:** Detecting anomalous connections is crucial for protecting military networks from intrusions and cyberattacks.

B. Data Exploration & Preprocessing

1. **Feature Overview**

1.1 Time & Size Features

- **flow_time**

  Definition: Total elapsed time from the first to the last packet in a flow (e.g. seconds or milliseconds).

  Usage: Differentiates short (“chatty”) flows from long bulk transfers or anomalies.

- **header_size**

  Definition: Total bytes of IP/TCP headers exchanged.

  Usage: Higher header overhead may indicate many small packets or certain protocol behaviors.

- **packet_duration**

  Definition: Average (or total) duration per packet within the flow.

  Usage: Distinguishes interactive traffic (short durations) from bulk transfers (long durations).

1.2 Throughput / Rate Features

- **overall_rate**

  Definition: Total bytes divided by flow_time (bytes/sec).

  Usage: Captures end-to-end throughput of the flow. src_rate / dst_rate

- **src_rate / dst_rate**

  Definition: Throughput in source→destination and destination→source directions.

  Usage: Asymmetric rates often flag client/server roles or scanning.

1.3 Packet-Count & Flag-Count Features

- **Packet counts**

  (fin_packets , urg_packets , rst_packets)

  Definition: Number of packets carrying the FIN, URG, or RST flags.

  Usage: FIN ⇒ normal teardown

  RST ⇒ abrupt reset (scans, errors)

  URG ⇒ rare urgent data

- **TCP-flag counts**
  fin_flags , syn_flags , rst_flags , psh_flags , ack_flags
  Definition: Number of packets with each TCP control flag set.
  Usage: SYN ⇒ connection initiation
  ACK ⇒ acknowledgments
  PSH ⇒ push (application data)
  Patterns of these can reveal scans or DoS behavior.

1.4 Statistical Features

- **max_value**
  Definition: Maximum observed value in a per-packed metric (e.g. packet size).

- **value_covariance**
  Definition: Covariance (or variance) of a per-packet metric over the flow.
  Usage: Quantifies burstiness or variability in packet sizes/timings.

1.5 Protocol Indicators (One-Hot)

- **protocol_http, protocol_https, protocol_tcp, protocol_udp, protocol_icmp**

  Definition: Binary flags (0/1) indicating which protocol the flow used.
  Usage: Encodes protocol type directly; mutually-exclusive in most cases.

1.6 Target Label

- **label**
  Definition: Ground-truth class (e.g. benign vs. malicious, or specific attack types).

Usage: Supervised target—never apply feature transforms directly to this column.

Why You Must Handle Duplicates Before Outliers, Scaling, or Modeling

When your dataset contains a **significant portion of duplicates** — like **96,187 out of 938,583 rows (~10%)** — it is critical to resolve them *before* any other preprocessing steps such as:

Outlier detection

Feature scaling

Train–test splitting

Encoding categorical features

Ignoring duplicates early on can seriously skew your downstream pipeline.

What Happens If You Don't Deduplicate First?

Effect	Explanation
Outlier methods get biased	Techniques like IQR or modified Z-score will miscalculate spread by treating duplicates as valid, reinforcing values that might otherwise be seen as outliers.
Scalers get skewed	Min–Max, Z-Score, and others will calculate incorrect ranges and distributions due to repeated values.
Leakage between train/test	If duplicates are split across training and testing, your model may appear to perform better than it should.
False sense of accuracy	Your classifier may memorize repeated records instead of generalizing, leading to overfitting and inflated performance metrics.

Bottom line: Always handle duplicates first — it's fast, easy, and protects the

integrity of every step that follows.

Summary of Data Deduplication Techniques to Improve Model Accuracy

This guide outlines six key techniques to handle duplicates in your dataset, detailing what each method does, why it's worth trying, and how it can contribute to better model performance and reliability.

1. Direct Removal of Exact Duplicates

What it is:
Remove rows that are completely identical using

`pandas.drop_duplicates()` . **Why try it:**

- Eliminates over-represented rows that can distort training.
- Prevents data leakage into test sets.

How it helps:
Reduces overfitting and creates a fairer, more generalizable dataset.

2. Train–Test Aware Deduplication (or Grouped Cross Validation)

What it is:
Assign entire *duplicate groups* (i.e. identical rows) to only **one side** of the split — either the training set **or** the test set — never both. For cross-validation, use grouping logic like `GroupKFold` to ensure the same rule applies across folds.

Why use it:

Prevents **leakage**: no row (or its identical copy) ends up in both train and test. Preserves **natural frequency** for duplicates that reflect real-world distributions. Maintains **evaluation honesty**, especially when duplicates are frequent.

How it helps:
Avoids "cheating" by ensuring the model never sees test data during training — even indirectly via repetition. This leads to more **realistic generalization** performance and avoids overly optimistic accuracy.

3. Clustering-based Deduplication

What it is:
Group similar rows based on their feature values using clustering algorithms (e.g., DBSCAN), and remove near-duplicates.

Why try it:

Effective for large, structured datasets with **numerical and categorical features**. Automatically detects clusters of similar data, removing redundant records.

How it helps:
Reduces noise, eliminates near-duplicates, and improves generalization by ensuring that the model doesn't overfit on repeated data.

4. Instance Weighting

What it is:
Assign lower weights to duplicate records during model training to reduce their impact while keeping all data.

Why try it:

Preserves full data distribution.

Useful when repetitions may be informative but shouldn't dominate.

How it helps:
Balances learning from all data while reducing overrepresentation effects.

5. Combined Deduplication: Exact Removal + Clustering

What it is:
First remove exact duplicates, then apply clustering (e.g., DBSCAN) to identify and eliminate near-duplicates in feature space.

Why try it:

Double-layer defense: fast exact removal + intelligent group-level filtering. Combines the strengths of both approaches.

How it helps:
Maximizes deduplication effectiveness — catching both low-hanging duplicates and nuanced repetition patterns that could bias your model.

Outlier Handling Techniques Report:

Top 5 Outlier-Handling Techniques

1. Winsorization (Percentile-Based Capping)

Description:
Replace values below the p-th percentile and above the $(100 - p)$-th percentile with the corresponding percentile values.

Why Try:
Limits extreme distortions without dropping any records.

How It Helps:

Prevents rare, extreme flows from dominating tree splits or creating overly deep branches.

2. Direct Removal

Description:

Identify outliers (e.g. via IQR or Z-score rules) and drop those records entirely from the dataset.

Why Try:

Straightforward and effective when outliers are clearly erroneous or noise-driven.

How It Helps:

Ensures invalid or corrupted flow records (e.g. negative durations, infinities) do not skew model training.

3. Z-Score Trimming

Description:

Compute Z-scores $Z = \frac{x-\mu}{\sigma}$ and flag $|Z| > k$ (e.g. $k = 3$) as outliers; then drop or cap them at $\pm k\sigma$.

Why Try:

Provides a statistically principled rule for approximately Gaussian features.

How It Helps:

Systematically controls variance and removes or limits extreme packet-rate or duration values.

4. Log1p + Winsorization Combo

Description:

1. Apply a log1p transform $\log(x + 1)$ to compress right-skewed features. 2. Winsorize at chosen percentiles (e.g. 1st/99th).

Why Try:
Addresses long tails and any remaining extremes in one simple pipeline.

How It Helps:
Reduces skew, improves histogram quality for HGB and gain calculations for boosters, and caps residual outliers.

5. Isolation Forest

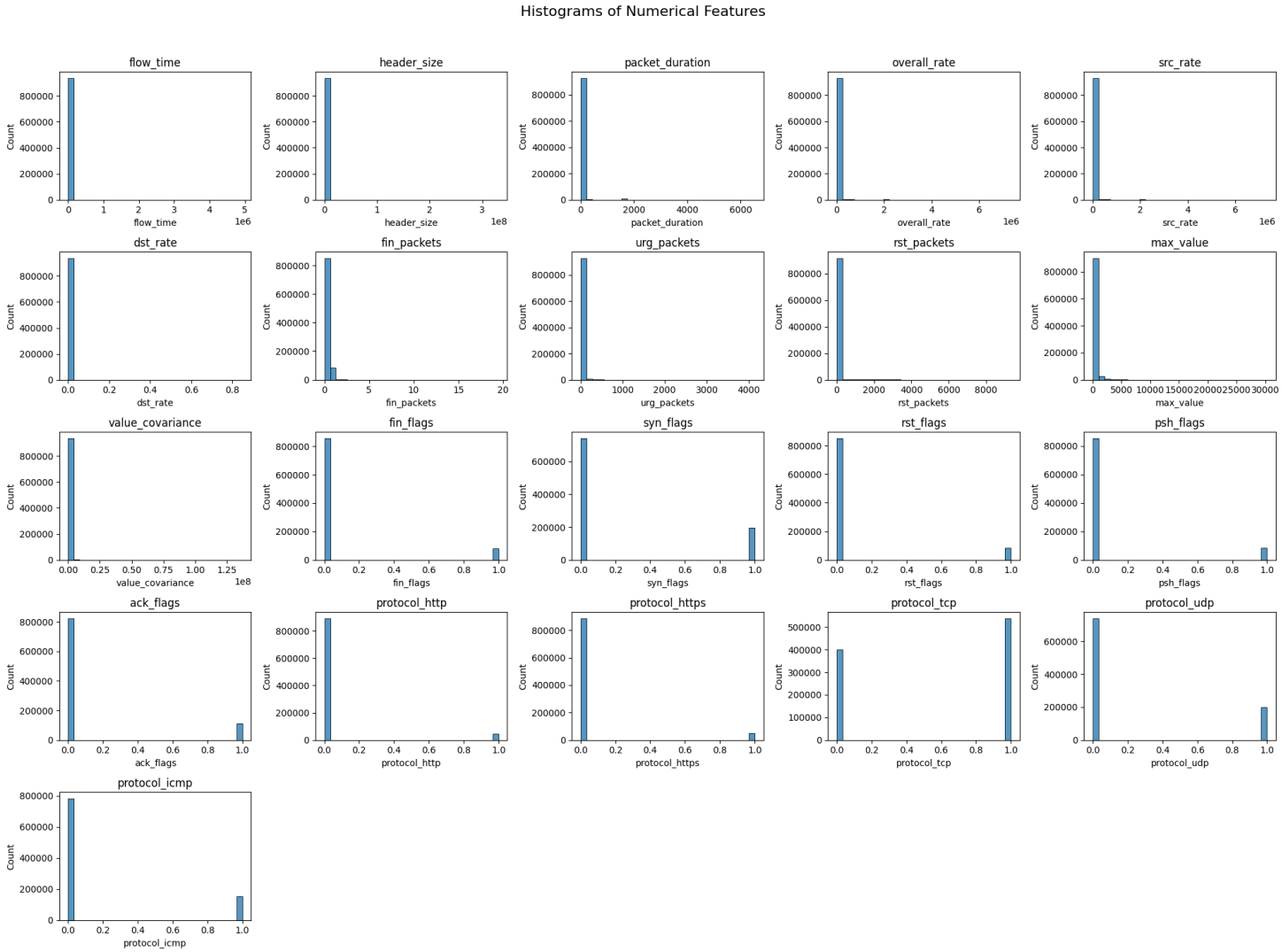
Description:
An unsupervised model that builds random “isolation trees”; points requiring fewer splits to isolate receive higher anomaly scores.

Why Try:
Captures complex, multivariate outliers that simple thresholds miss.

How It Helps:
Flags atypical combinations of flags, rates, and durations—remove or down-weight these before training

Exploratory Data Analysis (EDA)

- Visualized Numerical features using Histograms



Code:

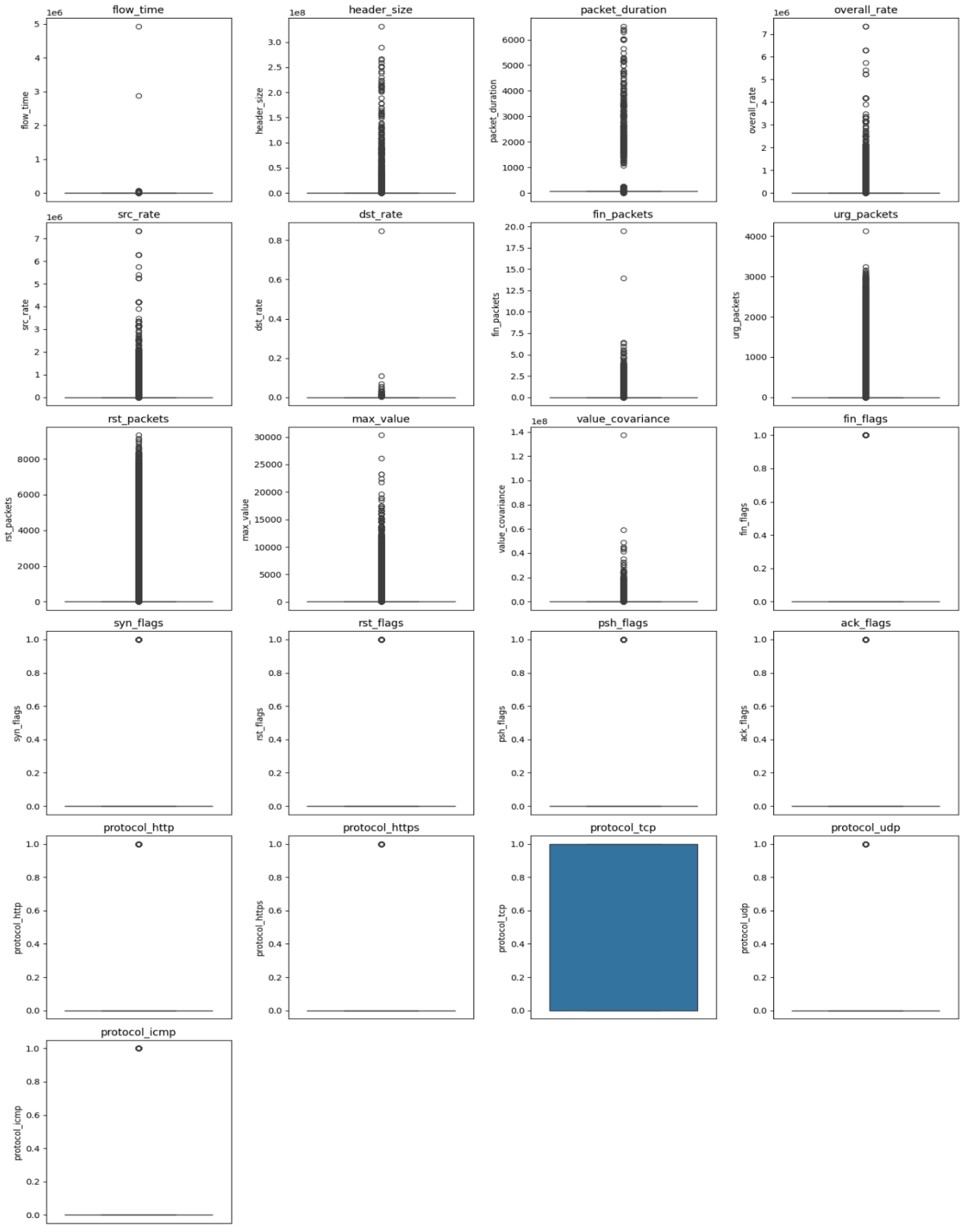
```
#Histograms

import math

len_of_num_cols= len(numerical_columns)
num_cols_list = list(numerical_columns)
cols = 5
rows = math.ceil(len_of_num_cols / cols)

fig, axes = plt.subplots(rows, cols, figsize=(cols * 4, rows * 3))
axes = axes.flatten()

for i, col in enumerate(num_cols_list):
    sns.histplot(data[col], ax=axes[i], bins=30, kde=False)
    axes[i].set_title(col)

for j in range(i+1, len(axes)):
    axes[j].set_visible(False)

fig.suptitle("Histograms of Numerical Features", fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

- Visualized Numerical features using Boxplots

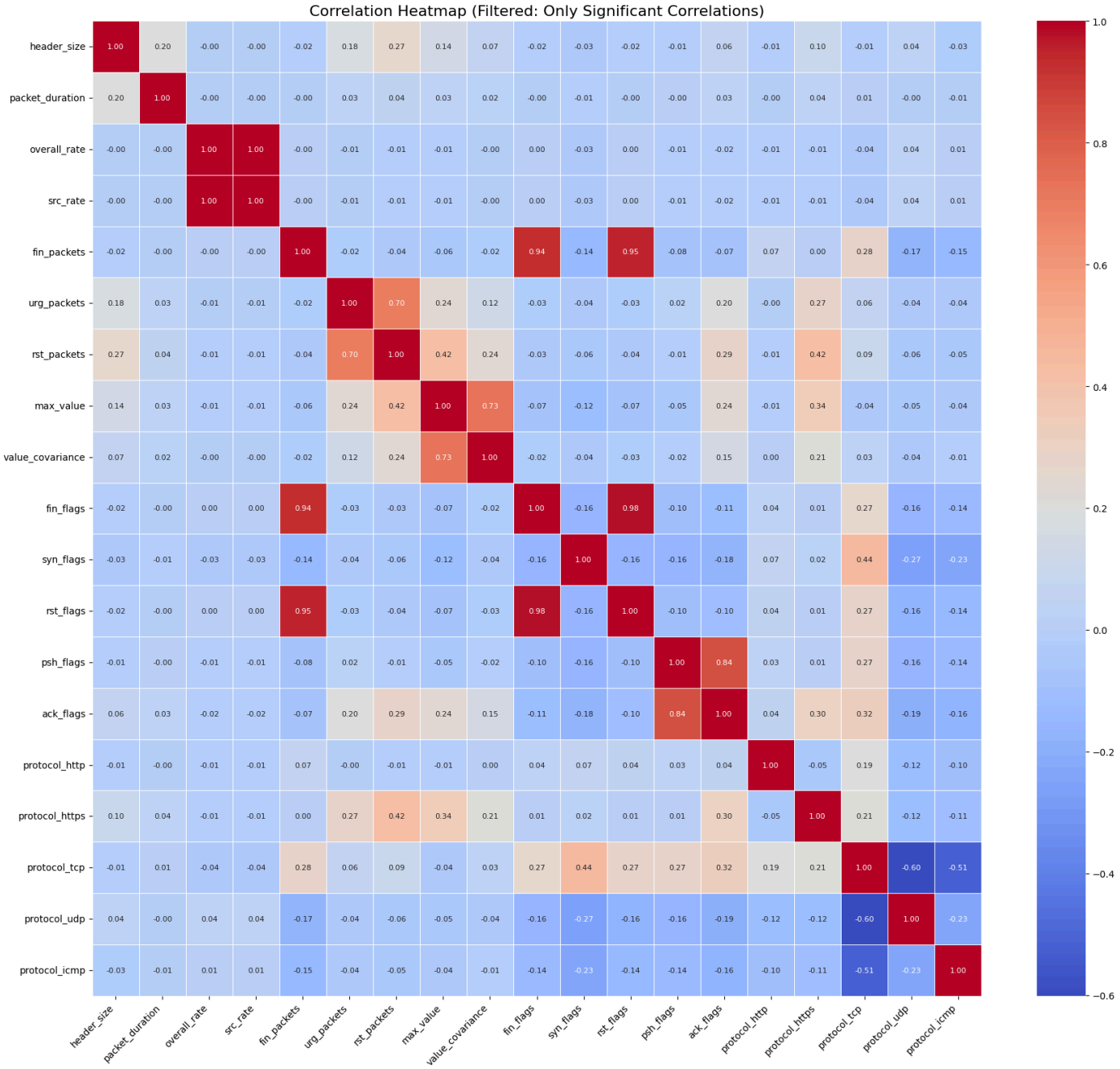
Code:

```
#Box plots

cols = 4
rows = math.ceil(len(numerical_columns) / cols)
plt.figure(figsize=(15, rows * 4))

for i, col in enumerate(numerical_columns):
    plt.subplot(rows, cols, i+1)
    sns.boxplot(y=data[col])
    plt.title(col)
    plt.xticks([])
plt.suptitle("Box Plots of Numerical Features", fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

Box Plots of Numerical Features



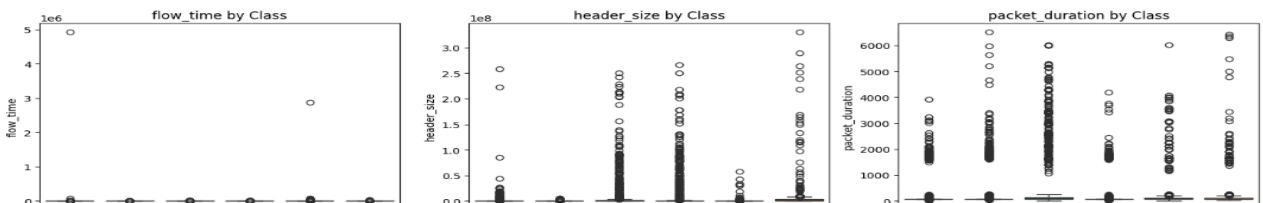
- Visualized Key Numerical features using the correlation matrix


Correlation Heatmap (Filtered: Only Significant Correlations)
header_size
packet_duration
overall_rate
src_rate
fin_packets
urg_packets
rst_packets
max_value
value_covariance
fin_flags
syn_flags
rst_flags
psh_flags
ack_flags
protocol_http
protocol_https
protocol_tcp
protocol_udp
protocol_icmp
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

Code:

```python
#Correlation heatmap

import numpy as np

corr_matrix = data[numerical_columns].corr()
abs_corr = corr_matrix.abs().copy()
np.fill_diagonal(abs_corr.values, 0)
threshold = 0.1

# Identify columns that have at least one correlation
cols_to_keep = abs_corr.columns[(abs_corr.max() >= threshold)]

#filter matrix
filtered_corr_matrix = corr_matrix.loc[cols_to_keep, cols_to_keep]
plt.figure(figsize=(18, 16))
sns.heatmap(filtered_corr_matrix,
            annot=True,
            fmt=".2f",
            cmap="coolwarm",
            cbar=True,
            linewidths=0.5,
            annot_kws={"size": 8})
plt.xticks(rotation=45, ha='right')
plt.title("Correlation Heatmap (Filtered: Only Significant Correlations)", fontsize=16)
plt.tight_layout()
plt.show()
```

- Visualized the relation between numerical features and the target using box plots



Code:

```
#Exploring relationship between numerical features and the target variable ("class")
cols = 3
rows = (len_of_num_cols // cols) + (len_of_num_cols % cols > 0)

plt.figure(figsize=(cols * 5, rows * 4))
for idx, col in enumerate(numerical_columns, 1):
    plt.subplot(rows, cols, idx)
    sns.boxplot(x='label', y=col, data=data, hue='label', palette='colorblind', dodge=False)
    plt.title(f'{col} by Class')
    plt.xlabel("Class")
    plt.ylabel(col)
    plt.legend([],[], frameon=False)

plt.tight_layout()
plt.show()
```

C. Feature Engineering

1. Group-Key Aggregations (optional)

When you pass a `group_key` (e.g. source IP), each of the following columns is aggregated within that group.

Base Column	Aggregations	Resulting Features
flow_time	mean, std, min, max	flow_time_mean, flow_time_std, flow_time_min, flow_time_max
header_size	mean, std	header_size_mean, header_size_std
packet_duration	mean, max	packet_duration_mean, packet_duration_max
fin_packets	sum	fin_packets_sum
urg_packets	sum	urg_packets_sum
rst_packets	sum	rst_packets_sum

syn_flags	sum	syn_flags_sum
ack_flags	mean	ack_flags_mean
protocol_http	sum	protocol_http_sum
protocol_https	sum	protocol_https_sum
label (if present)	mode	label_mode

Note: Group-level stats capture the context of each flow among its peers (e.g. burstiness per IP).

---

2. Per-Flow Derived Metrics

Always computed on each record (after any grouping).

Feature	Definition	Purpose
packet_rate	(fin_packets + urg_packets + rst_packets) / (packet_duration + ε)	Transactions per second
total_flags	fin_flags + syn_flags + rst_flags + psh_flags + ack_flags	Overall flag activity
flags_per_packet	total_flags / (fin_packets + urg_packets + rst_packets + ε)	Flag density per packet
syn_ack_ratio	syn_flags / (ack_flags + ε)	Half-open vs. completed handshakes
rate_asymmetry	(src_rate − dst_rate) / (src_rate + dst_rate + ε)	Throughput skew

header_to_payload_ratio	header_size / ((overall_rate / (packet_rate + ε)) + ε)	Header overhead vs. effective payload throughput

D. Feature Selection

Tried multiple different feature selection techniques including but not limited to:

1. Redundant Feature Elimination using Random Forest (RFE)
2. Mutual Information (MI)
3. Correlation Elimination
4. Variance Thresholding

Results indicated that **MI** and **RFE** were reducing our performance while **Variance Thresholding** and **Correlation Elimination** improved our performance. However, when applying any of them, predicting on **Kaggle's testing data** resulted in lower performance.

Code:

```
from sklearn.feature_selection import VarianceThreshold

vt = VarianceThreshold(threshold=0.01)
X_train_vt = vt.fit_transform(X_train)
selected_vt_cols = X_train.columns[vt.get_support()]

# Keep selected columns
X_train = pd.DataFrame(X_train_vt, columns=selected_vt_cols)
X_test = X_test[selected_vt_cols]  # align test set

print(f"Remaining features after Variance Thresholding: {len(X_train.columns)}")
```

```python
import numpy as np

corr_matrix = X_train.corr().abs()
upper_tri = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
to_drop = [column for column in upper_tri.columns if any(upper_tri[column] > 0.95)]

X_train = X_train.drop(columns=to_drop)
X_test = X_test.drop(columns=to_drop, errors='ignore')

# Ensure test set matches training columns
missing_cols = set(X_train.columns) - set(X_test.columns)
for col in missing_cols:
    X_test[col] = np.nan

X_test = X_test[X_train.columns]
X_test = X_test.reset_index(drop=True)
y_test = y_test["label"].reset_index(drop=True)

print("Removed redundant features (correlation-based):", to_drop)
print(f"Final number of features: {X_train.shape[1]}")
```

```python
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

rfe_model = RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=-1)
rfe = RFE(estimator=rfe_model, n_features_to_select=20)
rfe.fit(X_train, y_train)

#select the features
X_train_rfe = X_train.loc[:, rfe.support_]
X_test_rfe = X_test.loc[:, rfe.support_]

print(f"Features selected by RFE: {list(X_train_rfe.columns)}")

#train the model on the selected features
rfe_model.fit(X_train_rfe, y_train)

#predict
y_pred_rf = rfe_model.predict(X_test_rfe)

#evaluate
print("Random Forest Model Performance:")
print(f"Accuracy: {accuracy_score(y_test, y_pred_rf)}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred_rf))
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred_rf))
```

Why You Should Split Your Data Before Correcting Skewness

Executive Summary

Addressing skewness in your features is a critical preprocessing step for many machine learning models. However, fitting skew-correction transforms on the entire dataset *before* splitting into training and test sets introduces data leakage and inflates performance estimates. This report explains why you should always split your data first, then fit skewness-correction (and any other parameterized preprocessing) on the training set only, and finally apply the learned transformation to the test set.

1. Introduction

Proper validation of model performance relies on keeping the test data truly unseen during all aspects of model building, including preprocessing. Skewness—non-normal distributions with long tails—can impair many models' assumptions and convergence. Common corrections include logarithmic or square-root transforms, as well as parameterized methods like Box–Cox or Yeo–Johnson. When these transforms estimate parameters from the data, fitting them on the full dataset risks contaminating your evaluation.

2. Understanding Skewness and Its Correction

Skewness measures asymmetry in a variable's distribution. Positive skew (right tail) often appears in financial or count data; negative skew (left tail) appears in bounded scales. **Data-driven transforms** (e.g., Box–Cox, Yeo–Johnson) estimate one or more parameters (λ) that best normalize the distribution.

Fixed transforms (e.g., log(x + 1), square-root) do not estimate parameters and therefore do not introduce leakage—but including them in a train-fitted pipeline promotes consistency.

3. The Risk of Data Leakage

Data leakage occurs when information from the test set is inadvertently used during training. Fitting skew-correction on the entire dataset before splitting leaks statistics (such as the optimal λ) from the test set into the model. Consequences include:

Biased parameter estimates: The skew-correction becomes tuned to all available data, making training artificially easier.

Overoptimistic performance: Evaluation on "leaked" test data underestimates real-world error.

Reduced reproducibility: Future data may exhibit different skew patterns; relying on test-derived statistics hides this variability.

4. Best Practice: Split First, Then Fit

1. **Split the dataset** into training and test subsets (commonly an 80/20 or 70/30 ratio). 2. **Fit parameterized preprocessing** (e.g., estimating λ for Box–Cox) *only* on the training subset.
3. **Apply the learned transformation** to the test subset using the parameters derived from training.
4. **Train and evaluate** your model on the transformed training and test sets, respectively.

This workflow ensures a fair, unbiased evaluation and preserves the integrity of the test set as a proxy for unseen data.

5. Practical Example (Conceptual)

Data split: Reserve 20 % of the data as a hold-out test set.

Pipeline construction: Create a preprocessing sequence that includes skew-correction followed by any additional steps (e.g., scaling, encoding).

Pipeline fitting: Estimate skew-correction parameters using only the training data. **Transformation application:** Use the fitted pipeline to transform both training and test data.

Model training and evaluation: Train the model on the transformed training set, then evaluate on the transformed test set.

This approach keeps the test data unseen until the very last step, yielding realistic performance metrics.

6. Conclusion

Splitting your dataset *before* fitting skew-correction or any other parameterized preprocessing prevents data leakage, yields reliable performance estimates, and ensures your model generalizes effectively to new, unseen data. Adopting this discipline in every machine learning workflow promotes scientific rigor and robust model validation.

Proper Treatment of Test Data in SMOTE Workflows

*When addressing imbalanced datasets with SMOTE on the training portion, your test set must remain a faithful representation of real-world data. Below is a focused guide on **why** you cannot apply SMOTE to test data, **what** you can and cannot do to it, and **why** you should leave it unchanged.*

Why You Cannot Apply SMOTE to Test Data

Data Leakage: Generating synthetic samples on test data lets the model indirectly "see" test instances during training, leading to inflated and untrustworthy performance metrics. **Unrealistic Evaluation**: A balanced test set no longer reflects the natural class distribution your model will encounter in production, masking shortcomings on minority classes.

What You Cannot Do on the Test Set

No Oversampling/Undersampling: Avoid SMOTE, random oversampling, or undersampling.

No Refitting: Do not re-fit encoders, scalers, PCA, or any preprocessing on test features or labels.

No Peeking: Do not use test labels or feature statistics during training, feature selection, or hyperparameter tuning.

What You Can Do Instead

1. **Label Encoding Consistency**
   Apply the label encoder and feature transformers (scaler, PCA, etc.) fitted on training data— without refitting—to the test set.
2. **Stratified Split**
   At the data-splitting stage (e.g., 80/20), use stratified sampling to ensure the test set contains representative proportions of each class.
3. **Alternative Evaluation Strategies**
   Report class-wise metrics (precision, recall, F1‑score) rather than only overall accuracy. Use ROC AUC and Precision–Recall curves that are robust to imbalance. Consider confidence intervals or bootstrapping on test results for statistical reliability.

Should You Leave the Test Set As‑Is?

Yes. The test set should be left in its original, imbalanced state to serve as a true holdout:

1. **Preserves Realism**
   Mimics the operational environment's class distribution.
2. **Ensures Fair Comparison**
   Allows unbiased comparison between models and hyperparameter settings.
3. **Reveals True Generalization**
   Highlights strengths and weaknesses in handling minority classes without artificial aid.

Bottom Line: Restrict SMOTE strictly to your training data. On the test side, apply only transformations derived from training, leave the class imbalance intact, and leverage stratified sampling and robust metrics to assess model performance honestly.

E. Model Implementation

1. XGBoost (Extreme Gradient Boosting)

Why:

- Highly efficient boosted tree algorithm.
- Handles missing data, outliers, and imbalance well.
- Built-in regularization which helps avoid overfitting.
- Best for tabular data in most Kaggle competitions. (we're practicing)

Use Case:

- Data is already preprocessed and balanced which is ideal for XGBoost.
- Pushing for high accuracy, while managing generalization.

Implementation:

The final model was fine-tuned using a combination of manual experimentation and optimization techniques. Below is a detailed explanation of each hyperparameter and the rationale behind its chosen value.

- **learning_rate = 0.18296**: A moderately low learning rate allows the model to learn gradually and generalize better.
- **max_depth = 16**: A deeper tree enables capturing complex patterns in the data for multi-class classification.
- **subsample = 0.98218**: Using 98% of the data per tree introduces slight randomness to reduce overfitting.
- **colsample_bytree = 0.87002**: Sampling 87% of the features per tree increases model robustness and prevents feature dominance.
- **gamma = 0.18618**: A small gamma value regularizes tree splits by requiring meaningful gain to split further.
- **reg_alpha = 0.45208**: L1 regularization adds sparsity to the model, helping reduce overfitting.
- **reg_lambda = 5.87987**: L2 regularization smooths weights and stabilizes model learning.
- **min_child_weight = 2**: Prevents the model from creating nodes that represent too few instances, improving generalization.
- **n_estimators = 400**: A larger number of trees helps compensate for the slower learning rate.

- **tree_method = 'gpu_hist'**: Enables fast histogram-based tree building using GPU acceleration.
- **device = 'gpu'**: Utilizes GPU resources to speed up training.
- **objective = 'multi:softprob'**: Outputs class probabilities for multi-class classification tasks.
- **use_label_encoder = False**: Disables deprecated automatic label encoding for compatibility.
  eval_metric = 'logloss': Uses log loss to evaluate the quality of predicted class probabilities.

```
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score

improved_params = {
    'learning_rate':     0.18295938520166618,
    'max_depth':         16,
    'subsample':         0.9821770843130437,
    'colsample_bytree':  0.8700184979100425,
    'gamma':             0.18617821128570886,
    'reg_alpha':         0.452075050028747287,
    'reg_lambda':        5.879866631339201,
    'min_child_weight':  2,
    'n_estimators':      400,
    'tree_method':      'gpu_hist',
    'device':           'gpu',
    'objective':        'multi:softprob',
}
xgb_model = XGBClassifier(
    **improved_params,
    random_state=42,
    use_label_encoder=False,
    eval_metric='logloss'
)
xgb_model.fit(X_train_scaled, y_train)

y_pred_xgb = xgb_model.predict(X_test_scaled)
acc_xgb = accuracy_score(y_test, y_pred_xgb)
print(f"XGBoost Accuracy: {acc_xgb:.16f}")
```

2. Random Forest

Why:

- Handles high-dimensional data well (important post one-hot encoding).
- Not sensitive to feature scaling.
- Deals well with imbalanced datasets through bootstrapping (especially with class weighting or balanced subsampling).
- Provides feature importance, helping analyze your features further.

Use Case:

- Dataset has a mix of encoded categorical and numerical features.
- It's robust, interpretable, and resistant to overfitting due to ensembling.

Implementation:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train_scaled, y_train)
y_pred_rf = rf_model.predict(X_test_scaled)
acc_rf = accuracy_score(y_test, y_pred_rf)

print(f"Random Forest Accuracy: {acc_rf:.4f}")
```

3. LightGBM

Why:

- High performance and speed, especially on large datasets.
- Handles both categorical and numerical features natively.
- Built-in support for missing values and categorical encoding.

Challenges:

- Requires careful tuning to avoid overfitting.
- Less interpretable than single decision trees.

Use Case:

- Ideal for tabular data with many features and rows.
- Effective even with default settings, but fine-tuning improves results.

Implementation:

objective = "multiclass": Targets multiclass classification by predicting probabilities for each class.

num_class = len(set(y_train)): Automatically sets the number of output classes based on training labels.

learning_rate = 0.02: A small learning rate ensures gradual updates and better generalization.

n_estimators = 500: A larger number of trees compensates for the smaller learning rate.
max_depth = 9: Limits tree depth to prevent overfitting while still capturing important patterns.

num_leaves = 63: Controls the complexity of each tree and works in tandem with max_depth for regularization.

subsample = 0.9: Trains each tree on 90% of the data to add randomness and reduce overfitting.

colsample_bytree = 0.9: Uses 90% of features per tree to promote diversity among trees.

min_child_samples = 10: Ensures that leaf nodes have at least 10 data points to reduce noise.

reg_alpha = 0.05: Adds slight L1 regularization to encourage model sparsity.

reg_lambda = 0.5: Applies L2 regularization to smooth leaf weights and prevent overfitting.

eval_metric = "multi_logloss": Uses log loss, a suitable metric for probabilistic multi-class classification.

```
from lightgbm import LGBMClassifier
lgb_clf = {
    "device": "cpu",
    "objective": "multiclass",
    "num_class": len(set(y_train)),
    "learning_rate": 0.02,
    "n_estimators": 500,
    "max_depth": 9,
    "num_leaves": 63,
    "subsample": 0.9,
    "colsample_bytree": 0.9,
    "min_child_samples": 10,
    "reg_alpha": 0.05,
    "reg_lambda": 0.5,
    "random_state": 42,
    "eval_metric": "multi_logloss",
    "verbose": -1
}
lgbm_model = LGBMClassifier(**lgb_clf)
lgbm_model.fit(X_train_scaled, y_train)
y_pred_lgbm = lgbm_model.predict(X_test_scaled)
acc_lgbm = accuracy_score(y_test, y_pred_lgbm)
print(f"LightGBM Accuracy: {acc_lgbm:.16f}")
```

4. Decision Tree

Why:

- Easy to understand.
- Handles both categorical and numerical features.
- No need for scaling or dummy encoding (though we've already encoded which is still fine).

Challenges:

- Prone to overfitting.
- Lower performance than ensemble models.

Use Case:

- It's a transparent model that gives insight into the data.
- Acts as a base learner for other models (like Random Forest and XGBoost).

Implementation:

```
from sklearn.tree import DecisionTreeClassifier
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train_scaled, y_train)
y_pred_dt = dt_model.predict(X_test_scaled)
acc_dt = accuracy_score(y_test, y_pred_dt)
print(f"Decision Tree Accuracy: {acc_dt:.16f}")
```

5. Gradient Descent

Why:

- Easy to understand and implement.
- Works well with both categorical and numerical features (after encoding).
- Forms the core of many machine learning algorithms, including linear and logistic regression.

Challenges:

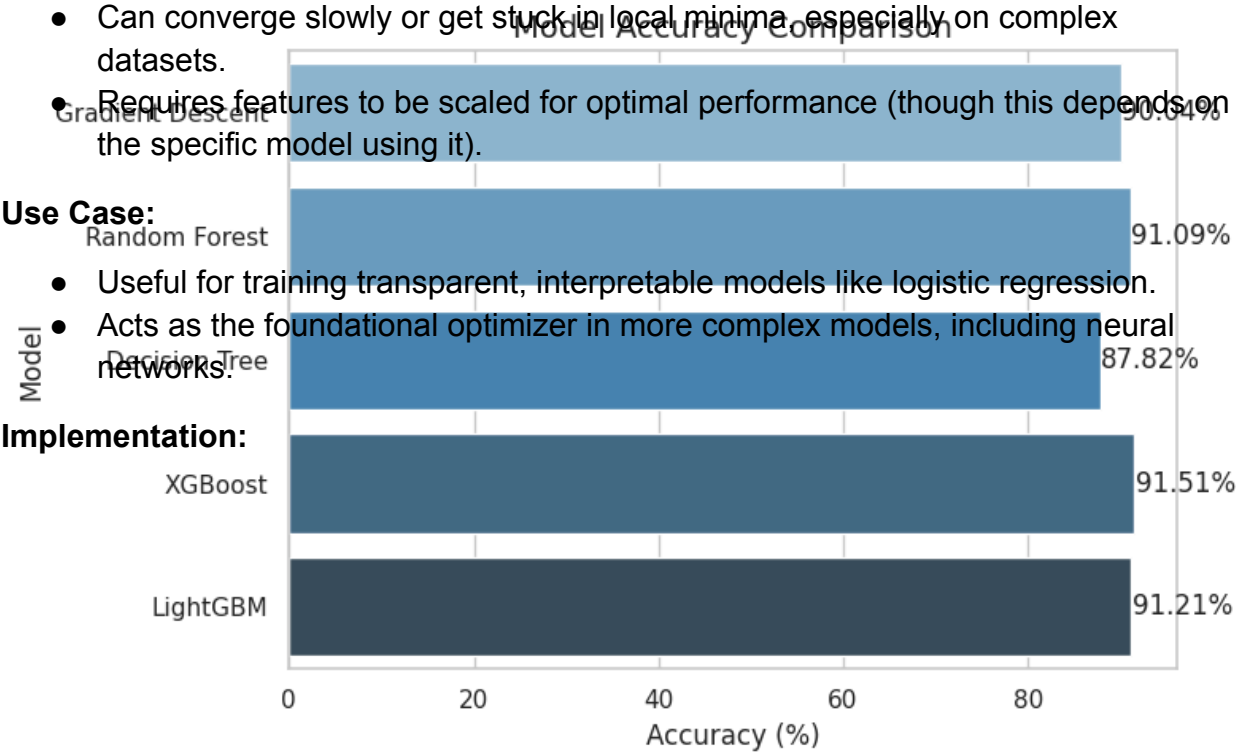
- Sensitive to learning rate selection.

- Can converge slowly or get stuck in local minima, especially on complex datasets.
- Requires features to be scaled for optimal performance (though this depends on the specific model using it).

Use Case:

- Useful for training transparent, interpretable models like logistic regression.
- Acts as the foundational optimizer in more complex models, including neural networks.

Implementation:




```
y_pred_gb = gb_model.predict(X_test_scaled)
acc_gb = accuracy_score(y_test, y_pred_gb)
print(f"Gradient Boosting Accuracy: {acc_gb:.4f}")
```

E. Model Evaluation

Accuracy Scores:

Gradient Boosting: 90.04%

Random Forest: 91.089%
Decision Tree: 87.82%
XGBoost: 91.51%
LightGBM: 91.21%

F. Conclusion

In summary, our Phase 2 efforts have produced a lean yet powerful pipeline for multi-class cyber-attack detection. After initially experimenting with deduplication and outlier removal, our final best accuracy was achieved without handling duplicates or outliers, focusing instead on targeted feature enrichment and selection.

We added six new aggregated features, clipped lower-bound values to guard against extreme lows, and applied label encoding for categorical protocols.

A QuantileTransformer (which performed on par with a Robust Scaler) normalized distributions, and we used a variance threshold of 0.01 on the target's features, dropping only one column and retaining 26. We then trained five models (Random Forest, XGBoost, Gradient Boosting, Decision Tree, and LightGBM), each with hyperparameters refined via random search, grid search, and manual tuning, reaching a peak test accuracy of 91.51 % with GPU-accelerated XGBoost.

We also tested a stacking ensemble (LightGBM, Random Forest, XGBoost base learners with an XGBoost meta-model), but it underperformed the standalone XGBoost and was therefore dropped.