CSAI 253 PROJECT

TEAM MEMBERS

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

Team:
Machine Not Learning

Team Email:
s-amr.anwar@zewailcity.edu.eg

TASKS WORKLOAD

TASK	DONE BY
Data Loading & Cleaning	All
EDA: Visualize data distributions using histograms, box plots, and scatter plots.	Amr Yasser
EDA: Identify correlations between features.	Amr Yasser
EDA: Explore potential relationships between features and the target variable.	Amr Yasser
FE: Create new features or transform existing ones if necessary.	Amr Yasser
FE: Handle categorical variables using appropriate encoding techniques.	Amr Mahmoud
FE: Choose 20 most important features to work with.	Momen
FE: Scale numerical features if required.	Aya
MI: K-Nearest Neighbors (KNN) & evaluate its performance & compare it with different models.	Aya
MI: Logistic Regression & evaluate its performance & compare it with different models.	Amr Yasser
MI: SVM & evaluate its performance & compare it with different models.	Amr Mahmoud
MI: Random Forest & evaluate its performance & compare it with different models.	Momen
Stacking Ensembling technique	Momen

1. Problem statement

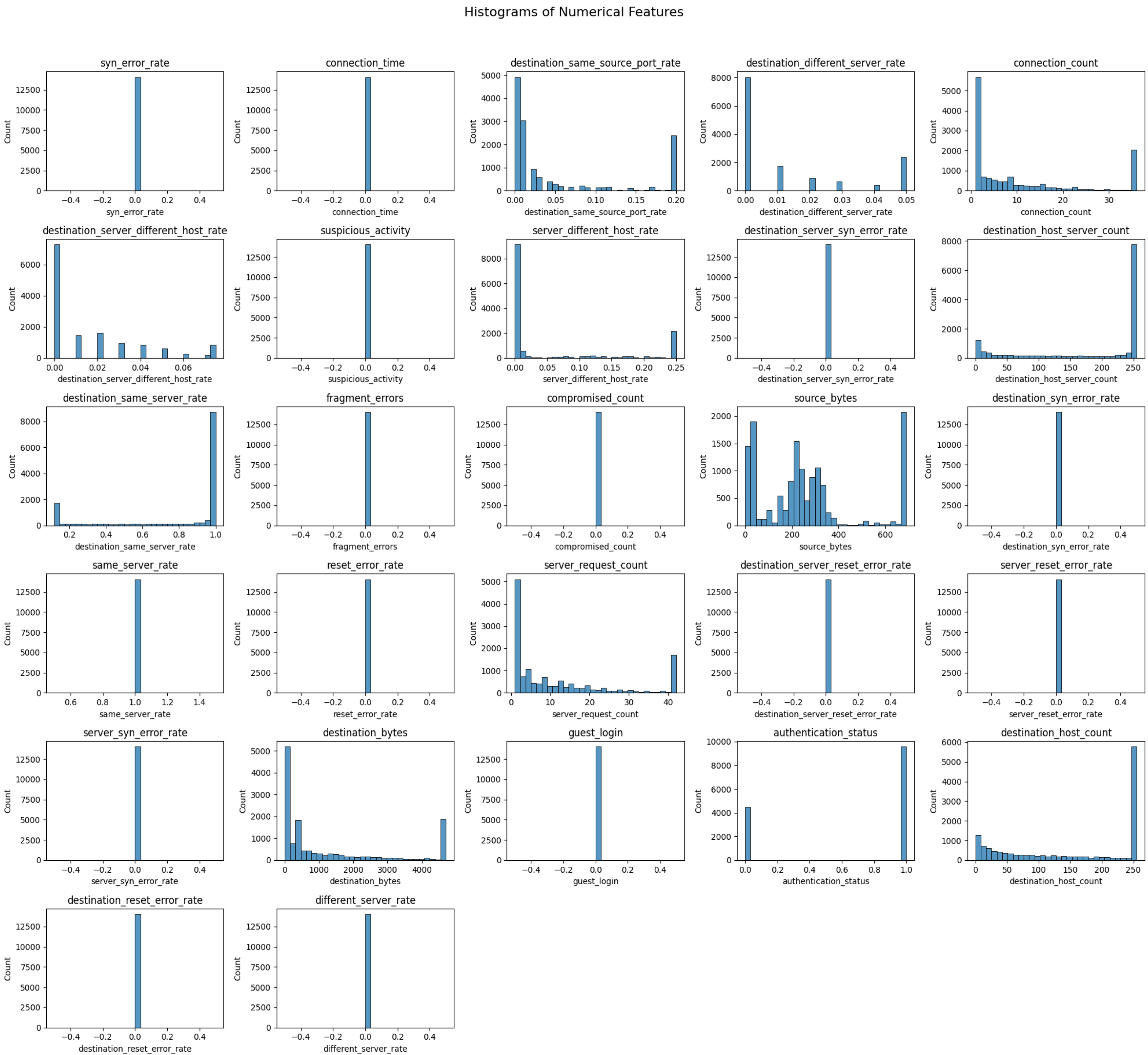
Network intrusion detection:

- **Objective:** Our objective is to develop four different machine learning models capable of classifying network connections as either normal or anomalous. Anomalous connections reveal malicious activities and potential threats.
- **Significance:** Detecting anomalous connections is crucial for protecting military networks from intrusions and cyberattacks.

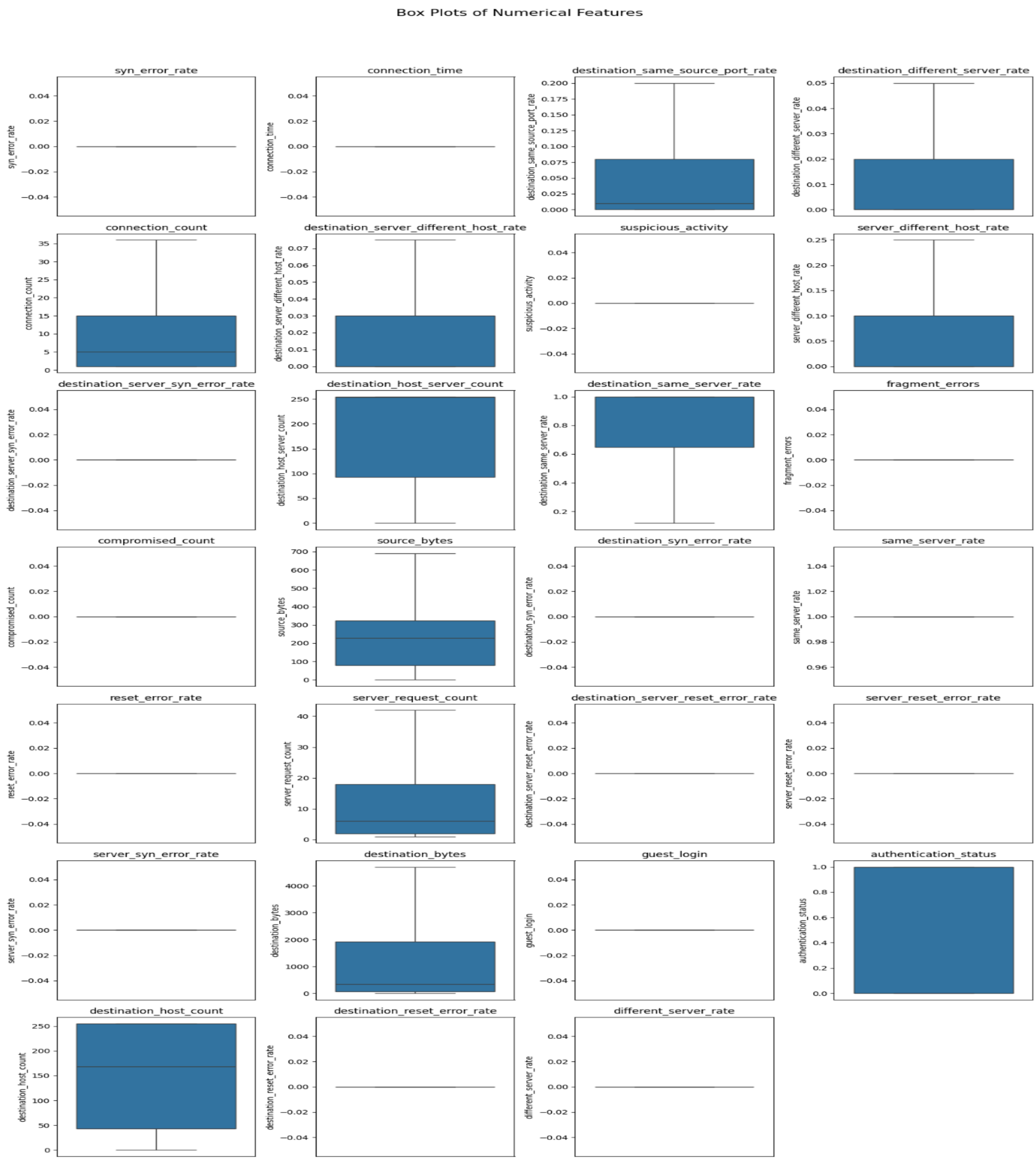
2. Data Exploration

Exploratory Data Analysis (EDA)

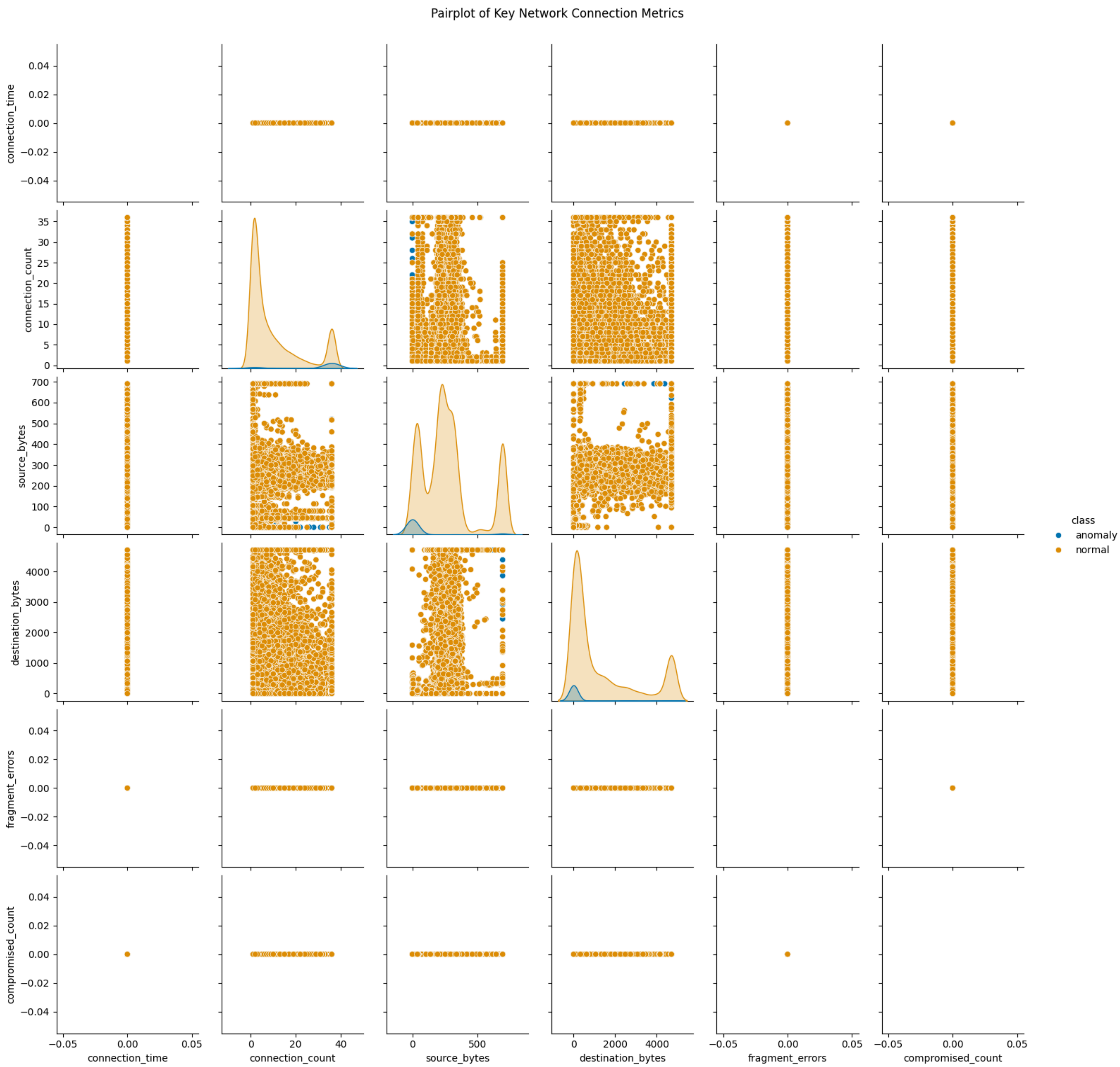
- Visualized **numerical** features using histograms



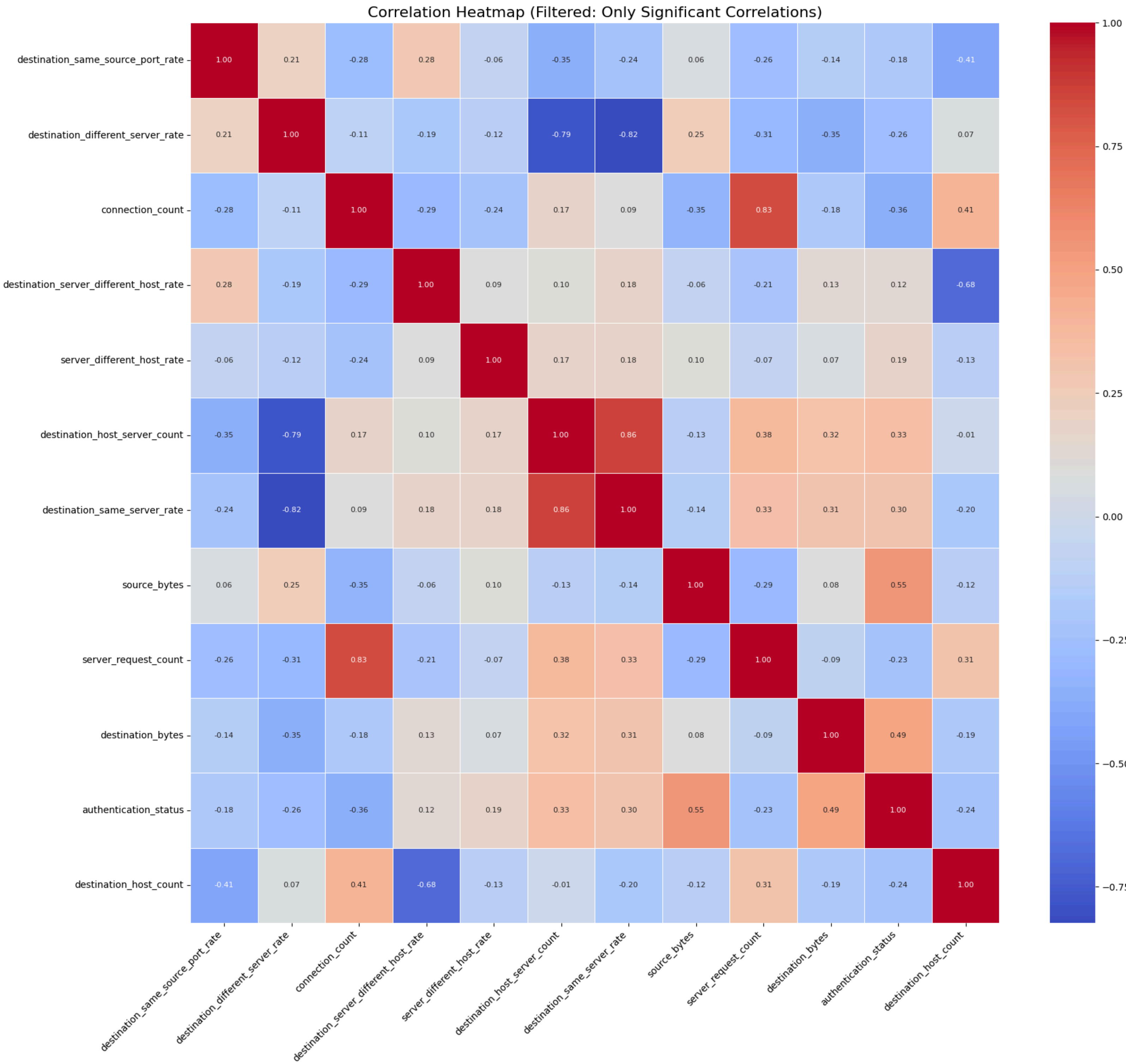
- Visualized **numerical** features using boxplots



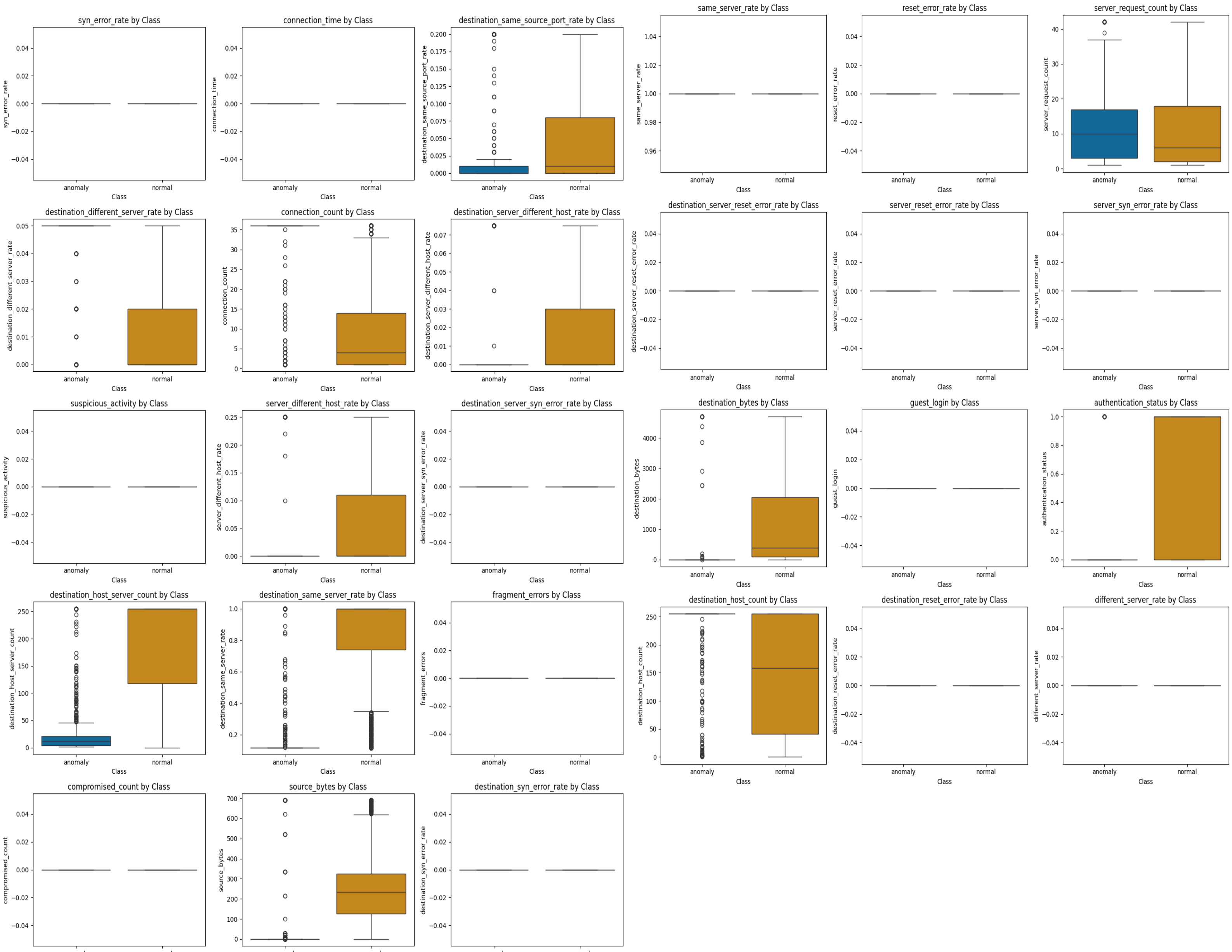
- Visualized key **numerical** features’ relation using scatter/pair plots



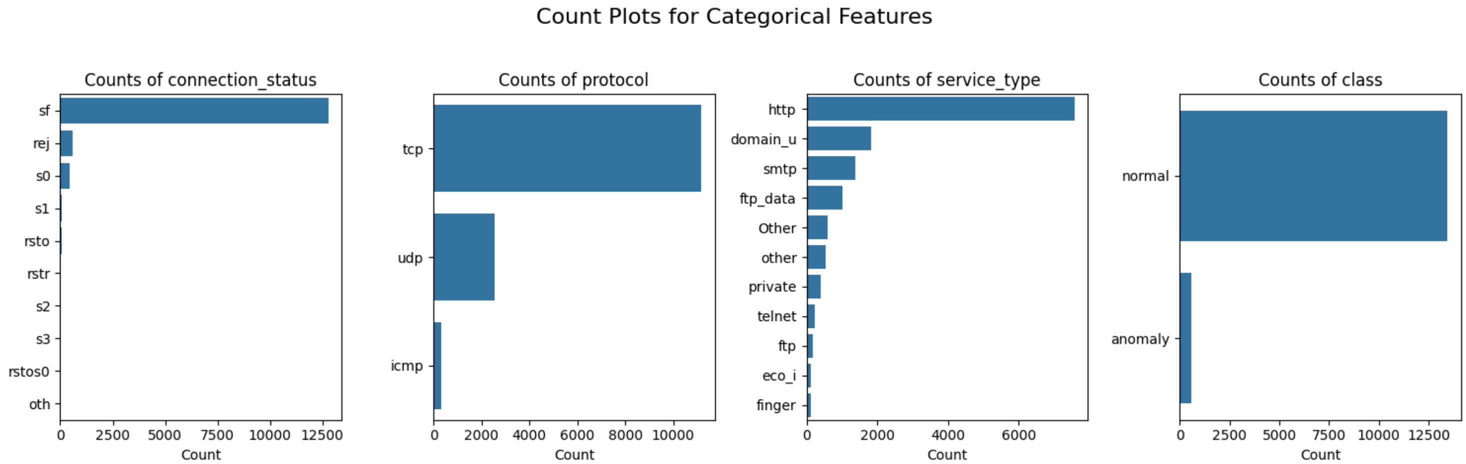
- Visualized the key **numerical** features’ relation using the correlation matrix



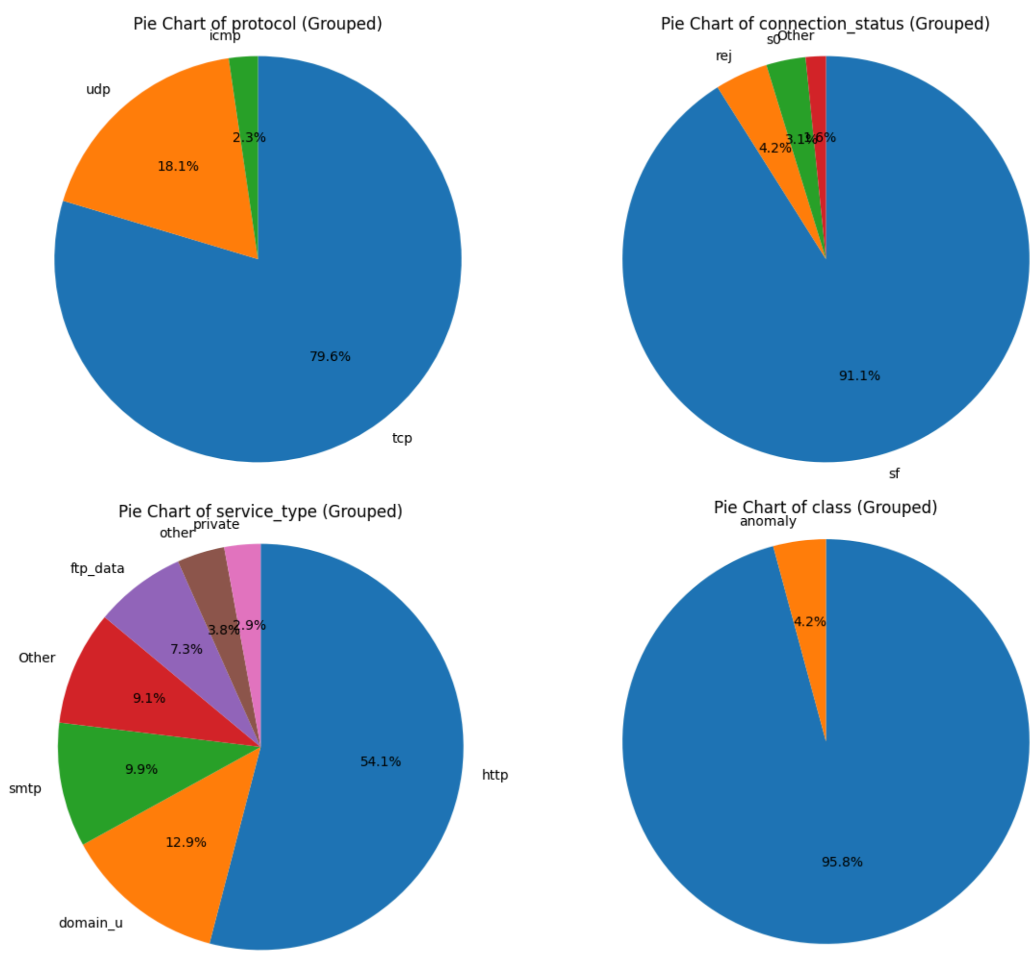
- Visualized the relation between **numerical features** and the **target** using box plots



- Visualized **categorical** features using count plots



- Visualized **categorical** features using pie charts

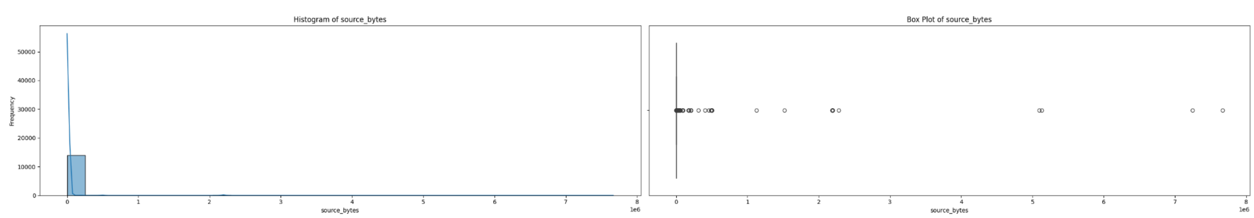


Summarize key insights from the exploratory data analysis.

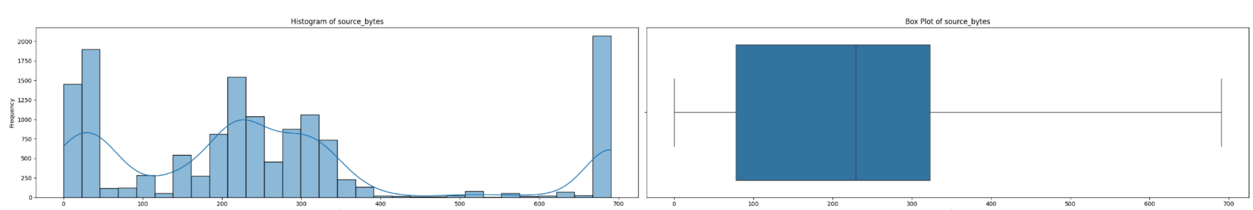
Data Preprocessing

Data Handling:

- **Missing values:** Dataset contained no missing values
- **Columns**: Divided columns into numerical and categorical
- **Target Variable:** class
- **Validations**: Validated the protocol column to ensure it only contains a single valid element and not a list of elements
- **Outliers:** Detected outliers using **Interquartile Range (IQR)**, then handled them using **winsorization algorithms.** We compared before and after handling them using histograms and boxplots, for example, with the **source_bytes** column:
    - **Before:**



    - **After:**



- **Inconsistencies:** Fixed inconsistencies by grouping similar columns into rate, count, binary, and categorical.
  - **Rate:** We ensured all rates are between 1 and 0 (`.clip(lower=0, upper=1`)
  - **Count:** We ensured all counts are a positive number (`>= 0`)
  - **Binary:** We ensured all binary is either 1 or 0 (`.isin([0, 1])`)
  - **Categorical:** We ensured all categoricals are lowercase and removed any unnecessary spaces (`.str.lower().str.strip()`)

Feature Engineering:

- **New Features**: We created 3 new features:
  - **Aggregated Error Rate:** This feature is calculated as the mean of four error-related columns, providing a consolidated metric for overall network error behavior.
  - **Log Transformations:** The `np.log1p` function is used on source_bytes and destination_bytes to mitigate skewness and make these features more suitable for modeling.
  - **Connection Intensity:** This derived feature captures the rate of connections relative to the connection time, potentially highlighting abnormal activity.

- **Categorical Features Splitting:**

  We separated the categorical features into:

  - **X:** All features **without** target class
  - **y:** The **target class**

  Then we used `train_test_split` to split them into:

  - **X_train, y_train:** 80% of the data was used for training the models
  - **X_test, y_test:** 20% of the data was used for testing the models

- **Encoding Techniques:**
  - **Label Encoding**
    - Applied to categorical features (connection_status, protocol, service_type) using sklearn's LabelEncoder
    - Created a separate encoder instance for each categorical feature to maintain independent mapping
    - Stored all encoders in a dictionary for potential inverse transformation or future reference
    - Handled unknown values in test set by replacing them with the most frequent value from the training set
    - Performed mutual information feature selection before final encoding to ensure only informative features are retained
    - Encoded the target variable 'class' for classification using LabelEncoder().fit_transform()

- Applied the same trained encoders to both training and test sets to ensure consistent transformation
- Used feature selection based on mutual information to identify and select only relevant categorical features

```
#
#   Apply  Label  Encoding
#
X_train_encoded = X_train.copy()
X_test_encoded = X_test.copy()

label_encoders = {}
for feature in categorical_features:
    model = LabelEncoder()
    X_train_encoded[feature] = model.fit_transform(X_train[feature])

    # handle potential unknown values
    X_test_feature = X_test[feature].copy()
    for unknown_value in set(X_test_feature) - set(X_train[feature]):
        X_test_feature.replace(unknown_value, X_train[feature].mode()[0], inplace=True)

    X_test_encoded[feature] = model.transform(X_test_feature)
    label_encoders[feature] = model

# -----------------------------------
```

- **& One-Hot encoding:**
  - Used **label encoding** with the target **class**, and then fit it on the training target using `.fit_transform(y_train)`
  - Used **One-Hot encoding** with the **rest of the features,** and then fit it on the training data using `.fit_transform(X_train[categorical_features])`
  - Gave the new encoded features meaningful names using `.get_feature_names_out(categorical_features)`, and then turned them into **DataFrames** so that they have column names and row indexes to match the original data.
  - Made sure X_train and X_test have the **same columns** by identifying any missing columns and adding them to X_test with all **values set to 0**, and then we **reordered** X_test to match X_train.

```python
#one hot encoding
one_hot_encoder = OneHotEncoder(sparse_output=False, drop='first', handle_unknown='ignore')

X_train_encoded = one_hot_encoder.fit_transform(X_train[categorical_features])
X_test_encoded = one_hot_encoder.transform(X_test[categorical_features])

encoded_feature_names = one_hot_encoder.get_feature_names_out(categorical_features)

X_train_encoded_df = pd.DataFrame(X_train_encoded, columns=encoded_feature_names, index=X_train.index)
X_test_encoded_df = pd.DataFrame(X_test_encoded, columns=encoded_feature_names, index=X_test.index)

#make sure x test and x train have same columns
missing_cols = set(X_train_encoded_df.columns) - set(X_test_encoded_df.columns)
for col in missing_cols:
    X_test_encoded_df[col] = 0  #add missing
```

- **Normalization:**
  - We normalized the data in **Non Tree-based** models using `StandardScaler` to ensure that all features contribute equally, as Non Tree-based models are sensitive to the range of the features.
  - **Not** normalizing the data in **Tree-based** models is better for efficiency since they rank each row either way whether it's between a range or not.

- **Feature Selection:**
  - In order to choose the 20 most important features, we compared trained an initial Random Forest model to use Recursive Feature Elimination (RFE) and extract the desired features:

['source_bytes', 'connection_intensity', 'log_source_bytes', 'connection_status', 'connection_count', 'destination_same_server_rate', 'destination_host_count', 'destination_host_server_count', 'log_destination_bytes', 'destination_same_source_port_rate', 'server_request_count', 'destination_different_server_rate', 'destination_bytes', 'service_type', 'destination_server_different_host_rate', 'protocol', 'authentication_status', 'server_different_host_rate', 'connection_time', 'syn_error_rate']

- **Numerical Features Scaling:**
  - Before scaling there was 544 **(Anomaly)**, while on the other hand, the **(Normal)** was 13800.
  - We used `SMOTE` to upscale the minority features **(Anomaly)** in our target column "**class".**
  - The final result 10750 **(Anomaly)**, 10750 **(Normal)**, showing that we now have a balanced column resulting in no bias when training.

- **Feature Selection Techniques:**

Due to the nature of One-Hot encoding, many new columns are added, to combat this, we used each of the following Feature Selection Techniques:

- Redundant Feature Elimination (RFE)
- Correlation Matrix Selection
- Variance Thresholding
- SMOTE

```
from sklearn.ensemble import RandomForestClassifier

# Use label-encoded training features and target
X = X_train1.copy()
y = y_train1.copy()

# Train Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X, y)

# Get feature importances
importances = rf_model.feature_importances_
feature_names = X.columns

# Create DataFrame and select top 20
importance_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': importances
}).sort_values(by='Importance', ascending=False)

top_20_features = importance_df.head(20)['Feature'].tolist()

# Filter train/test sets
X_train = X_train1[top_20_features].copy()
X_test = X_test[top_20_features].copy()

print("Top 20 selected features:")
print(top_20_features)
```

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_scaled_df, y_train1)
```

```
from sklearn.feature_selection import VarianceThreshold

var_thresh = VarianceThreshold(threshold=0.01)
X_train_selected = var_thresh.fit_transform(X_train_resampled)
X_test_selected = var_thresh.transform(X_test_scaled_df)
selected_feature_names = X_train_scaled_df.columns[var_thresh.get_support()]

X_train_selected_df = pd.DataFrame(X_train_selected, columns=selected_feature_names)
X_test_selected_df = pd.DataFrame(X_test_selected, columns=selected_feature_names)

X_test_selected_df = X_test_selected_df.reset_index(drop=True)
y_test = pd.Series(y_test).reset_index(drop=True)

#final train and test datasets
data_train_final = pd.concat([X_train_selected_df, pd.Series(y_train_resampled, name='class')], axis=1)
data_test_final = pd.concat([X_test_selected_df, pd.Series(y_test, name='class')], axis=1)
```

Model Selection and Implementation

Model Selection and Implementation:

1. **Random Forest:**
2. **KNN:**
3. **SVM:**
4. **Logistic Regression:**
5. **Decision Tree:**
6. **XGBoost:**

1. **Random Forest**

Why:

- Handles high-dimensional data well (important post one-hot encoding).
- Not sensitive to feature scaling.
- Deals well with imbalanced datasets (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, classification_report, confusion_matrix
from sklearn.utils.class_weight import compute_class_weight


class_weights = compute_class_weight(class_weight='balanced', y=data_train_final['class'])
weights = dict(zip([0,1], class_weights))

#train random forest
rf_model = RandomForestClassifier(class_weight=weights, random_state=42)
rf_model.fit(data_train_final.drop(columns='class'), data_train_final['class'])

#predict
y_pred_rf = rf_model.predict(data_test_final.drop(columns='class'))

#calculate
print("Random Forest Model Performance:")
print(f"Accuracy: {accuracy_score(data_test_final['class'], y_pred_rf)}")
print("\nClassification Report:")
print(classification_report(data_test_final['class'], y_pred_rf))
print("\nConfusion Matrix:")
print(confusion_matrix(data_test_final['class'], y_pred_rf))
```

2. K-Nearest Neighbors (KNN)

Why:

- Simple, intuitive baseline.
- Makes no assumptions about the data distribution.
- Learns non-linear patterns well (especially in clean datasets).

Challenges:

- Sensitive to high dimensionality (due to one-hot encoding).
- Needs feature scaling (like StandardScaler).
- Slower with large datasets.

Use Case:

- Good benchmark to compare how more complex models perform.
- Can help identify if local patterns exist in your feature space.

Implementation:

```
from sklearn.neighbors import KNeighborsClassifier

#train KNN
knn_model = KNeighborsClassifier()
knn_model.fit(data_train_final.drop(columns='class'), data_train_final['class'])

#predict
y_pred_knn = knn_model.predict(data_test_final.drop(columns='class'))

#calculate
print("KNN Model Performance:")
print(f"Accuracy: {accuracy_score(data_test_final['class'], y_pred_knn)}")
print("\nClassification Report:")
print(classification_report(data_test_final['class'], y_pred_knn))
print("\nConfusion Matrix:")
print(confusion_matrix(data_test_final['class'], y_pred_knn))
```

3. Support Vector Machine (SVM)

Why:

- Effective in high-dimensional spaces.
- Works well for clear margin of separation between classes.
- Powerful for non-linear classification with kernel tricks.

Challenges:

- Not scalable for very large datasets.
- Needs careful tuning (kernel, C, gamma).
- Requires feature scaling.

Use Case:

- My dataset is well-prepared, and relatively balanced after SMOTE.
- A powerful classifier that's sensitive to decision boundaries.

Implementation:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC


#grid Search for Hyperparameters
param_grid = {
    'C': [0.1, 1, 5, 7, 10],              # try more values for finer tuning
    'gamma': ['scale', 0.01, 0.001],       # 'scale' is usually a good default
    'kernel': ['rbf', 'linear', 'poly']    # try different kernel types
}

svm = SVC(random_state=42)

grid_search = GridSearchCV(estimator=svm, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1, verbose=1)

grid_search.fit(data_train_final.drop(columns='class'), data_train_final['class'])
best_svm_model = grid_search.best_estimator_

#predict with best model
y_pred_svm = best_svm_model.predict(data_test_final.drop(columns='class'))

#evaluate
print("SVM Model Performance:")
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Accuracy: {accuracy_score(data_test_final['class'], y_pred_svm)}")
print("\nClassification Report:")
print(classification_report(data_test_final['class'], y_pred_svm))
print("\nConfusion Matrix:")
print(confusion_matrix(data_test_final['class'], y_pred_svm))
```

4. Logistic Regression

Why:

- Fast, interpretable, and mathematically elegant.
- Ideal if your classes are linearly separable (or nearly so).
- Works great with dummy or one-hot encoding.

Challenges:

- Can underperform if data is non-linear.
- Assumes independent features.

Use Case:

- It's a baseline model.
- Useful for checking how far complex models go beyond linear separation.

Implementation:

```
from sklearn.linear_model import LogisticRegressionCV

#train logistic regression
logreg_model = LogisticRegression(random_state=42)
logreg_model.fit(data_train_final.drop(columns='class'), data_train_final['class'])

#predict
y_pred_logreg = logreg_model.predict(data_test_final.drop(columns='class'))

#calculate
print("Logistic Regression Model Performance:")
print(f"Accuracy: {accuracy_score(data_test_final['class'], y_pred_logreg)}")
print("\nClassification Report:")
print(classification_report(data_test_final['class'], y_pred_logreg))
print("\nConfusion Matrix:")
print(confusion_matrix(data_test_final['class'], y_pred_logreg))
```

5. 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

#train decision tree
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(data_train_final.drop(columns='class'), data_train_final['class'])

#predict
y_pred_dt = dt_model.predict(data_test_final.drop(columns='class'))

#calculate
print("Decision Tree Model Performance:")
print(f"Accuracy: {accuracy_score(data_test_final['class'], y_pred_dt)}")
print("\nClassification Report:")
print(classification_report(data_test_final['class'], y_pred_dt))
print("\nConfusion Matrix:")
print(confusion_matrix(data_test_final['class'], y_pred_dt))
```

6. 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:

```
import xgboost as xgb

classes = np.array([0, 1])
class_weights = compute_class_weight(class_weight='balanced', classes=classes, y=data_train_final['class'])
weights = dict(zip(classes, class_weights))

sample_weights = data_train_final['class'].map(weights)

#train
xgb_model = xgb.XGBClassifier(
    objective='binary:logistic',
    use_label_encoder=False,
    eval_metric='logloss',
    random_state=42
)

xgb_model.fit(
    data_train_final.drop(columns='class'),
    data_train_final['class'],
    sample_weight=sample_weights
)

#predict
y_pred_xgb = xgb_model.predict(data_test_final.drop(columns='class'))

#calculate
print("XGBoost Model Performance:")
print(f"Accuracy: {accuracy_score(data_test_final['class'], y_pred_xgb)}")
print("\nClassification Report:")
print(classification_report(data_test_final['class'], y_pred_xgb))
print("\nConfusion Matrix:")
print(confusion_matrix(data_test_final['class'], y_pred_xgb))
```

Model Evaluation (One-Hot Encoding)

Random Forest: Random forest **best** performed with an overall **accuracy of 0.9979**

1. The **classification report** showed that:

- **Class 0** had: **Precision=1**. **Recall= 0.95,** and **F1-score= 0.97**
- **Class 1** had: **Precision=1, Recall= 1**, and **F1-score= 1**

2. The **Confusion Matrix** showed that:

- True Negatives**(TN) = 111,** True Positives**(TP) = 2691**
- False Negatives**(FN) = 0,** False Positives**(FP) = 6**

Output:

```
Random Forest Model Performance:
Accuracy: 0.9978632478632479

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.95      0.97       117
           1       1.00      1.00      1.00      2691

    accuracy                           1.00      2808
   macro avg       1.00      0.97      0.99      2808
weighted avg       1.00      1.00      1.00      2808


Confusion Matrix:
[[ 111    6]
 [   0 2691]]
```

KNN: The K-Nearest Neighbors (KNN) model achieved an overall **accuracy of 0.9954**.

1. The classification report showed that:

- **Class 0** had: **Precision = 0.93**, **Recall = 0.97**, and **F1-score = 0.95**
- **Class 1** had: **Precision = 1.00**, **Recall = 1.00**, and **F1-score = 1.00**

2. The confusion matrix showed that:

- True Negatives**(TN) = 113**, True Positives**(TP) = 2682**
- False Negatives**(FN) = 9,** False Positives**(FP) = 4**

Output:

```
KNN Model Performance:
Accuracy: 0.9953703703703703

Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.97      0.95       117
           1       1.00      1.00      1.00      2691

    accuracy                           1.00      2808
   macro avg       0.96      0.98      0.97      2808
weighted avg       1.00      1.00      1.00      2808


Confusion Matrix:
[[ 113    4]
 [   9 2682]]
```

SVM: SVM performed with an overall **accuracy** of **99.64%**

1. The classification report showed that:

- Class 0 had: **Precision = 0.94**, **Recall = 0.97**, and **F1-score = 0.96**
- Class 1 had: **Precision = 1.00**, **Recall = 1.00**, and **F1-score = 1.00**

1. The confusion matrix showed that:

- True Negatives**(TN) = 2684**, True Positives **(TP) = 114**
- False Negatives**(FN) = 7**, False Positives **(FP) = 3**

Output:

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
SVM Model Performance:
Best Parameters: {'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}
Accuracy: 0.9964387464387464

Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.97      0.96       117
           1       1.00      1.00      1.00      2691

    accuracy                           1.00      2808
   macro avg       0.97      0.99      0.98      2808
weighted avg       1.00      1.00      1.00      2808


Confusion Matrix:
[[ 114    3]
 [   7 2684]]
```

Logistic Regression: Logistic Regression performed with an overall **accuracy of 98.04%**

1. The classification report showed that:

- Class 0 had: **Precision = 0.7**, **Recall = 0.93**, and **F1-score = 0.80**
- Class 1 had: **Precision = 1.00**, **Recall = 0.98**, and **F1-score = 0.99**

2. The confusion matrix showed that:

- True Negatives**(TN) = 109**, True Positives **(TP) = 2644**
- False Negatives**(FN) = 47**, False Positives **(FP) = 8**

Output:

```
Logistic Regression Model Performance:
Accuracy: 0.9804131054131054

Classification Report:
              precision    recall  f1-score   support

           0       0.70      0.93      0.80       117
           1       1.00      0.98      0.99      2691

    accuracy                           0.98      2808
   macro avg       0.85      0.96      0.89      2808
weighted avg       0.98      0.98      0.98      2808


Confusion Matrix:
[[ 109    8]
 [  47 2644]]
```

Decision Tree: Logistic Regression performed with an overall **accuracy of 99.75%**

1. The classification report showed that:

- Class 0 had: **Precision = 0.98**, **Recall = 0.96**, and **F1-score = 0.97**
- Class 1 had: **Precision = 1.00**, **Recall = 1.00**, and **F1-score = 1.00**

2. The confusion matrix showed that:

- True Negatives**(TN) = 112**, True Positives **(TP) = 2689**
- False Negatives**(FN) = 2**, False Positives **(FP) = 5**

Output:

```
Decision Tree Model Performance:
Accuracy: 0.9975071225071225

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.96      0.97       117
           1       1.00      1.00      1.00      2691

    accuracy                           1.00      2808
   macro avg       0.99      0.98      0.98      2808
weighted avg       1.00      1.00      1.00      2808


Confusion Matrix:
[[ 112    5]
 [   2 2689]]
```

XGBoost: Logistic Regression performed with an overall **accuracy of 99.82%**

1. The classification report showed that:

- Class 0 had: **Precision = 0.99**, **Recall = 0.97**, and **F1-score = 0.98**
- Class 1 had: **Precision = 1.00**, **Recall = 1.00**, and **F1-score = 1.00**

2. The confusion matrix showed that:

- True Negatives**(TN) = 2690**, True Positives **(TP) = 113**
- False Negatives**(FN) = 1**, False Positives **(FP) = 4**

Output:

```
XGBoost Model Performance:
Accuracy: 0.9982193732193733

Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.97      0.98       117
           1       1.00      1.00      1.00      2691

    accuracy                           1.00      2808
   macro avg       0.99      0.98      0.99      2808
weighted avg       1.00      1.00      1.00      2808


Confusion Matrix:
[[ 113    4]
 [   1 2690]]
```

Model Evaluation (Label Encoding)

Random Forest: Random forest **best** performed with an overall **accuracy of 99.75%**

3. The **classification report** showed that:

- **Class 0** had: **Precision=0.99**. **Recall= 0.95,** and **F1-score= 0.97**
- **Class 1** had: **Precision=1, Recall= 1**, and **F1-score= 1**

4. The **Confusion Matrix** showed that:

- True Negatives**(TN) = 111,** True Positives**(TP) = 6**
- False Negatives**(FN) = 1,** False Positives**(FP) = 2690**

Output:

```
Random Forest Model Performance:
Accuracy: 0.9975071225071225

Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.95      0.97       117
           1       1.00      1.00      1.00      2691

    accuracy                           1.00      2808
   macro avg       0.99      0.97      0.98      2808
weighted avg       1.00      1.00      1.00      2808


Confusion Matrix:
[[ 111    6]
 [    1 2690]]
```

KNN: The K-Nearest Neighbors (KNN) model achieved an overall **accuracy of 99.43%**

3. The classification report showed that:

- **Class 0** had: **Precision = 0.92**, **Recall = 0.95**, and **F1-score = 0.93**
- **Class 1** had: **Precision = 1.00**, **Recall = 1.00**, and **F1-score = 1.00**

4. The confusion matrix showed that:

- True Negatives**(TN) = 111**, True Positives**(TP) = 6**
- False Negatives**(FN) = 10,** False Positives**(FP) = 2681**

Output:

```
KNN Model Performance:
Accuracy: 0.9943019943019943

Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.95      0.93       117
           1       1.00      1.00      1.00      2691

    accuracy                           0.99      2808
   macro avg       0.96      0.97      0.96      2808
weighted avg       0.99      0.99      0.99      2808


Confusion Matrix:
[[ 111    6]
 [  10 2681]]
```

SVM: SVM performed with an overall **accuracy** of **99.57%**

2. The classification report showed that:

- Class 0 had: **Precision = 0.95**, **Recall = 0.95**, and **F1-score = 0.95**
- Class 1 had: **Precision = 1.00**, **Recall = 1.00**, and **F1-score = 1.00**

2. The confusion matrix showed that:

- True Negatives**(TN) = 111**, True Positives **(TP) = 6**
- False Negatives**(FN) = 6**, False Positives **(FP) = 2685**

Output:

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
SVM Model Performance:
Best Parameters based on Accuracy: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
Accuracy: 0.9957264957264957

Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.95      0.95       117
           1       1.00      1.00      1.00      2691

    accuracy                           1.00      2808
   macro avg       0.97      0.97      0.97      2808
weighted avg       1.00      1.00      1.00      2808


Confusion Matrix:
[[ 111    6]
 [   6 2685]]
```

Logistic Regression: Logistic Regression performed with an overall **accuracy of 96.79%**

3. The classification report showed that:

- Class 0 had: **Precision = 0.57**, **Recall = 0.92**, and **F1-score = 0.71**
- Class 1 had: **Precision = 1.00**, **Recall = 0.97**, and **F1-score = 0.98**

4. The confusion matrix showed that:

- True Negatives**(TN) = 108**, True Positives **(TP) = 9**
- False Negatives**(FN) = 81**, False Positives **(FP) = 2610**

Output:

```
Logistic Regression Model Performance:
Accuracy: 0.967948717948718

Classification Report:
              precision    recall  f1-score   support

           0       0.57      0.92      0.71       117
           1       1.00      0.97      0.98      2691

    accuracy                           0.97      2808
   macro avg       0.78      0.95      0.84      2808
weighted avg       0.98      0.97      0.97      2808


Confusion Matrix:
[[ 108    9]
 [  81 2610]]
```

Decision Tree: Logistic Regression performed with an overall **accuracy of 99.60%**

3. The classification report showed that:

- Class 0 had: **Precision = 0.96**, **Recall = 0.95**, and **F1-score = 0.95**
- Class 1 had: **Precision = 1.00**, **Recall = 1.00**, and **F1-score = 1.00**

4. The confusion matrix showed that:

- True Negatives**(TN) = 111**, True Positives **(TP) = 6**
- False Negatives**(FN) = 5**, False Positives **(FP) = 2686**

Output:

```
Decision Tree Model Performance:
Accuracy: 0.9960826210826211

Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.95      0.95       117
           1       1.00      1.00      1.00      2691

    accuracy                           1.00      2808
   macro avg       0.98      0.97      0.98      2808
weighted avg       1.00      1.00      1.00      2808


Confusion Matrix:
[[ 111    6]
 [   5 2686]]
```

XGBoost: Logistic Regression performed with an overall **accuracy of 99.67%**

3. The classification report showed that:

- Class 0 had: **Precision = 0.97**, **Recall = 0.95**, and **F1-score = 0.96**
- Class 1 had: **Precision = 1.00**, **Recall = 1.00**, and **F1-score = 1.00**

4. The confusion matrix showed that:

- True Negatives**(TN) = 111**, True Positives **(TP) = 6**
- False Negatives**(FN) = 3**, False Positives **(FP) = 2688**

Output:

```
XGBoost Model Performance:
Accuracy: 0.9967948717948718

Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       117
           1       1.00      1.00      1.00      2691

    accuracy                           1.00      2808
   macro avg       0.99      0.97      0.98      2808
weighted avg       1.00      1.00      1.00      2808


Confusion Matrix:
[[ 111    6]
 [   3 2688]]
```

Conclusion:

The **XGBoost** model has the **best accuracy, precision, recall, and F-1 scores** out of the 6 models. However, the winner is **Random Forest** since its confusion matrix showed **false negative(FN) =0** which is the **least** out of all the models alongside the **second best accuracy, precision, recall, and F-1 score**. Given the nature of our field, **false negatives (undetected intrusions)** are intolerable thus we must pick the least false negatives out of the models.

Ensembling Technique:

Stacking:

- Stacking using **Random forest**, **SVM**, and **Decision tree**, with **Random Forest** as the **meta-model** showed performance measures the same as using only Random forest.
- This shows that these 6 false positives are either **borderline** or **noisy** or **mislabeled** as class 1 instead of class 0.
- Choosing to pursue 100% accuracy will either reduce real-world **generalizability** or cause **overfitting** which isn't worth it as all false negatives are 0 and all true positives are correctly classified given the nature of the dataset.

```
Stacking Model Performance:
Accuracy: 0.9978632478632479

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.95      0.97       117
           1       1.00      1.00      1.00      2691

    accuracy                           1.00      2808
   macro avg       1.00      0.97      0.99      2808
weighted avg       1.00      1.00      1.00      2808


Confusion Matrix:
[[ 111    6]
 [   0 2691]]
```

Conclusion

- To summarize, we used **IQR** to detect outliers and **Winsorization** to handle them.
- Handled data **inconsistencies** by making sure columns conform to a specific range.
- Applied **log transformation** to deal with skewed data by **compressing** larger values and **expanding** smaller ones.
- **Split** the data first into **80% training** and **20% testing** data.
- Applied **Mutual Information** to find whether current categorical columns affect the target column (class) or not.
- Applied encoding techniques like **Label Encoding** and **One-Hot Encoding.**
- **Normalized** the dataset using **StandardScaler** which is important for non-tree based models.

- Used a set of **Feature Selection** techniques like:

1. Variance Thresholding
2. Mutual Information (MI)
3. Lasso Regression
4. Redundant Feature Selection (RFE)
5. Correlation Matrix Selection

- Handled target column data imbalance using **imblearn's SMOTE** technique

After finishing all of pre and post processing along with encoding, it was time to train the initial models.

1. Trained **KNN** to determine the **linear separability** of the data.
2. Trained **Logistic Regression** to determine the **complexity** of the data
3. Trained **SVM** to determine whether this data has a **clear margin of separation** or not while implementing **Grid Search** to optimize the hyperparameters.
4. Trained **Decision Tree** as it's the **baseline** for the upcoming training models
5. Trained **Random Forest** to determine the **extent** the best models can reach
6. Trained **XGBoost** as a final model to reach a **concrete judgment** on the data and the best model for its nature.

- Ran **RFE** after training the initial **Random Forest** model to remove useless columns not affecting the outcome.

- Trained the final model which is an ensembling (stacking) model; Using base models of **Random Forest, Decision Tree, SVM** and Random Forest as my **meta-model** which showed the same performance measures as Random Forest.
- To verify our conclusion, we trained a final stacking model with **Random Forest, Decision Tree, SVM** as the base models with **XGBoost** as the **meta-model** which also showed the same performance. This proved that our hypothesis was correct.

Conclusion:

The best performing model reached a confusion matrix of [[111 6]
[0 2691]]

Which highlights how there are **zero** false negatives, where this is sought after in this field of network intrusion. The other 6 false positives are either **borderline** or **noisy** or **mislabeled** as class 1 instead of class 0
Choosing to pursue **100%** accuracy will either reduce real-world **generalizability** or cause **overfitting** which isn't worth it as all false negatives are 0 and all true positives are correctly classified given the nature of the dataset.

—---