MICROSOFT-CYBER-SECURITY-INCIDENT-GRADE-CLASSIFICATION

Vinoothna Nadikatla

PROJECT OVERVIEW

The Classifying Cyber security Incidents project is a machine learning-based solution designed to enhance the efficiency of Security Operation Centers (SOCs) at Microsoft. The project focuses on building a classification model that accurately predicts the triage grade of cyber security incidents, assisting SOC analysts in prioritizing their efforts and responding to threats efficiently.

The goal is to classify incidents as:

*True Positive (TP): A confirmed threat.

*Benign Positive (BP): A false alarm.

*False Positive (FP): A misidentified threat.

DATASET

Shape of the Dataset

The dataset initially contained 9,516,837 rows and 45 columns.

Feature Description The dataset contains a variety of features describing cyber security incidents, their attributes, and metadata. Below are the features:

Id: Unique ID for each OrgId-IncidentId pair.

IncidentId: Organizationally unique incident identifier.

AlertId: Unique identifier for an alert.

Timestamp: Time the alert was created.

DetectorId: Unique ID for the alert-generating detector.

AlertTitle: Title of the alert.

- ActionGrouped: SOC alert remediation action (high level).
- ActionGranular: SOC alert remediation action (fine-grain).
- EntityType: Type of entity involved in the alert
- EvidenceRole: Role of the evidence in the investigation.
- RolesAdditional: Metadata on evidence role in the alert.
- DeviceId: Unique identifier for the device.
- DeviceName: Name of the device. Sha256: SHA-256 hash of the file.
- IpAddress: IP address involved.
- Url: URL involved.
- AccountSid: On-premises account identifier.
- AccountUpn: Email account identifier.
- AccountObjectId: Entra ID account identifier.
- AccountName: Name of the on-premises account.

- NetworkMessageId: Org-level identifier for the email message.
- EmailClusterId: Unique identifier for the email cluster.
- RegistryKey: Registry key involved.
- RegistryValueName: Name of the registry value.
- RegistryValueData: Data of the registry value.
- ApplicationId: Unique identifier for the application.
- ApplicationName: Name of the application.
- OAuthApplicationId: OAuth application identifier.
- ThreatFamily: Malware family associated with a file.
- FileName: Name of the file.
- FolderPath: Path of the file folder.

- ResourceIdName: Name of the Azure resource.
- ResourceType: Type of Azure resource.
- OSFamily: Family of the operating system.
- OSVersion: Version of the operating system.
- AntispamDirection: Direction of the antispam filter.
- SuspicionLevel: Level of suspicion.
- LastVerdict: Final verdict of threat analysis.
- CountryCode: Country code where evidence appears.
- State: State where evidence appears.
- City: City where evidence appears.

Data Exploration:

- **Data Loading:** The dataset was loaded in chunks to handle its large size.

- **Summary Statistics:** The data was analyzed to check its structure, data types, and missing values.

- **Visualizations:** The distribution of key features, including the target variable (Incident Grade), was visualized to understand class imbalances.

- **Class Imbalance:** The Benign Positive class is significantly overrepresented compared to the other classes.

Dropping Pure Identifier Columns:
Removed and handled missing and duplicate values

10 columns were identified with more missing values:
- •MitreTechniques,
- •ActionGrouped,
- •ActionGranular,
- •EmailClusterId,
- •ThreatFamily,
- •ResourceType,
- •Roles,
- •AntispamDirection,
- •SuspicionLevel,
- •LastVerdict.

Columns Dropped:

These 10 columns were removed from the dataset to eliminate high-missing attributes that could introduce noise or reduce model performance.

Missing value imputation for categorical columns:

•train_df.select_dtypes(): Selects columns based on data type.

•include=['object', 'category']: Filters columns that are either string (object) or categorical (category).

•train_df[col].isnull().any(): Checks if the column contains any missing values.

•train_df[col].mode()[0]: Computes the most frequently occurring value (mode) of the column.

•train_df[col].fillna(mode_val): Replaces missing values in the column with the mode.

Why Mode Imputation?

Categorical Data: The mode is a sensible choice for filling missing values in categorical columns since it preserves the majority class and minimizes bias.
Preserves Distribution: Mode imputation ensures the distribution of the data remains largely intact.

Missing values for numerical columns using the median value :

•train_df.select_dtypes(): Selects columns based on their data type.
•include=['int64', 'float64']: Filters columns with integer or floating-point data types.
•train_df[col].median(): Calculates the median of the column.
•train_df[col].fillna(median_val): Replaces all missing values in the column with its median.

Why Median Imputation?

•**Robustness:** The median is less affected by outliers compared to the mean, making it a more stable choice for imputation, especially when dealing with skewed data.

•**Preserves Data Integrity**: Imputing with the median keeps the overall distribution of the column closer to its original state.

EXPLORATORY DATA ANALYSIS

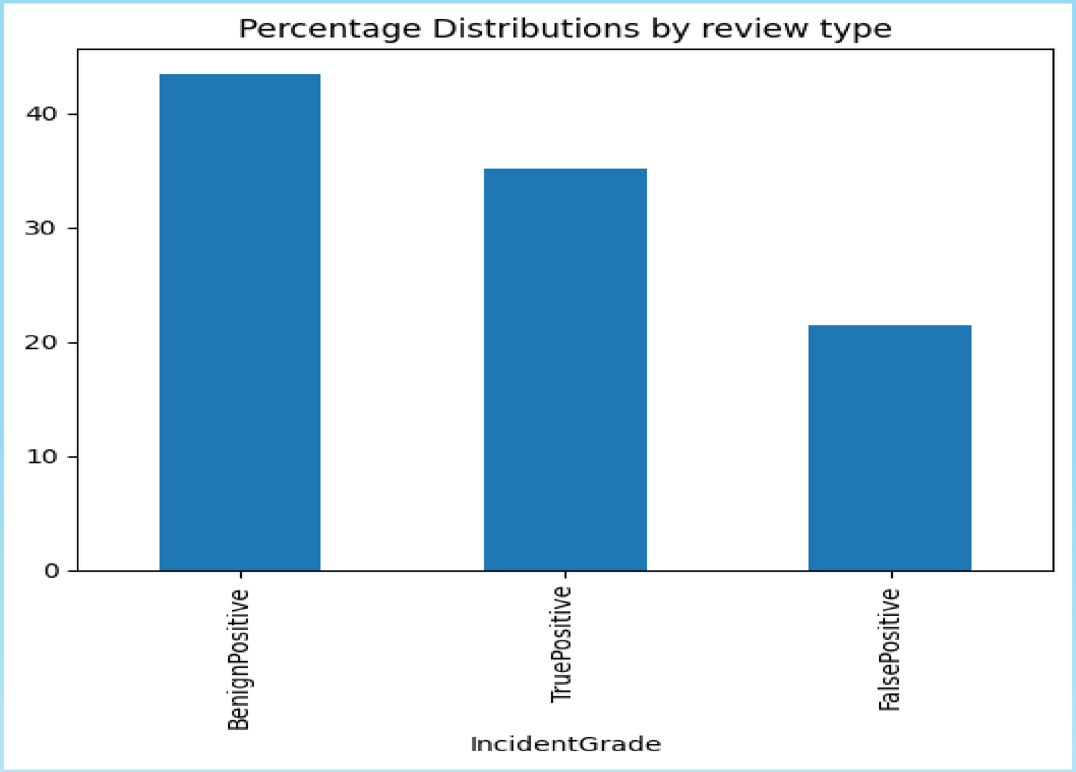
Value Counts for Key Features:

Incident Grade

❖Benign Positive – 4110817 (The highest count)

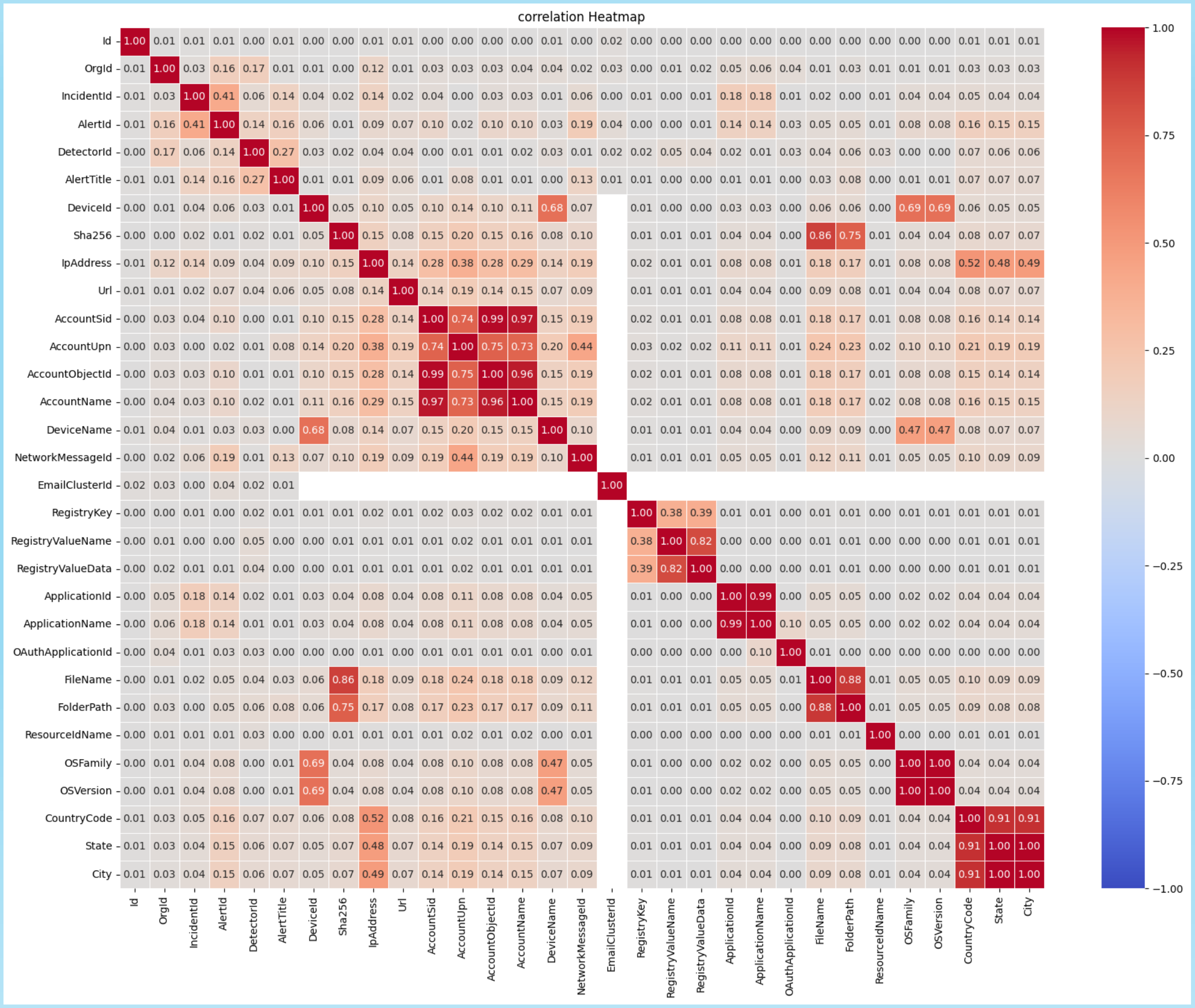
❖True Positive - 3322713

❖False Positive - 2031967



EXPLORATORY DATA ANALYSIS

Numerical columns HeatMap:



FEATURE ENGINEERING:

Timestamp Analysis:

The Timestamp column was converted into several derived time-related features:

•Converts the **Timestamp** column to a Pandas **datetime** object.
•Extracts the year from the **Timestamp**
•Extracts the month (1–12) from the **Timestamp**
•Extracts the day of the week (0 = Monday, 6 = Sunday)
•Extracts the hour of the day (0–23)
•Removes the original **Timestamp** column as its information is now represented in the derived features.

Encoding Categorical Variables

label encoding to convert categorical features into numerical values

- categorical_columns = ['Category', 'IncidentGrade', 'EntityType', 'EvidenceRole']
- Creates an instance of the LabelEncoder class, which will be used to convert categorical labels into numeric labels.
- Loops through each column in the categorical_columns list.
- if col in train_df.columns: Checks if the column exists in the DataFrame to avoid errors.
- .astype(str): Ensures the column is of string type before encoding (some categorical variables might have numeric representations, and label encoding requires them to be in string form).
- label_encoder.fit_transform(): Encodes the categorical values as integers. It assigns a unique integer to each unique category in the column.

Why Use Label Encoding?
•Efficient Transformation:

Converts categorical values (such as strings) into integers (e.g., "Low", "Medium", "High" becomes 0, 1, 2).

•Model Compatibility:

Many machine learning models (e.g., decision trees, random forests) can directly work with numeric values, and label encoding provides a simple transformation.

•Ordered Data:
For ordinal data (e.g., IncidentGrade), label encoding maintains the inherent order between categories

Data Preprocessing:

• Handling Missing Data: Missing values were imputed using forward fill and mean imputation. Columns with more than 50% missing values were dropped.

• Feature Engineering: Derived timestamp-based features and removed redundant columns.

• Encoding Categorical Variables: Categorical features were converted into numerical formats using encoding technique.
• Scaling: Standardized numerical features to ensure equal contribution during model training.

Final Cleanup
•Removed duplicated columns to prevent redundancy:

```
train_df.drop_duplicates(inplace=True)
```

•Dataset saves the cleaned and preprocessed DataFrame as a CSV file

Export and Repeat for Test Data

The processed dataset was saved as test_df.csv, and the same preprocessing steps were applied to the test data.

Data Exploration:

• Data Loading: The dataset was loaded in chunks to handle its large size.

• Summary Statistics: The data was analysed to check its structure, data types, and missing values.

• Visualizations: The distribution of key features, including the target variable (Incident Grade), was visualized to understand class imbalances.

• Class Imbalance: The Benign Positive class is significantly overrepresented compared to the other classes.

Data Preprocessing:

• Handling Missing Data: Missing values were imputed using forward fill and mean imputation. Columns with more than 50% missing values were dropped.

• Feature Engineering: Derived timestamp-based features and removed redundant columns.

Encoding Categorical Variables: Categorical features were converted into numerical formats using encoding technique.

• Scaling: Standardized numerical features to ensure equal contribution during model training.

Data Splitting:

Split the data into training and validation sets to evaluate model performance.

Train-Validation Split: Data was split into 80% for training and 20% for validation, while maintaining the balance between classes.

Model Selection and Training:

- **Logistic Regression:** A simple model used as a baseline for comparison.
- **Decision Tree:** A non-linear model that works well for small datasets and easy interpretability.
- **Random Forest:** An ensemble of decision trees that provides more accuracy and stability.
- **Gradient boosting:** improves model accuracy through sequential weak learners .
- **XGBoost:** A powerful algorithm that handles large datasets efficiently.
- **LightGBM:** Free and open-source distributed gradient-boosting framework for machine learning, originally developed by Microsoft.

Model	Accuracy	Macro-F1 Score	Precision	Recall
Logistic Regression	0.59	0.57	0.59	0.59
Random Forest	0.70	0.65	0.78	0.70
Gradient Boosting	0.72	0.69	0.79	0.72
XGBoost	0.80	0.74	0.81	0.77
LightGBM	0.75	0.61	0.78	0.70

XGBoost emerges as the best model based on Macro-F1 Score (0.74) and other metrics.

Logistic Regression has the lowest scores, as expected for a simpler linear model, and may not be suited for this complex dataset.

Model Evaluation and Tuning:
Evaluate the model's performance using cross-validation and optimize it using hyperparameter tuning.

Metrics Used

• **Accuracy:** Measures overall correctness.

• **Precision:** Measures how many positive predictions were correct.

• **Recall:** Measures how well the model identifies actual positives.

• **Macro-F1 Score:** A balanced metric that treats all classes equally.
Hyperparameter Tuning: RandomizedSearchCV was used to find the best settings for Random Forest and XGBoost

Key Outcomes
• **XGBoost performed best, achieving an accuracy of 80% and a macro-F1 score of 74%.**