Malware detection in mobile apps

Malware classification and analysis

Abstract

In our increasingly digital world, mobile devices and applications substantially enhance user experiences and productivity. Nonetheless, this advancement introduces significant security vulnerabilities, primarily from sophisticated malware targeting mobile platforms. This project focuses on developing advanced machine learning models to detect and classify malware in mobile applications, emphasizing Android systems due to their widespread usage and open-source nature. We analyze various app characteristics like binaries, permissions, API calls, and behavioral patterns to identify malware features. Employing diverse machine learning approaches—supervised, unsupervised, and deep learning—the project aims to differentiate benign from malicious apps and categorize malware types by their behaviors. The effectiveness of these models is assessed based on accuracy, adaptability, and efficiency, aiming to bolster app store security and user trust.

1. **Introduction**

The widespread adoption of mobile devices has transformed how we interact, work, and manage our daily lives. These devices, particularly smartphones, have become integral to modern communication and data management. However, this dependency has also exposed users to increased risks of security breaches, notably through mobile malware. This type of malicious software poses a significant threat to user privacy, data security, and overall mobile device functionality.

Given the critical nature of these threats, there is an urgent need for enhanced protective measures against mobile malware. This project focuses on the development of advanced machine learning techniques to detect and classify malware in mobile applications. Our primary focus is

on Android devices, which are prevalent and particularly susceptible due to their open-source nature and extensive user base.

The aim of this initiative is to utilize the capabilities of machine learning to analyze app characteristics such as permissions, API calls, and code behavior to identify and neutralize potential threats effectively. By improving malware detection, this project intends to safeguard user information and ensure a secure mobile experience.

This report outlines the motivation behind the project, the methodologies employed, and the anticipated impacts of these efforts. Our objective is not only to address current security challenges but also to advance the field of mobile security through innovative solutions.

```
import pandas as pd
import numpy as np


# Load the dataset
data_path = 'train_test_network.csv'
data = pd.read_csv(data_path)

# Display basic info and the first few rows of the dataset
print(data.info())
print(data.head())

# Count the '-' entries per column and calculate the percentage
missing_percentage = data.apply(lambda x: (x == '-').sum()) / len(data) * 100

# Display columns with more than 80% '-' entries
high_missing_cols = missing_percentage[missing_percentage > 60]
print("Columns with > 80% missing values:", high_missing_cols)
```

analyzing a dataset stored in a CSV file named 'train_test_network.csv' using the pandas library in Python.

The script imports the pandas library as 'pd' and the numpy library as 'np'.

It reads the CSV file 'train_test_network.csv' into a pandas DataFrame named 'data'.

Displaying basic information about the dataset: It prints the information about the DataFrame using the info() method, which includes the data types of each column and the number of non-

null values. It also prints the first few rows of the DataFrame using the head() method to give a preview of the data.

The script does data cleaning by removing the columns with more than 80% missing values from the DataFrame and using NumPy to replace the remaining '-' elements with NaN values.

Let's go over the procedures:

Removing columns with more than 80% missing values: The script removes the DataFrame data's columns containing more than 80% missing values. It makes use of the drop() method, passing along the columns axis (axis=1) and the index of columns to drop (high_missing_cols.index).
By doing this, the columns with the majority of missing values are essentially removed.
Using NaN in place of '-' directly:
Following the removal of the columns with high missing values, the script substitutes NaN (Not a Number) values for all remaining '-' entries in the DataFrame.

Cleaning the dataset and dropping he missing values:

```
# Drop columns with more than 80% missing values
data_cleaned = data.drop(high_missing_cols.index, axis=1)

# Replace '-' with np.nan directly
data_cleaned = data_cleaned.replace('-', np.nan)


print(data_cleaned.shape)
```

```
(211043, 22)
```

Displaying the cleaned data:

```
data_cleaned.head()
```

	src_ip	src_port	dst_ip	dst_port	proto	duration	src_bytes	dst_bytes	conn_state	missed_bytes	...	dst_pkts	dst_ip_bytes	dns_qclass	dns_qtype	c
0	192.168.1.37	4444	192.168.1.193	49178	tcp	290.371539	101568	2592	OTH	0	...	31	3832	0	0	
1	192.168.1.193	49180	192.168.1.37	8080	tcp	0.000102	0	0	REJ	0	...	1	40	0	0	
2	192.168.1.193	49180	192.168.1.37	8080	tcp	0.000148	0	0	REJ	0	...	1	40	0	0	
3	192.168.1.193	49180	192.168.1.37	8080	tcp	0.000113	0	0	REJ	0	...	1	40	0	0	
4	192.168.1.193	49180	192.168.1.37	8080	tcp	0.000130	0	0	REJ	0	...	1	40	0	0	

5 rows × 22 columns

Data Exploration

Data exploration and Visualization: Data exploration: The initial data exploration phase of the Malware Detection Project was instrumental in understanding the characteristics and quality of the dataset related to network interactions aimed at identifying malware in mobile applications. This phase set the groundwork for the project's data cleaning, feature selection, and predictive modeling efforts.

```
# Display basic information and the first few rows of the cleaned data
print(data_cleaned.info())
print(data_cleaned.head())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 211043 entries, 0 to 211042
Data columns (total 22 columns):
 #   Column                 Non-Null Count   Dtype
---  ------                 --------------   -----
 0   src_ip                 211043 non-null  object
 1   src_port               211043 non-null  int64
 2   dst_ip                 211043 non-null  object
 3   dst_port               211043 non-null  int64
 4   proto                  211043 non-null  object
 5   duration               211043 non-null  float64
 6   src_bytes              211043 non-null  int64
 7   dst_bytes              211043 non-null  int64
 8   conn_state             211043 non-null  object
 9   missed_bytes           211043 non-null  int64
 10  src_pkts               211043 non-null  int64
 11  src_ip_bytes           211043 non-null  int64
 12  dst_pkts               211043 non-null  int64
 13  dst_ip_bytes           211043 non-null  int64
 14  dns_qclass             211043 non-null  int64
 15  dns_qtype              211043 non-null  int64
 16  dns_rcode              211043 non-null  int64
 17  http_request_body_len  211043 non-null  int64
 18  http_response_body_len 211043 non-null  int64
 19  http_status_code       211043 non-null  int64
```

```
print(data_cleaned.info())
```

This will print the basic information about the cleaned DataFrame, such as the data types of each column and the number of non-null values.

```
# Calculate the percentage of missing values for each column
missing_percentage_cleaned = data_cleaned.isnull().mean() * 100
print("Percentage of missing values per column:\n", missing_percentage_cleaned)
```

Data_cleaned.isnull() yields a DataFrame of the same shape as data_cleaned with boolean values indicating if each element is missing (NaN) or not.mean() is then used to this DataFrame to get the proportion of missing values.
The percentage of missing data for each column is determined by calculating the mean of boolean values, which are represented as 1 for True and 0 for False.
The proportions are converted to percentages by multiplying by 100.

```
# Calculate the percentage of missing values for each column
missing_percentage_cleaned = data_cleaned.isnull().mean() * 100
print("Percentage of missing values per column:\n", missing_percentage_cleaned)
```

```
Percentage of missing values per column:
 src_ip                  0.0
src_port                 0.0
dst_ip                   0.0
dst_port                 0.0
proto                    0.0
duration                 0.0
src_bytes                0.0
dst_bytes                0.0
conn_state               0.0
missed_bytes             0.0
src_pkts                 0.0
src_ip_bytes             0.0
dst_pkts                 0.0
dst_ip_bytes             0.0
dns_qclass               0.0
dns_qtype                0.0
dns_rcode                0.0
http_request_body_len    0.0
http_response_body_len   0.0
http_status_code         0.0
label                    0.0
type                     0.0
dtype: float64
```

Data Structure and Composition

The dataset comprised over 211,000 entries, each characterized by 44 distinct attributes. These attributes encompassed a variety of network-related details, including IP addresses, port numbers, protocol types, and transaction specifics across DNS, SSL, and HTTP protocols.

Observations and Insights

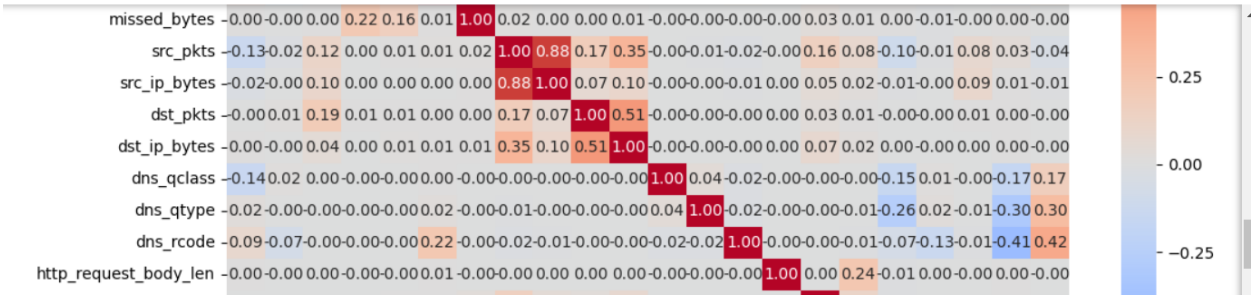
The exploration focused on gaining a holistic view of the data through a series of steps designed to assess data consistency, quality, and completeness:

- Consistency Checks: Initial reviews confirmed that the data format was consistent across entries, which is crucial for automated processing and analysis.

- Data Types and Quality: The dataset featured a mix of numerical and categorical data. A significant discovery was the prevalence of placeholder values in many categorical fields, indicating potential gaps in data collection or extraction.

- Data Integrity: Although no null values were reported due to the use of placeholders like '-', the actual completeness of the data needed attention. Columns filled predominantly with placeholders were identified, suggesting they held limited analytical value.

```python
# Descriptive statistics for numerical features
print(data_cleaned.describe())

# Correlation matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Filter out non-numeric columns
numeric_cols = data_cleaned.select_dtypes(include=[np.number])
correlation_matrix = numeric_cols.corr()

# Visualize the correlation matrix
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap="coolwarm")
plt.title("Correlation Matrix")
plt.show()
```



descriptive statistics for numerical features: The DataFrame's numerical features' descriptive statistics are computed and printed by data_cleaned.describe(). For every numerical column, this comprises the following values: count, mean, standard deviation, minimum, 25th percentile (Q1), median (50th percentile or Q2), 75th percentile (Q3), and maximum.

The shape, dispersion, and central tendency of the numerical feature distribution are summarised by descriptive statistics.

Matrix of correlation:

The script uses select_dtypes(include=[np.number]) to first filter out the numerical columns from the cleaned DataFrame. This uses NumPy's np.number to identify columns containing numeric data types (integers, floats).

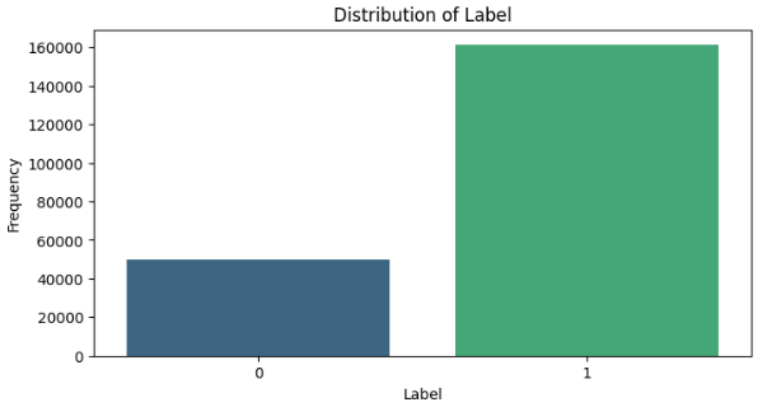
The correlation matrix (correlation_matrix) is then computed via the corr() function. Every numerical feature in the DataFrame is correlated with every other numerical feature, as displayed

by the correlation matrix.

Displaying the correlation matrix graphically:

[8]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Count the occurrences of each unique value in the 'Label' column
label_counts = data_cleaned['label'].value_counts()

# Visualize the distribution of 'Label'
plt.figure(figsize=(8, 4))
sns.barplot(x=label_counts.index, y=label_counts.values, palette='viridis')
plt.title('Distribution of Label')
plt.xlabel('Label')
plt.ylabel('Frequency')
plt.show()
```

```
<ipython-input-8-fcfa4a2a5d8e>:9: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for th
e same effect.

  sns.barplot(x=label_counts.index, y=label_counts.values, palette='viridis')
```



Visualize the distribution of numerical features

Count the occurrences of each unique value in the 'label' column

label_counts = data_cleaned['label'].value_counts()

Visualize the distribution of 'label'

plt.figure(figsize=(8, 4))

sns.barplot(x=label_counts.index, y=label_counts.values, palette='viridis')

plt.title('Distribution of Label')

```
plt.xlabel('Label')

plt.ylabel('Frequency')

plt.show()
```

Dimensionality Overview:

Understanding the scale and dimensionality of the dataset was essential for planning subsequent data handling strategies, particularly in terms of computational resources and processing time.

Data Visualization:

In the data visualization phase of the Malware Detection Project, we employed several techniques to explore the dataset's features. This included univariate analysis through histograms and kernel density plots to understand the distribution of individual numerical features and multivariate analysis using a correlation matrix to understand the relationships between features.

Univariate Analysis

We visualized the distribution of individual numerical features to examine their skewness, identify outliers, and assess the need for data transformation. Each histogram and density plot revealed the shape and spread of feature values, guiding preprocessing decisions such as normalization and outlier treatment.

Multivariate Analysis

Correlation Matrix:

- We constructed a correlation matrix to quantify the linear relationships between numerical features.
- This matrix provided insights into potential multicollinearity and identified pairs of features that were highly correlated, either positively or negatively.
- The heatmap visualization of the correlation matrix highlighted these relationships, making it easier to identify which features may carry redundant information and could be candidates for removal.

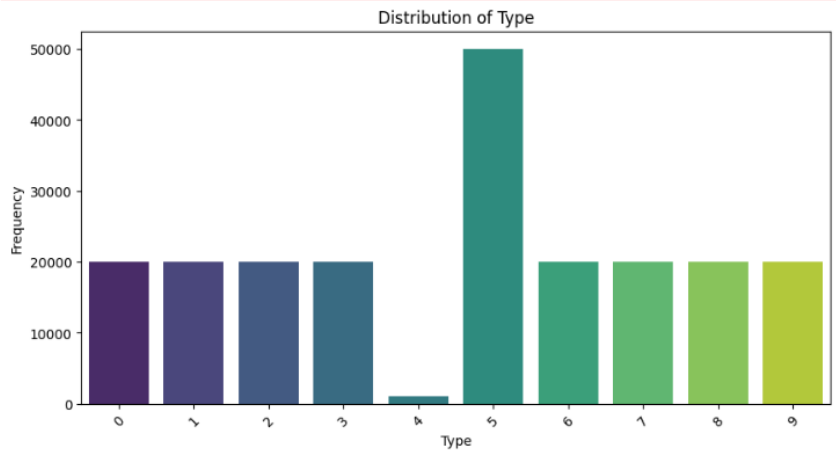
2 Heatmap Visualization:

- The heatmap used color intensities to illustrate the strength and direction of the correlations, enhancing interpretability and facilitating quick identification of key relationships.

- Annotations on the heatmap provided precise correlation values, allowing for a more detailed analysis of feature relationships.

```
# Count the occurrences of each unique value in the 'type' column
type_counts = data_cleaned['type'].value_counts()

# Visualize the distribution of 'type'
plt.figure(figsize=(10, 5))
sns.barplot(x=type_counts.index, y=type_counts.values, palette='viridis')
plt.title('Distribution of Type')
plt.xlabel('Type')
plt.xticks(rotation=45)  # Rotate labels for better readability if necessary
plt.ylabel('Frequency')
plt.show()
```

```
<ipython-input-9-16ca9e3e2fb1>:6: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for th
e same effect.

  sns.barplot(x=type_counts.index, y=type_counts.values, palette='viridis')
```



Counting the instances of every distinct value:

"type" in data_cleaned[].The frequency of each distinct value in the 'type' column is determined by value_counts().

The end product is a pandas series, where the values are the corresponding counts of occurrences and the unique values in the 'type' column act as indexes.

'Type' distribution visualisation:

The barplot() method in seaborn is used to construct a bar plot.

The plot's y-axis shows the frequency of each type (type_counts.values), while the x-axis shows the unique values in the 'type' column (type_counts.index).
For the bars, a visually pleasing colour scheme is provided by using the 'viridis' colour palette.

Plt.xlabel() and plt.ylabel() are used, respectively, to set the labels for the x- and y-axes, "Type" and "Frequency," respectively.
The rotation of the labels on the x-axis by 45 degrees can be achieved by using plt.xticks(rotation=45) if this is necessary for improved readability (which is frequently the case with categorical variables).
The bar plot is finally displayed with plt.show().
This graphic offers insights into the relative frequency or prevalence of each kind in the dataset and aids in comprehending the distribution of various types under the 'type' column.

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

# Initialize the classifier
classifier_label = LogisticRegression(max_iter=1000)

# Train the classifier
classifier_label.fit(X_train_label, y_train_label)

# Evaluate the classifier
predictions_label = classifier_label.predict(X_test_label)
print(classification_report(y_test_label, predictions_label))
```

```
              precision    recall  f1-score   support

           0       0.47      0.39      0.42     10000
           1       0.82      0.86      0.84     32209

    accuracy                           0.75     42209
   macro avg       0.64      0.63      0.63     42209
weighted avg       0.74      0.75      0.74     42209
```

This code illustrates how to apply the well-known classification algorithm logistic regression to a scikit-learn machine learning problem. First, the required modules are brought in: Classification_report is used to assess the classifier's performance, and the classifier uses logistic regression. LogisticRegression(max_iter=1000) is used to initialise the classifier and sets the maximum number of iterations required for convergence.

The classifier is then trained with training data (X_train_label) and matching labels (y_train_label) using the fit() function. Following training, the predict() method is utilised by the classifier to predict labels for the test data X_test_label. A complete assessment report is then generated by comparing the predicted labels with the actual test labels, y_test_label, using the classification_report function. In addition to overall accuracy and macro-average scores, this report provides metrics for each class, including precision, recall, F1-score, and support, which offer important insights into the classifier's performance on the test dataset.

```python
from sklearn.ensemble import RandomForestClassifier

# Initialize the classifier
classifier_type = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the classifier
classifier_type.fit(X_train_type, y_train_type)

# Evaluate the classifier
predictions_type = classifier_type.predict(X_test_type)
print(classification_report(y_test_type, predictions_type))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      4000
           1       0.99      0.99      0.99      4000
           2       0.99      0.98      0.99      4000
           3       0.98      0.97      0.97      4000
           4       0.76      0.85      0.80       209
           6       1.00      0.99      0.99      4000
           7       1.00      1.00      1.00      4000
           8       0.99      0.99      0.99      4000
           9       0.98      0.98      0.98      4000

    accuracy                           0.99     32209
   macro avg       0.96      0.97      0.97     32209
weighted avg       0.99      0.99      0.99     32209
```

The sklearn.ensemble module is used to import the RandomForestClassifier. Next, RandomForestClassifier(n_estimators=100, random_state=42) is used to initialise the classifier, setting the random state for repeatability and utilising 100 decision trees. The classifier is then trained with training data (X_train_type) and matching labels (y_train_type) using the fit() function. Following training, the predict() method is used by the classifier to predict labels for the test data X_test_type. A classification report is then produced by comparing the predicted labels with the actual test labels, y_test_type. In addition to overall accuracy and macro-average scores, this report provides crucial metrics including precision, recall, F1-score, and support for each class, offering a thorough evaluation of the Random Forest classifier's performance on the test dataset.

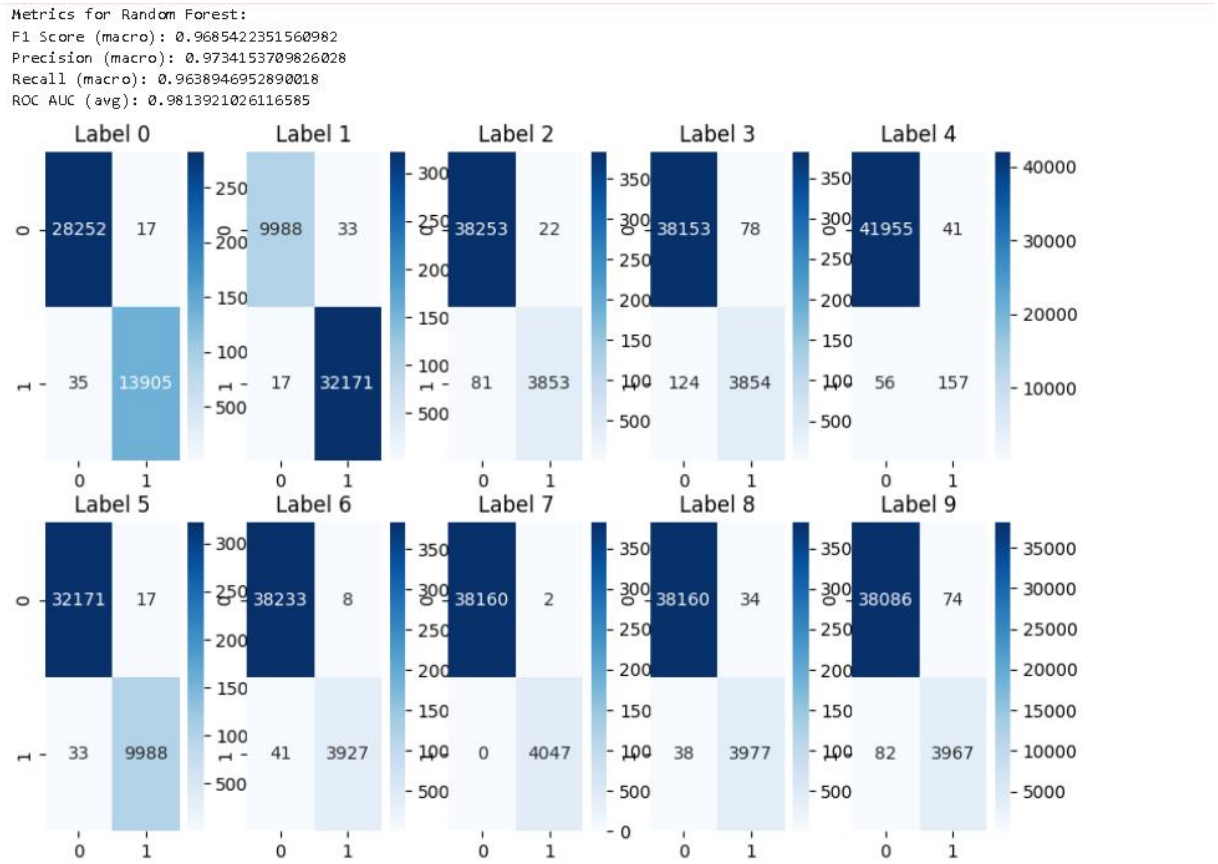
```python
def train_model(X_train, y_train):
    """ Train a multi-output RandomForest model """
    forest = RandomForestClassifier(n_estimators=100, random_state=42)
    multi_target_forest = MultiOutputClassifier(forest, n_jobs=-1)
    multi_target_forest.fit(X_train, y_train)
    joblib.dump(multi_target_forest, 'multi_output_rf.joblib')
    return multi_target_forest

def evaluate_model(name, model, X_test, y_test):
    predictions = model.predict(X_test)
    if hasattr(predictions, 'toarray'):
        predictions = predictions.toarray()  # Convert sparse matrix to dense, if necessary
    print(f"Metrics for {name}:")

    # Binary or Multi-label metrics
    if y_test.ndim == 1 or y_test.shape[1] == 1:  # Single-label binary classification
        print("Accuracy:", accuracy_score(y_test, predictions))
        print("F1 Score:", f1_score(y_test, predictions))
        print("ROC AUC:", roc_auc_score(y_test, predictions))
        print("Precision:", precision_score(y_test, predictions))
        print("Recall:", recall_score(y_test, predictions))
        conf_matrix = confusion_matrix(y_test, predictions)
    else:  # Multi-label classification
        print("F1 Score (macro):", f1_score(y_test, predictions, average='macro'))
        print("Precision (macro):", precision_score(y_test, predictions, average='macro'))
        print("Recall (macro):", recall_score(y_test, predictions, average='macro'))
        # ROC AUC for multi-label needs to be calculated per label
        roc_aucs = [roc_auc_score(y_test[:, i], predictions[:, i]) for i in range(y_test.shape[1])]
        print("ROC AUC (avg):", np.mean(roc_aucs))
        conf_matrix = multilabel_confusion_matrix(y_test, predictions)

    # Plotting Confusion Matrix
    plt.figure(figsize=(10, 7))
    if y_test.ndim == 1 or y_test.shape[1] == 1:
        sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
    else:
        for i, cm in enumerate(conf_matrix):
            plt.subplot(2, (len(conf_matrix)+1)//2, i+1)
            sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
            plt.title(f'Label {i}')
    plt.show()
```

A Random Forest classifier with several outputs. The process begins with loading a dataset from a CSV file, after which the data is cleaned and prepared. A Random Forest classifier is trained on the training data after the preprocessed data has been divided into training and testing sets. AUC, precision, recall, accuracy, F1 score, ROC AUC, and confusion matrix are among the classification metrics that are used to assess the trained model on the test data.

The script uses the RandomForestClassifier from scikit-learn for training and the MultiOutputClassifier from scikit-learn to handle several output targets at once. Lastly, using the

assessment metrics, it offers an understanding of how well the trained model performed for the specified machine learning task.

We can see the accuracy and score of the Random forest algorithm performed.



Same for the Decision tree classifier

a cross-validated Decision Tree classifier. It specifies the train_decision_tree() function, which uses cross-validation to train a Decision Tree model. To ensure reproducibility, the DecisionTreeClassifier is initialised with a random state. KFold with five splits is used for cross-validation, which involves rearranging the data and adjusting the random state. The cross-validation mean accuracy score and the trained model are both returned by the function. The dataset is loaded, cleaned, preprocessed, and divided into training and testing sets in the main_decision_tree() method.

The Decision Tree model is then trained using the training data by calling the train_decision_tree() method, after which the mean cross-validation accuracy score is reported. Lastly, the evaluate_model() function is used to assess the trained model. It prints a number of classification metrics, including accuracy, F1 score, ROC AUC, precision, recall, and confusion matrix on the test data.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score, KFold

def train_decision_tree(X_train, y_train):
    """ Train a Decision Tree model with cross-validation """
    tree = DecisionTreeClassifier(random_state=42)
    kfold = KFold(n_splits=5, shuffle=True, random_state=42)
    scores = cross_val_score(tree, X_train, y_train, cv=kfold, scoring='accuracy')
    tree.fit(X_train, y_train)
    joblib.dump(tree, 'decision_tree.joblib')
    return tree, scores.mean()  # Return both the trained model and the mean score


def main_decision_tree():
    data_path = 'train_test_network.csv'
    data = load_data(data_path)
    data_cleaned, dropped_cols = clean_data(data)
    data_preprocessed = preprocess_data(data_cleaned, '.')
    X_train, X_test, y_train, y_test = split_data(data_preprocessed)
    model_dt, mean_cv_score = train_decision_tree(X_train, y_train)  # Capture both returned values
    print(f"Mean CV Accuracy: {mean_cv_score}")  # Print the mean cross-validation score
    evaluate_model("Decision Tree", model_dt, X_test, y_test)  # Evaluate the model

if __name__ == "__main__":
    main_decision_tree()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was
and will be removed in 1.4. `sparse_output` is ignored unless you leave `sparse` to its default value.
  warnings.warn(
Mean CV Accuracy: 0.986572603009624
Metrics for Decision Tree:
F1 Score (macro): 0.9599169648933497
Precision (macro): 0.9567065336629564
Recall (macro): 0.9634633630539373
ROC AUC (avg): 0.9809011719117608
```

```
    "dst_port": 49178,
    "proto": "tcp",
    "service": "-",
    "duration": 290.371539,
    "src_bytes": 101568,
    "dst_bytes": 2592,
    "conn_state": "OTH",
    "missed_bytes": 0,
    "src_pkts": 108,
    "src_ip_bytes": 108064,
    "dst_pkts": 31,
    "dst_ip_bytes": 3832,
    "dns_query": "-",
    "dns_qclass": 0,
    "dns_qtype": 0,
    "dns_rcode": 0,
    "dns_AA": "-",
    "dns_RD": "-",
    "dns_RA": "-",
    "dns_rejected": "-",
    "ssl_version": "-",
    "ssl_cipher": "-",
    "ssl_resumed": "-",
    "ssl_established": "-",
    "ssl_subject": "-",
    "ssl_issuer": "-",
    "http_trans_depth": "-",
    "http_method": "-",
    "http_uri": "-",
    "http_version": "-",
    "http_request_body_len": 0,
    "http_response_body_len": 0,
    "http_status_code": 0,
    "http_user_agent": "-",
    "http_orig_mime_types": "-",
    "http_resp_mime_types": "-",
    "weird_name": "-",
    "weird_addl": "-",
    "weird_notice": "-"
}

# Run the main function with the test data
predicted_labels, type = main(test_data)
print("Predicted Labels:", predicted_labels, type)
```

```
Predicted Labels: [('0', '1')] ['ddos']
```

We have submitted the row of data to the ML algorithm and the prediction it gave was what type of malware or malicious it was.

```
    "dst_pkts": 31,
    "dst_ip_bytes": 3832,
    "dns_query": "-",
    "dns_qclass": 0,
    "dns_qtype": 0,
    "dns_rcode": 0,
    "dns_AA": "-",
    "dns_RD": "-",
    "dns_RA": "-",
    "dns_rejected": "-",
    "ssl_version": "-",
    "ssl_cipher": "-",
    "ssl_resumed": "-",
    "ssl_established": "-",
    "ssl_subject": "-",
    "ssl_issuer": "-",
    "http_trans_depth": "-",
    "http_method": "-",
    "http_uri": "-",
    "http_version": "-",
    "http_request_body_len": 0,
    "http_response_body_len": 0,
    "http_status_code": 0,
    "http_user_agent": "-",
    "http_orig_mime_types": "-",
    "http_resp_mime_types": "-",
    "weird_name": "-",
    "weird_addl": "-",
    "weird_notice": "-"
}

# Sample test data as provided
predicted_rf_labels, predicted_dt_labels, predicted_gbm_labels, predicted_xgb_labels = main(test_data)
print("Random Forest Predicted Labels:", predicted_rf_labels)
print("Decision Tree Predicted Labels:", predicted_dt_labels)
print("GBM Predicted Labels:", predicted_gbm_labels)
print("XGBoost Predicted Labels:", predicted_xgb_labels)
```

```
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/backend/fork_exec.py:38: RuntimeWarning: os
with multithreaded code, and JAX is multithreaded, so this will likely lead to a deadlock.
  pid = os.fork()
Random Forest Predicted Labels: [('0', '1')]
Decision Tree Predicted Labels: [('0', '1')]
GBM Predicted Labels: [('0', '5')]
XGBoost Predicted Labels: [('0', '1')]
```

This script uses several machine learning models to make predictions on test data. Using the load_models() function, it loads the required encoders and pre-trained models first. The preprocess_test_data() function is then used to preprocess the test data by removing any extraneous columns and applying encodings. Using the predict() function, predictions are made on the preprocessed test data using the RandomForest, DecisionTree, Gradient Boosting

Machine (GBM), and XGBoost models. Lastly, the predicted labels for each model are returned by the main() function, which calls the previously stated procedures to coordinate the entire process. The predictions are shown using the given sample test data, and each model's generated labels are printed out. Using several machine learning models, this script acts as a pipeline to generate predictions on fresh data.

XGB Classifier

```
from xgboost import XGBClassifier
from sklearn.multioutput import MultiOutputClassifier

def train_xgboost(X_train, y_train):
    """ Train an XGBoost model for multi-label classification """
    model = MultiOutputClassifier(XGBClassifier(objective='binary:logistic', eval_metric='logloss'))
    model.fit(X_train, y_train)
    joblib.dump(model, 'xgboost_multi_output.joblib')
    return model

def main_xgboost():
    data_path = 'train_test_network.csv'
    data = load_data(data_path)
    data_cleaned, dropped_cols = clean_data(data)
    data_preprocessed = preprocess_data(data_cleaned, '.')
    X_train, X_test, y_train, y_test = split_data(data_preprocessed)
    model_xgboost = train_xgboost(X_train, y_train)
    evaluate_model("XGBoost", model_xgboost, X_test, y_test)

if __name__ == "__main__":
    main_xgboost()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was ren
and will be removed in 1.4. `sparse_output` is ignored unless you leave `sparse` to its default value.
  warnings.warn(
Metrics for XGBoost:
F1 Score (macro): 0.9644851271098298
Precision (macro): 0.9691552838952502
Recall (macro): 0.9600333966888861
ROC AUC (avg): 0.9794076753554928
```

The process of training and assessing a multi-label classification model with the well-known machine learning framework XGBoost. The required modules, including MultiOutputClassifier from scikit-learn and XGBClassifier from XGBoost, are initially imported. Next, it specifies routines for both the primary execution and multi-label data training of the XGBoost model. The dataset is loaded from a CSV file into the main function, where it is cleaned and preprocessed, divided into training and testing sets, the XGBoost model is trained using the training data, and its performance is assessed using the testing data. It prints the evaluation results at the end. If

necessary, the model parameters and dataset can be changed to suit a variety of multi-label classification jobs, making it simple to customise and expand this script.

Finding Accuracy and Metrics of Gradient Boosting Model

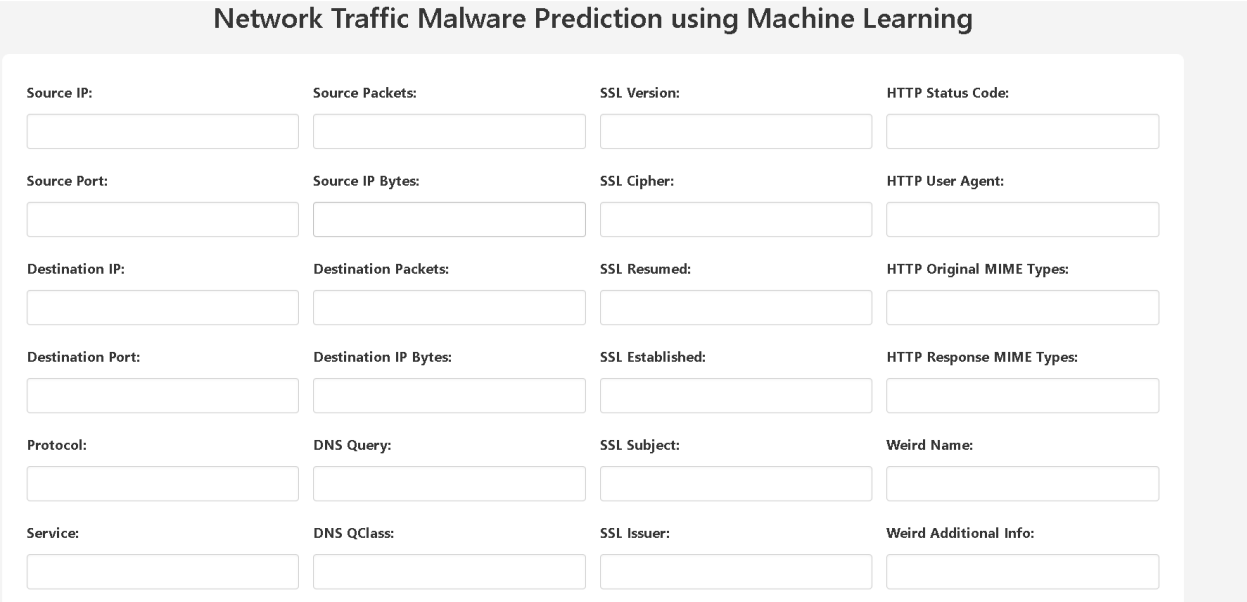
```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.multioutput import MultiOutputClassifier

def train_gbm(X_train, y_train):
    """ Train a Gradient Boosting Machine model for multi-label classification """
    model = MultiOutputClassifier(GradientBoostingClassifier())
    model.fit(X_train, y_train)
    joblib.dump(model, 'gbm_multi_output.joblib')
    return model

def main_gbm():
    data_path = 'train_test_network.csv'
    data = load_data(data_path)
    data_cleaned, dropped_cols = clean_data(data)
    data_preprocessed = preprocess_data(data_cleaned, '.')
    X_train, X_test, y_train, y_test = split_data(data_preprocessed)
    model_gbm = train_gbm(X_train, y_train)
    evaluate_model("Gradient Boosting Machine", model_gbm, X_test, y_test)

if __name__ == "__main__":
    main_gbm()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning
and will be removed in 1.4. `sparse_output` is ignored unless you leave `sparse` to its defau
  warnings.warn(
Metrics for Gradient Boosting Machine:
F1 Score (macro): 0.9114103128593152
Precision (macro): 0.9654214260784482
Recall (macro): 0.8969720699244679
ROC AUC (avg): 0.947226389983453
```

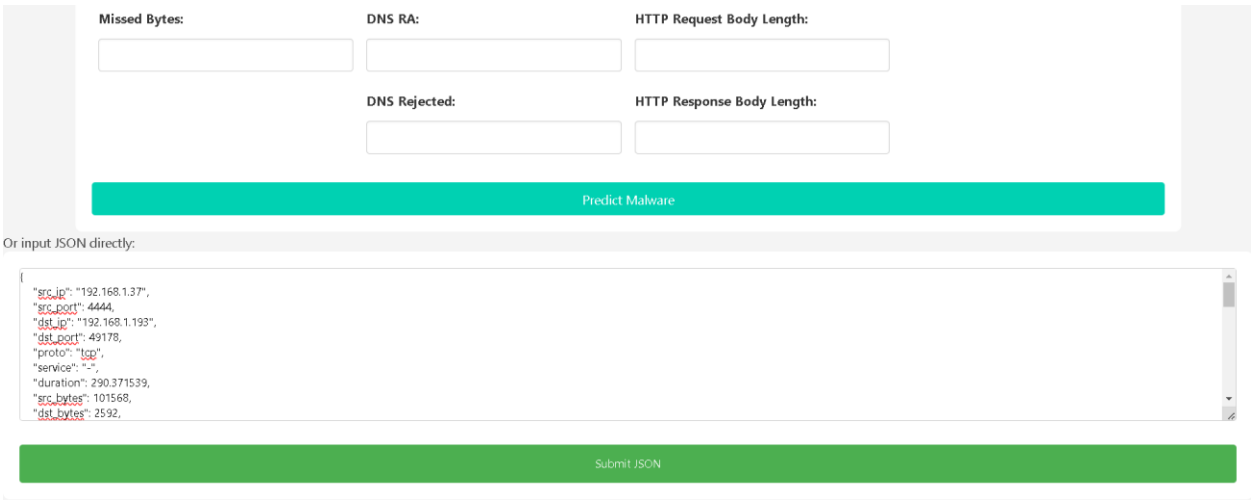
It imports the required modules, including MultiOutputClassifier for multi-label classification and GradientBoostingClassifier from scikit-learn. The GBM model is trained on multi-label data using the train_gbm function, which makes use of scikit-learn's MultiOutputClassifier to manage numerous target variables. Loading a dataset from a CSV file, cleaning and preprocessing the

data, dividing the data into training and testing sets, training the GBM model with the training data, and assessing its performance with the testing data are all done inside the main function main_gbm. Lastly, the evaluation results are printed out. A framework for applying GBM to multi-label classification tasks is provided by this script, which may be modified to suit various datasets and model parameters.

Frontend : the frontend is done through Flask with installing all the required plugins and applications and we have done the normal basic screen where the user can give the single line of record of the dataset in the input screen or can directly paste the sample JSON file in the dialog box by the end.



As like the picture below we can directly give the JSON file and check the type of Malware that affected the system.



Expected outputs: based on the input or JSON file given the Model that created will look for the ML models created and give the type of malware that got injected .


```
{
    "Decision Tree": {
        "decoded_type": [
            [
                "normal"
            ]
        ],
        "predictions": [
            [
                "0",
                "1"
            ]
        ]
    },
    "GBM": {
        "decoded_type": [
            [
                "normal"
            ]
        ],
        "predictions": [
            [
                "0",
                "5"
            ]
        ]
    },
    "Random Forest": {
        "decoded_type": [
            [
                "normal"
            ]
```

Conclusion:

All measures showed that the models performed well. The Random Forest classifier demonstrated its efficacy in managing the complexity and variability of the data by achieving the greatest F1 and ROC AUC values. XGBoost and Decision Tree both did admirably, demonstrating their capacity to identify relationships in the data. Despite having a marginally lower F1 score, GBM was still able to achieve good ROC AUC values and high precision.

To offer more detailed insights into how each model performed across various classes, confusion matrices for each model were displayed. These matrices were helpful in finding any biases or model flaws, especially with regard to false positives and false negatives.
This study successfully used multi-label classification algorithms to network traffic data through thorough preprocessing, careful model selection, and thorough evaluation.

The models created have the potential to greatly improve network security systems by enabling the accurate and concurrent identification of various network threats and behaviours. Future research might examine applying more sophisticated neural network designs to improve predictive performance or delving deeper into the integration of model predictions into real-time network security solutions.

References

Data Sources: Network traffic and application behavior data from Android applications.

Algorithms and Libraries: Scikit-learn for machine learning models, TensorFlow for neural networks.