Sage Plagarism Report

Powered by schoolworkspro.com

Hari Sharan Shrestha
150212
Word Count: 3924

37 Results Found

15.38 % Match Percentage

Submitted to: Sunway	**1.18%**
Submitted to: Softwarica	**1.04%**
Submitted to: Sunway	**0.87%**
Submitted to: Softwarica	**0.86%**
Submitted to: Softwarica	**0.69%**
Submitted to: Softwarica	**0.68%**
Submitted to: Softwarica	**0.61%**
Submitted to: Softwarica	**0.54%**
Submitted to: Texas	**0.51%**
Source: SMOTE | Towards Data Science Link: https://towardsdatascience.com/smote-fdce2f605729	**0.48%**
Submitted to: Softwarica	**0.48%**
Submitted to: Softwarica	**0.46%**
Submitted to: Softwarica	**0.43%**
Submitted to: Softwarica	**0.43%**
Submitted to: Softwarica	**0.41%**

Source: Decision Tree Algorithm, Explained - KDnuggets Link: https://www.kdnuggets.com/2020/01/decision-tree-algorithm-explained.html	**0.4%**
Submitted to: Softwarica	**0.36%**
Submitted to: Softwarica	**0.36%**
Submitted to: Softwarica	**0.33%**
Submitted to: Softwarica	**0.33%**
Submitted to: Bktscl	**0.33%**
Submitted to: Softwarica	**0.31%**
Source: Spark SQL and DataFrames Spark 220 Documentation Link: https://spark.apache.org/docs/2.2.0/sql-programming-guide.html	**0.31%**
Submitted to: Softwarica	**0.28%**
Submitted to: Softwarica	**0.28%**
Source: What is Overfitting in Deep Learning [+10 Ways to Avoid It] Link: https://www.v7labs.com/blog/overfitting	**0.25%**
Submitted to: Softwarica	**0.23%**
Submitted to: Softwarica	**0.23%**
Submitted to: Softwarica	**0.23%**
Submitted to: Softwarica	**0.2%**
Submitted to: Softwarica	**0.2%**
Submitted to: Softwarica	**0.2%**
Submitted to: Softwarica	**0.2%**
Submitted to: Sunway	**0.2%**
Submitted to: Softwarica	**0.18%**

REFLECTIVE REPORT
Android Malware Detection using
PySpark and Machine learning algorithm Logistic regression or
Random Forest
STW7082CEM – Big Data
Management and Data

Name: Hari sharan shrestha College ID: 150212
GitHub Repository Link: https://github.com/softwarica-github/Hari_sharan_shrestha_150212_stw7082cem

Contents

Android malware detection is a critical aspect of mobile security due to the widespread use of Android devices and the increasing number of malicious apps targeting them.

This documentation aims to provide an overview of the implementation of an Android malware detection system using machine learning techniques.

It covers the implementation process, discussions on findings, and concludes with insights into the effectiveness of the approach.

The rapid proliferation of Android devices has led to an alarming increase in mobile malware threats.

Malicious apps can compromise user data, privacy, and device functionality.

Therefore, effective Android malware detection is crucial to ensure user security.

This documentation presents a comprehensive approach to Android malware detection using PySpark, a powerful tool for large-scale data processing, and machine learning algorithms such as Logistic Regression and Random Forest.

Malicious software in android is designed targeting the devices running the Android Operating system.
Like other types of malwares, malware for android system is also developed with the primary motto to perform unauthorized access to the android system to perform malicious system such as access the information, control devices.
It can be observed that, people these days are using smart phone for emails, online financial transactions which has raised the critical security issue.
One of the precious methods of detection and identification of the malicious software is by monitoring the network on the android devices.
The data gathered during monitoring can be provided as input in the learning engine and use it to learn themselves.
Machine learning algorithms can be used to evaluate the incoming data to the device which further predicts and detect the anomalies.
Further, by building the predictive model that analyze the pattern associated with the malware behavior can be applied to identify malicious android applications.
Problem Statement
With the rapid advance in the use of technology, threats carried along are also spiked.
Online transaction, important credentials and information are stored in the mobile phone which might be accessed by the malicious software and information might carried over to the fraud.
So, the primary goal of the project is to analyze the android data using Spark and form the available data, use of machine learning algorithm to predict the malicious software.
The result of the project is expected to detect the malicious software in the android platform which can be used to prevent the installation of the suspicious applications.
Aims/ Objective
The primary aim of the application is to make the use of Spark and machine learning algorithms to detect malicious activities in the android operating system.
Implementation Part
The code is a PySpark-based script for performing binary classification of Android malware detection using both Logistic Regression and Random Forest classifiers.
The dataset is loaded from a CSV file, preprocessed, and then used to train and evaluate the mentioned classification models.
The accuracy and precision metrics are calculated for both models and stored in dictionaries.
Here's a summary of what the code does:
- Import Required Libraries: Import necessary libraries and modules from PySpark for data processing, machine learning, and evaluation.
- Initialize Spark Session: Create a Spark session with a specified application name.
- Load CSV Data: Read the CSV file "Android_Malware_Data.csv" into a DataFrame named 'data'. The first row is used as the header, and the data types of columns are inferred.
- Clean Data: Remove leading and trailing spaces from column names and rename a specific column.
- Data Preprocessing: Prepare the data for machine learning.
Create a list of feature columns by excluding the irrelevant columns ('Flow ID', 'Label', etc.).
Use VectorAssembler to combine these features into a single 'features' column.
- Label Indexing: Convert the categorical labels into numerical values using StringIndexer.
- Split Data: Split the preprocessed data into training and testing sets using a 70-30 split ratio.
- Train Logistic Regression Model: Train a Logistic Regression model using the training data.
Specify parameters like features column, label column, regularization parameters, and others.

- Make Predictions with Logistic Regression: Use the trained Logistic Regression model to make predictions on the test data.
- Evaluate Logistic Regression Model: Use the MulticlassClassificationEvaluator to evaluate the Logistic Regression model's accuracy and precision on the test data.
- Train Random Forest Model: Train a Random Forest classifier using the training data.
- Make Predictions with Random Forest: Use the trained Random Forest model to make predictions on the test data.
- Evaluate Random Forest Model: Evaluate the Random Forest model's accuracy and precision on the test data using the MulticlassClassificationEvaluator.
- Store Metrics in Dictionaries: Store the calculated accuracy and precision metrics for both models in separate dictionaries, 'lr_dict' and 'rf_dict'.
Dataset Selection and Preprocessing: The dataset contain various columns representing features extracted from Android apps.
These features include aspects such as permissions requested by the app, API calls made, resource usage, network behavior, and more.
The dataset have a column labeled 'Label' indicating whether an app is considered malware or benign.
Relevance to Android Malware Detection: The dataset's relevance lies in its potential to provide insights into the characteristics and behaviors of malicious Android apps.
By analyzing the features of both malware and benign apps, machine learning models can learn to distinguish between the two and predict whether a given app is likely to be harmful.
This is crucial for protecting users from potentially harmful apps that may compromise their devices or personal data.
Preprocessing Steps:
- Loading Data: The dataset is loaded using the SparkSession and DataFrame API provided by PySpark.
- Handling Missing Values: Missing values are often problematic for machine learning algorithms.
The code you provided removes rows with missing values using the dropna() function.
This can help ensure that the dataset used for training and evaluation is complete.
- Data Cleaning: Data cleaning involves tasks like removing leading/trailing spaces from column names, renaming columns, and other operations to ensure consistent and meaningful column names.
- Feature Selection: In the preprocessing section, features are selected for model training.
The line feature_columns = [col_name for col_name in data.columns if col_name not in ['Flow ID','Label','Source IP', ...]] is used to select the columns to be considered as features.
It's essential to choose relevant features that contribute to the classification task and remove any irrelevant or redundant ones.
- Feature Engineering: The VectorAssembler is used to combine selected features into a single 'features' column.
This step prepares the data for model training by creating a format suitable for machine learning algorithms.
- Label Indexing: The 'Label' column containing categorical values (malware or benign) is transformed into numerical labels using the StringIndexer.
This numeric representation is necessary for the machine learning models to process the labels.
- Data Splitting: The dataset is divided into training and testing sets using a 70-30 split ratio.

This allows for training the models on one subset and evaluating their performance on another.
- Overall, the preprocessing steps are geared towards preparing the dataset for training machine learning models.
These steps ensure that the data is in a suitable format, contains relevant features, and is ready for use in building and evaluating classification models for Android malware detection.
Feature engineering: This is a critical step in building effective machine learning models, especially for tasks like Android malware detection.
Relevant features provide the necessary information to the model to differentiate between benign and malicious apps.
In the context of Android malware detection, features often include aspects related to permissions, API calls, intent filters, resource usage, and more.
Here's how feature engineering is typically done for such a task: 1.
Permissions: Permissions are an essential aspect of Android apps.
Malicious apps may request sensitive permissions that they don't actually need.
Relevant features derived from permissions might include: - Total number of permissions requested by the app.
- Specific dangerous permissions requested (e.g., access to SMS, camera, microphone).
- Ratio of normal to dangerous permissions.
API Calls: API calls made by an app can reveal its behavior.
Malware might use unusual API calls or those associated with unauthorized actions.
Features based on API calls include: - Count of unique API calls used by the app.
- Frequencies of specific critical API calls.
- Behavior patterns based on sequences of API calls.
Intent Filters: Intent filters define the types of intents an app can respond to.
Analyzing these can uncover the app's functionality.
Features be: - Number of intent filters registered.
- Specific intent filter actions and categories.
- Correlations between intent filters and other features.
Resource Usage: Resource-intensive apps be suspicious.
Features might include: - CPU and memory usage characteristics.
- Data usage patterns (if relevant).
Code Analysis: Analyzing the source code or disassembled code (in the case of APKs) can reveal insights into the app's logic and behavior.
Features from code analysis might include: - Presence of obfuscated code.
- Dynamic class loading.
- Use of reflection.
Network Behavior: Malicious apps might communicate with command and control servers or send sensitive information over the network.
Features from network behavior include: - Number of network requests.
- IP addresses and domains contacted.
- Protocol usage (HTTP, HTTPS, etc.).
- Data volume transmitted.
Manifest Information: The AndroidManifest.xml file contains metadata about the app.
Features might include: - Activities, services, and receivers declared.
- Use of permissions in the manifest.
- Version information.
Feature Selection: Selecting relevant features involves domain knowledge and experimentation.

Techniques like correlation analysis, mutual information, and model-based feature importance can aid in selecting the most discriminative features.
Transformation: Raw data, such as permission strings or API call sequences, needs to be transformed into a format suitable for machine learning algorithms.
This might involve one-hot encoding, vectorization, or creating custom numerical representations.
VectorAssembler is used to combine selected features into a single 'features' column, which is a common way to transform multiple feature columns into a format that can be consumed by machine learning algorithms.
Overall, feature engineering is a blend of domain expertise, data understanding, and creative thinking to represent the most important aspects of the data in a way that enables machine learning models to make accurate predictions
Implementation of Logistic Regression: In your provided code, the Logistic Regression algorithm is implemented using the following steps:
# Train Logistic Regression model lr = LogisticRegression(featuresCol='features', labelCol='label', regParam=0.0,
maxIter=10, tol=1e-4, elasticNetParam=0.25)
lr_model = lr.fit(train_data)
Here's what's happening: - The LogisticRegression class is instantiated, where you specify parameters like featuresCol (name of the feature column), labelCol (name of the label column), regularization parameter (regParam), maximum number of iterations (maxIter), convergence tolerance (tol), and elastic net mixing parameter (elasticNetParam).
- The .fit(train_data) method is called to train the Logistic Regression model on the training data.
The resulting lr_model is a trained Logistic Regression model.
Training and Testing Phases: In your code, you split the data into training and testing sets using the following line: python
train_data, test_data = data.randomSplit([0.7, 0.3], seed=123)
This randomly splits the data into approximately 70% for training (train_data) and 30% for testing (test_data).
The seed parameter ensures reproducibility.
Model Evaluation Metrics: MulticlassClassificationEvaluator to evaluate the model's performance:
evaluator = MulticlassClassificationEvaluator(labelCol='label', metricName='accuracy') precision = MulticlassClassificationEvaluator(labelCol='label',
metricName='weightedPrecision')
lr_accuracy = evaluator.evaluate(lr_predictions) lr_precision = precision.evaluate(lr_predictions)
The evaluator calculates the accuracy of the model's predictions on the test data.
The precision evaluator calculates the weighted precision, which takes into account class imbalances.
Implementation of Random Forest: # Train a Random Forest classifier rf_classifier = RandomForestClassifier(featuresCol='features', labelCol='label') model = rf_classifier.fit(train_data)
Make predictions on the test data predictions = model.transform(test_data)
Here's how it works: - You instantiate a RandomForestClassifier with specified parameters like featuresCol (name of the feature column) and labelCol (name of the label column).
- The .fit(train_data) method trains the Random Forest model on the training data.

The resulting model is a trained Random Forest classifier.
- The trained model is then used to make predictions on the test data using the .transform(test_data) method.
Discussion of Findings
Accuracy and Precision: Certainly, let's compare the performance of the Logistic Regression and Random Forest models using various evaluation metrics: 1.
Accuracy: Accuracy measures the proportion of correctly classified instances out of the total instances.
Precision: Precision measures the ratio of true positive predictions to the total predicted positives.
It's a measure of how many of the predicted positive cases were actually positive.
Now, Logistic Regression Metrics: - Accuracy: 0.700870 - Precision: 0.700042 Random Forest Metrics: - Accuracy: 0.677101 - Precision: 0.667586 Comparison and Interpretation: - The Logistic Regression outperforms Random Forest in all metrics, indicating better overall performance.
- Both models have similar accuracy, but the Logistic regression model's accuracy is slightly higher, indicating that it makes fewer misclassifications.
- Logistic Regression has higher precision, suggesting that it strikes a better balance between avoiding false positives and capturing true positives.
In summary, based on the comparison of these metrics, the Logistic Regression appears to be the better-performing model for this Android malware detection task.
It demonstrates better accuracy, precision compared to the Random Forest model.
However, the choice of the best model consider factors like computational complexity and ease of interpretability, depending on the specific requirements and constraints of application.
Graphical Representation of SQL results: Group by Fwd Packet Length Max: The maximum length of forward packets might be an indicator of certain types of network activity associated with malware.
Malicious traffic might exhibit unusual packet size patterns.
Tableau presentation of forward packet length max:
Bwd Packet Length Max: Similarly, the maximum length of backward packets could provide insights into network behavior that is indicative of malware.
Tableau presentation of backward packet length max:
Forward IAT Total: The total of forward IAT could be a useful feature.
Malware might exhibit specific patterns in terms of the total forward IAT it initiates.
Flow Duration: The duration of a flow could be a useful feature.
Malware might exhibit specific patterns in terms of the duration of network flows it initiates.
Tableau presentation of flow duration:
Advantages of Using PySpark: Distributed Computing: PySpark utilizes the power of distributed computing, allowing it to process large datasets by distributing tasks across a cluster of machines.
This significantly improves processing speed and scalability.
Parallel Processing: PySpark enables parallel processing of data, where different tasks can be executed concurrently on different nodes.
This leads to faster data transformation and model training.
In-Memory Processing: PySpark's ability to cache data in-memory reduces the need for repetitive data loading from disk, leading to quicker access and processing.

Ease of Use: PySpark provides a user-friendly API for working with big data, making it accessible to data scientists and analysts familiar with Python.
Integration with MLlib: PySpark's MLlib library offers a wide range of machine learning algorithms that can handle large datasets.
This includes classification, regression, clustering, and more.
Scalability: PySpark can scale horizontally by adding more nodes to the cluster, enabling it to process even larger datasets efficiently.
Scalability of Models: Logistic Regression: Logistic Regression is a linear model and is well-suited for parallel processing.
PySpark's implementation of Logistic Regression can handle large datasets with ease, making it scalable to handle Android malware detection on big datasets.
Random Forest: Random Forest is an ensemble method that combines multiple decision trees.
Each tree can be trained independently, and the ensemble aggregates the results.
PySpark's implementation of Random Forest is designed for distributed processing, which makes it scalable for large datasets.
Real-Time Performance: While PySpark is excellent for processing large datasets, real-time performance might be a challenge for some use cases.
Real-time performance depends on factors such as the complexity of the model, the number of features, and the data preprocessing steps.
For Android malware detection, real-time performance can be maintained by: Data Sampling: Using a subset of the data for initial testing and model development can speed up the process. Once the model is optimized, it can be applied to the complete dataset.
Model Complexity: Simplifying the model architecture can improve real-time performance. For instance, using fewer features or a less complex ensemble for Random Forest.
Parallelization: Leveraging PySpark's parallel processing capabilities can help maintain reasonable processing times for real-time applications.
Optimization: Profiling and optimizing the code can identify bottlenecks and areas for improvement in terms of speed and resource usage.
In scenarios where real-time processing is crucial, a combination of techniques, such as batch processing for training and real-time processing for inference, can be employed to strike a balance between accuracy and speed
In conclusion, this documentation presents an in-depth exploration of Android malware detection using PySpark and machine learning algorithms.
The implementation of Logistic Regression and Random Forest models demonstrates their effectiveness in classifying Android apps as benign or malicious.
The findings underscore the importance of feature engineering, model evaluation in achieving accurate malware detection.
Logistic regression model provide good accuracy as compare to random forest model.
As the mobile threat landscape continues to evolve, leveraging PySpark and machine learning algorithms proves to be a promising approach to bolster Android security and protect user devices from potential threats.
datacamp, 2023.
Pyspark Tutorial: Getting Started with Pyspark.
[Online] Available at: https://www.datacamp.com/tutorial/pyspark-tutorial-getting-started-with-pyspark [Accessed 1 09 2023].
Detection, A. M., 2023.
Android Malware Detection: A Literature Review.

[Online] Available at: https://link.springer.com/chapter/10.1007/978-981-99-0272-9_18 [Accessed 31 08 2023].
geeksforgeeks, 2018.
Logistic Regression in Machine Learning.
[Online] Available at: https://www.geeksforgeeks.org/understanding-logistic-regression/ [Accessed 29 08 2023].
IBM, 203.
What is random forest.
[Online] Available at: https://www.ibm.com/topics/random-forest [Accessed 31 08 2023].
```
!pip install pyspark !pip install pandas
from pyspark.sql import SparkSession from pyspark.sql.functions import col from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import RandomForestClassifier from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.classification import LogisticRegression from pyspark.ml.evaluation import MulticlassClassificationEvaluator
import pandas as pd
# Initialize Spark session spark = SparkSession.builder \ .appName("Android Malware Detection") \ .getOrCreate()
# Load CSV file into DataFrame data = spark.read.csv('./Android_Malware_Data.csv', header=True, inferSchema=True)
# Remove leading spaces from column names for column_name in data.columns: new_column_name = column_name.strip() # Remove leading and trailing spaces data = data.withColumnRenamed(column_name, new_column_name)
data = data.withColumnRenamed('Fwd Header Length.1', 'Fwd Header Length_1')
# Drop rows with missing values data = data.dropna()
# Data preprocessing feature_columns = [col_name for col_name in data.columns if col_name not in ['Flow ID','Label','Source IP','Destination IP','Timestamp','CWE Flag Count','Down/Up Ratio','Fwd Avg Bytes/Bulk']]
assembler = VectorAssembler(inputCols=feature_columns, outputCol='features') data = assembler.transform(data)
# Label indexing (converting labels to numerical values) from pyspark.ml.feature import StringIndexer label_indexer = StringIndexer(inputCol='Label', outputCol='label') data = label_indexer.fit(data).transform(data)
# Split the data into training and testing sets train_data, test_data = data.randomSplit([0.7, 0.3], seed=123)
from pyspark.ml.tuning import ParamGridBuilder,TrainValidationSplit
# Train Logistic Regression model lr = LogisticRegression(featuresCol='features',labelCol="label",regParam=0.0,maxIter=10,to l=1e-4,elasticNetParam=0.25) lr_model = lr.fit(train_data)
# Make predictions on the test data lr_predictions = lr_model.transform(test_data)
# Evaluate the model evaluator = MulticlassClassificationEvaluator(labelCol="label", metricName="accuracy") precision = MulticlassClassificationEvaluator(labelCol='label', metricName='weightedPrecision') lr_accuracy = evaluator.evaluate(lr_predictions) lr_precision = precision.evaluate(lr_predictions) print("Logistic Regression Accuracy:", lr_accuracy)
print("Logistic Regression Precision:", lr_precision)
```

```
# Train a Random Forest classifier rf_classifier =
RandomForestClassifier(featuresCol='features', labelCol='label') model =
rf_classifier.fit(train_data)
# Make predictions on the test data
predictions = model.transform(test_data)
# Evaluate the model using MulticlassClassificationEvaluator evaluator =
MulticlassClassificationEvaluator(labelCol='label', predictionCol='prediction',
metricName='accuracy') precision = MulticlassClassificationEvaluator(labelCol='label',
predictionCol='prediction', metricName='weightedPrecision')
rf_accuracy = evaluator.evaluate(predictions) rf_precision = precision.evaluate(predictions)
print(f"Random Forest Accuracy: {rf_accuracy}") print(f"Random Forest Precision:
{rf_precision}")
lr_dict = {'Accuracy': lr_accuracy, 'Precision': lr_precision} rf_dict = {'Accuracy': rf_accuracy,
'Precision': rf_precision}
dicts = [lr_dict, rf_dict] results = pd.DataFrame(dicts) results['Models'] = ['Logistic Regression',
'Random Forest'] results.set_index(['Models'])
data.createOrReplaceTempView("android_malware") spark.sql(""" SELECT `am`.`Bwd Packet
Length Max`, SUM(CASE WHEN `am`.`Label` = 0 THEN 1 ELSE 0 END) AS AndroidAdware,
SUM(CASE WHEN `am`.`Label` = 1 THEN 1 ELSE 0 END) AS AndroidScareWare, SUM(CASE
WHEN `am`.`Label` = 2 THEN 1 ELSE 0 END) AS
AndroidSMSScareWare, SUM(CASE WHEN `am`.`Label` = 3 THEN 1 ELSE 0 END) AS Benign
FROM android_malware am GROUP BY `am`.`Bwd Packet Length Max` ORDER BY `am`.`Bwd
Packet Length Max` """).show()
from pyspark.sql import SparkSession import matplotlib.pyplot as plt
# Execute the SQL query query_result = spark.sql(""" SELECT `am`.`Bwd Packet Length Max`,
SUM(CASE WHEN `am`.`Label` = 0 THEN 1 ELSE 0 END) AS AndroidAdware, SUM(CASE WHEN
`am`.`Label` = 1 THEN 1 ELSE 0 END) AS AndroidScareWare, SUM(CASE WHEN `am`.`Label` = 2
THEN 1 ELSE 0 END) AS AndroidSMSScareWare, SUM(CASE WHEN `am`.`Label` = 3 THEN 1
ELSE 0 END) AS Benign FROM android_malware am GROUP BY `am`.`Bwd Packet Length Max`
ORDER BY `am`.`Bwd Packet Length Max` """).toPandas()
# Plot the bar graph plt.figure(figsize=(10, 6)) plt.bar(query_result["Bwd Packet Length Max"],
query_result["AndroidAdware"], label="Android Adware") plt.bar(query_result["Bwd Packet Length
Max"], query_result["AndroidScareWare"], label="Android ScareWare") plt.bar(query_result["Bwd
Packet Length Max"], query_result["AndroidSMSScareWare"], label="Android SMSScareWare")
plt.bar(query_result["Bwd Packet Length Max"], query_result["Benign"], label="Benign")
plt.xlabel("Bwd Packet Length Max") plt.ylabel("Count") plt.title("Malware Distribution by Bwd
Packet Length Max") plt.legend() plt.xticks(rotation=45) plt.tight_layout() plt.show()
data.createOrReplaceTempView("android_malware") spark.sql(""" SELECT `am`.`Fwd Packet
Length Max`, SUM(CASE WHEN `am`.`Label` = 0 THEN 1 ELSE 0 END) AS AndroidAdware,
SUM(CASE WHEN `am`.`Label` = 1 THEN 1 ELSE 0 END) AS AndroidScareWare, SUM(CASE
WHEN `am`.`Label` = 2 THEN 1 ELSE 0 END) AS
AndroidSMSScareWare, SUM(CASE WHEN `am`.`Label` = 3 THEN 1 ELSE 0 END) AS Benign
FROM android_malware am GROUP BY `am`.`Fwd Packet Length Max` ORDER BY `am`.`Fwd
Packet Length Max` """).show()
import pandas as pd import matplotlib.pyplot as plt
# Execute the SQL query and convert the result to a Pandas DataFrame query_result =
spark.sql(""" SELECT `am`.`Fwd Packet Length Max`, SUM(CASE WHEN `am`.`Label` = 0 THEN 1
```

```
ELSE 0 END) AS AndroidAdware, SUM(CASE WHEN `am`.`Label` = 1 THEN 1 ELSE 0 END) AS AndroidScareWare, SUM(CASE WHEN `am`.`Label` = 2 THEN 1 ELSE 0 END) AS AndroidSMSScareWare, SUM(CASE WHEN `am`.`Label` = 3 THEN 1 ELSE 0 END) AS Benign FROM android_malware am GROUP BY `am`.`Fwd Packet Length Max` ORDER BY `am`.`Fwd Packet Length Max` """).toPandas()
# Stop the Spark session spark.stop()
# Plot the data using matplotlib plt.plot(query_result["Fwd Packet Length Max"], query_result["AndroidAdware"], label="Android Adware") plt.plot(query_result["Fwd Packet Length Max"], query_result["AndroidScareWare"], label="Android ScareWare") plt.plot(query_result["Fwd Packet Length Max"], query_result["AndroidSMSScareWare"], label="Android SMSScareWare") plt.plot(query_result["Fwd Packet Length Max"], query_result["Benign"], label="Benign") plt.xlabel("Fwd Packet Length Max") plt.ylabel("Number of Samples") plt.title("Malware Detection by Fwd Packet Length Max") plt.legend() plt.grid() plt.show()
data.createOrReplaceTempView("android_malware") spark.sql(""" SELECT `am`.`Fwd IAT Total`, SUM(CASE WHEN `am`.`Label` = 0 THEN 1 ELSE 0 END) AS AndroidAdware, SUM(CASE WHEN `am`.`Label` = 1 THEN 1 ELSE 0 END) AS AndroidScareWare,
SUM(CASE WHEN `am`.`Label` = 2 THEN 1 ELSE 0 END) AS AndroidSMSScareWare, SUM(CASE WHEN `am`.`Label` = 3 THEN 1 ELSE 0 END) AS Benign FROM android_malware am GROUP BY `am`.`Fwd IAT Total` ORDER BY `am`.`Fwd IAT Total` """).show()
import pandas as pd import matplotlib.pyplot as plt
# Execute the SQL query query_result = spark.sql(""" SELECT `am`.`Fwd IAT Total`, SUM(CASE WHEN `am`.`Label` = 0 THEN 1 ELSE 0 END) AS AndroidAdware, SUM(CASE WHEN `am`.`Label` = 1 THEN 1 ELSE 0 END) AS AndroidScareWare, SUM(CASE WHEN `am`.`Label` = 2 THEN 1 ELSE 0 END) AS AndroidSMSScareWare, SUM(CASE WHEN `am`.`Label` = 3 THEN 1 ELSE 0 END) AS Benign FROM android_malware am GROUP BY `am`.`Fwd IAT Total` ORDER BY `am`.`Fwd IAT Total` """).toPandas()
# Plot the pie chart labels = ['Android Adware', 'Android ScareWare', 'Android SMSScareWare', 'Benign'] sizes = query_result.iloc[0, 1:] plt.pie(sizes, labels=labels, autopct='%1.1f%%', shadow=True, startangle=140) plt.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title("Malware Distribution by Fwd IAT Total") plt.show()
data.createOrReplaceTempView("android_malware") spark.sql(""" SELECT `am`.`Flow Duration`, SUM(CASE WHEN `am`.`Label` = 0 THEN 1 ELSE 0 END) AS AndroidAdware, SUM(CASE WHEN `am`.`Label` = 1 THEN 1 ELSE 0 END) AS AndroidScareWare,
SUM(CASE WHEN `am`.`Label` = 2 THEN 1 ELSE 0 END) AS AndroidSMSScareWare, SUM(CASE WHEN `am`.`Label` = 3 THEN 1 ELSE 0 END) AS Benign FROM android_malware am GROUP BY `am`.`Flow Duration` ORDER BY `am`.`Flow Duration` """).show()
```