

SOUTHERN ARKANSAS UNIVERSITY

COURSE

Information and Network security

Course Project: Android Malware Detection Using Machine Learning

By

Students of MCIS: Venkata Sai Siva Padmini Surekha Aripirala - 999903261

Keerthi Reddy Chimalapati - 999903554

Sai Kiran Duvvuri - 999902946

Professor

Izzat Alsmadi

Abstract

The pervasive use of mobile and electronic devices in settings ranging from metropolitan cities to rural villages has brought about a significant increase in data security risks, particularly through the loss or leakage of confidential information. A particularly pernicious threat is posed by malware, which targets devices primarily through Android applications. These applications, often downloaded and used by millions, can be weaponized to steal personal information or disrupt device functionality. The Android Malware Dataset, available on Kaggle, comprises feature vectors extracted from 15,036 Android applications, including 5,560 malware apps and 9,476 benign apps. This dataset provides a robust foundation for understanding and identifying malware threats, with 215 distinct attributes derived from API signatures, intents, command signatures, and manifest permissions.

In this project, we evaluate the effectiveness of machine learning techniques for Android malware detection. We employed a variety of classical machine learning methods—Logistic Regression (LR), Decision Trees (DT), Random Forest (RF), K-Nearest Neighbors (KNN), and Support Vector Classifier (SVC)—as well as a Deep Learning approach to analyze and classify the applications. Each model was rigorously trained and tested on the dataset, and their performances were benchmarked against one another. Metrics such as accuracy, recall, and ROC AUC were used to compare the models' capabilities in detecting malicious versus benign applications. This comparative analysis not only highlights the strengths and weaknesses of each method but also guides future efforts towards the most effective strategies for malware detection. The findings from this study aim to enhance the security measures in place to protect against Android malware, contributing to safer use of mobile technologies worldwide.

Introduction

In an era dominated by digital transformation, the use of Android devices has skyrocketed, integrating into nearly every aspect of daily life. This surge in connectivity, while beneficial, has also escalated the vulnerabilities associated with mobile technologies. Among these threats, Android malware presents a significant challenge, with malicious applications increasingly engineered to mimic benign software. These applications compromise user security by engaging in unauthorized activities such as stealing personal information, eavesdropping on communications, and hijacking device functionality.

The Android platform, owing to its open-source nature and extensive market share, is particularly susceptible to such attacks. Malware can be embedded within seemingly harmless applications, making it difficult for average users to detect. The consequences of such infections can be devastating, ranging from personal data loss to substantial financial theft. Given these risks, the development of robust machine learning models to detect and classify Android malware is not only relevant but essential.

This report delves into an Android malware dataset, which consists of feature vectors extracted from a wide array of applications. The dataset, sourced from Kaggle, includes data from 15,036 applications categorized into 5,560 malware and 9,476 benign instances. The features encompass API signatures, intent data, command signatures, and manifest permissions, amounting to 215 attributes in total. This comprehensive dataset provides a fertile ground for developing and evaluating various machine learning techniques.

Our study begins with the crucial step of data acquisition, followed by preprocessing to ensure the quality and usability of the data for model training. We employ a suite of machine learning algorithms, including traditional classifiers like Logistic Regression (LR), Decision Trees (DT), Random Forest (RF), K-Nearest Neighbors (KNN), and Support Vector Classifier (SVC), alongside advanced Deep Learning techniques. Each model is rigorously trained and evaluated based on its ability to accurately distinguish between malicious and benign applications, using metrics such as accuracy, recall, and the area under the receiver operating characteristic (ROC) curve.

By comparing these models, the study aims to not only identify the most effective techniques for Android malware detection but also to contribute valuable insights into the behavioral patterns of malware applications. This could potentially guide future developments in cybersecurity measures for mobile platforms.

Research Goal

The primary aim of this research is to enhance Android malware detection by employing and comparing various machine learning techniques. This project focuses on assembling a dataset of Android applications, identifying critical features indicative of malware, and experimenting with both classical and advanced machine learning models. We will evaluate these models based on accuracy, precision, recall, and F1-score to determine their effectiveness in identifying malware threats. Additionally, the research will explore the practical aspects of deploying these models into real-world settings, aiming to improve the security protocols on Android devices against the increasing threats of malicious software.

Methodology

The Dataset Android Malware Detection consists of:

- Data Collection
- Data Preprocessing
- Feature Engineering

Data Collection:

Importing the required python libraries (example: pandas, sklearn, scipy is for visualization) which are required for data loading, preprocessing and visualization. In the below two screenshots, the two data files are loaded into data frames using (read_csv).

The screenshot shows a Jupyter Notebook window titled 'INS_Project_Milestone1'. The interface includes a menu bar (File, Edit, View, Run, Kernel, Settings, Help) and a toolbar. A message at the top states: 'Package Importing: Imports required packages (pandas, numpy, tensorflow, matplotlib.pyplot, seaborn, LabelEncoder) for data manipulation, analysis, and visualization.'

The notebook contains two code cells:

```
[4]: # Import the installed packages
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
```

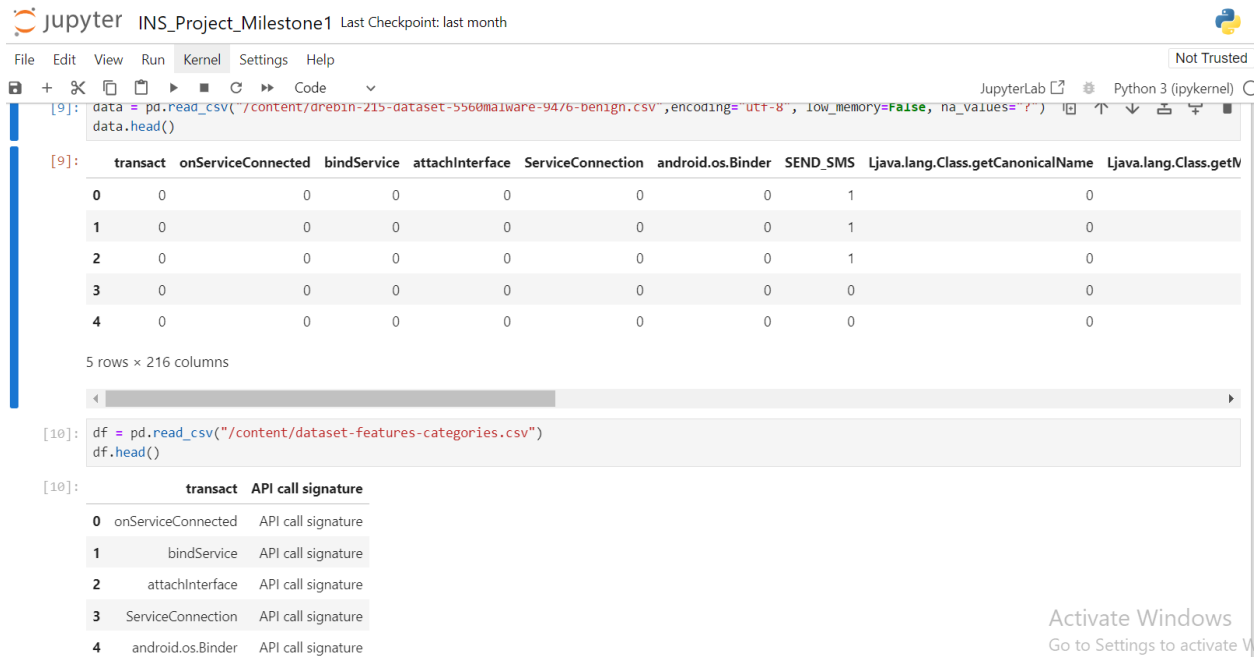
```
[9]: data = pd.read_csv("/content/drebin-215-dataset-5560malware-9476-benign.csv", encoding="utf-8", low_memory=False, na_values="?")
data.head()
```

The output of the second cell shows the first five rows of the CSV file as a table:

	transact	onServiceConnected	bindService	attachInterface	ServiceConnection	android.os.Binder	SEND_SMS	Ljava.lang.Class.getCanonicalName	Ljava.lang.Class.getV
0	0	0	0	0	0	0	1	0	
1	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0

Below the table, it indicates '5 rows x 216 columns'.

Android Malware Detection



JupyterLab interface showing the initial data loading and inspection. The code cell [9] loads a CSV file from the content directory. The output shows the first 5 rows of the dataset, which has 216 columns. The columns include transaction status, service connection, interface attachment, and various Android OS services. The code cell [10] loads a second CSV file containing feature categories, and its output shows the first 5 rows, which include transaction status and API call signatures.

```
[9]: data = pd.read_csv("/content/drebin-215-dataset-5560malware-9476-denign.csv", encoding="utf-8", low_memory=False, na_values="?",)
data.head()

[9]:
```

	transact	onServiceConnected	bindService	attachInterface	ServiceConnection	android.os.Binder	SEND_SMS	Ljava.lang.Class.getCanonicalName	Ljava.lang.Class.getV
0	0	0	0	0	0	0	1	0	
1	0	0	0	0	0	0	1	0	
2	0	0	0	0	0	0	1	0	
3	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	

5 rows × 216 columns

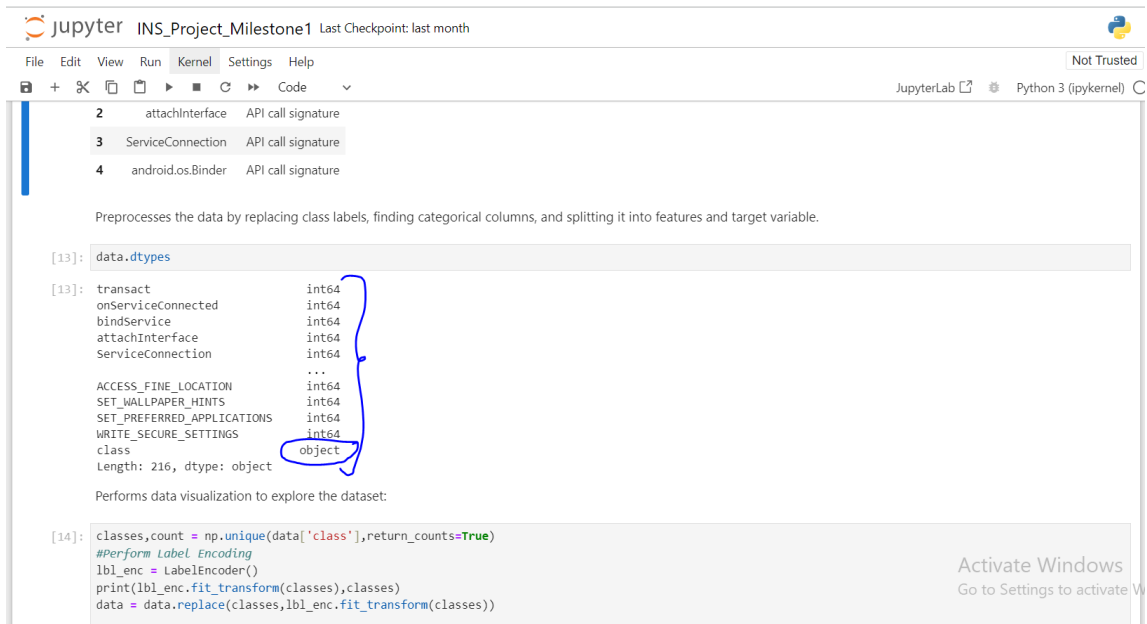
```
[10]: df = pd.read_csv("/content/dataset-features-categories.csv")
df.head()

[10]:
```

	transact	API call signature
0	onServiceConnected	API call signature
1	bindService	API call signature
2	attachInterface	API call signature
3	ServiceConnection	API call signature
4	android.os.Binder	API call signature

Data Preprocessing

- As part of preprocessing there are various methods to process the data like eliminating duplicates, finding missing values, discarding null values, changing data format as per requirement. This dataset has no null values.
- As part of this dataset out of 216 attributes only one attribute 'class' with data type **object**.



JupyterLab interface showing the data preprocessing steps. The code cell [13] displays the data types for the dataset, where the 'class' attribute is highlighted as 'object'. The code cell [14] performs label encoding on the 'class' attribute using LabelEncoder and replaces the original values with the encoded values.

```
[13]: data.dtypes

[13]:
```

	dtype
transact	int64
onServiceConnected	int64
bindService	int64
attachInterface	int64
ServiceConnection	int64
...	...
ACCESS_FINE_LOCATION	int64
SET_WALLPAPER_HINTS	int64
SET_PREFERRED_APPLICATIONS	int64
WRITE_SECURE_SETTINGS	int64
class	object

Length: 216, dtype: object

Preprocesses the data by replacing class labels, finding categorical columns, and splitting it into features and target variable.

```
[14]: classes, count = np.unique(data['class'], return_counts=True)
#Perform Label Encoding
lbl_enc = LabelEncoder()
print(lbl_enc.fit_transform(classes), classes)
data = data.replace(classes, lbl_enc.fit_transform(classes))
```

- c. In below screenshot following operations have been performed:
- The data & data type in 'class' attribute is converted to 'int' as (B-0,B-1) from 'object' to make analysis more precise. The reason to convert the data types is to eliminate any incompatibility between the attributes for the analysis.
 - Finding the 'missing' values. Also, having missing values may cause problem further like data inconsistency, incomplete analysis which can leads to incomplete conclusions in the project.
 - The special character, missing values, are setting them to NaN and are eliminated using 'dropna()'.

Jupyter INS_Project_Milestone1 Last Checkpoint: last month

```
[14]: classes,count = np.unique(data['class'],return_counts=True)
#Perform Label Encoding
lbl_enc = LabelEncoder()
print(lbl_enc.fit_transform(classes),classes)
data = data.replace(classes,lbl_enc.fit_transform(classes))

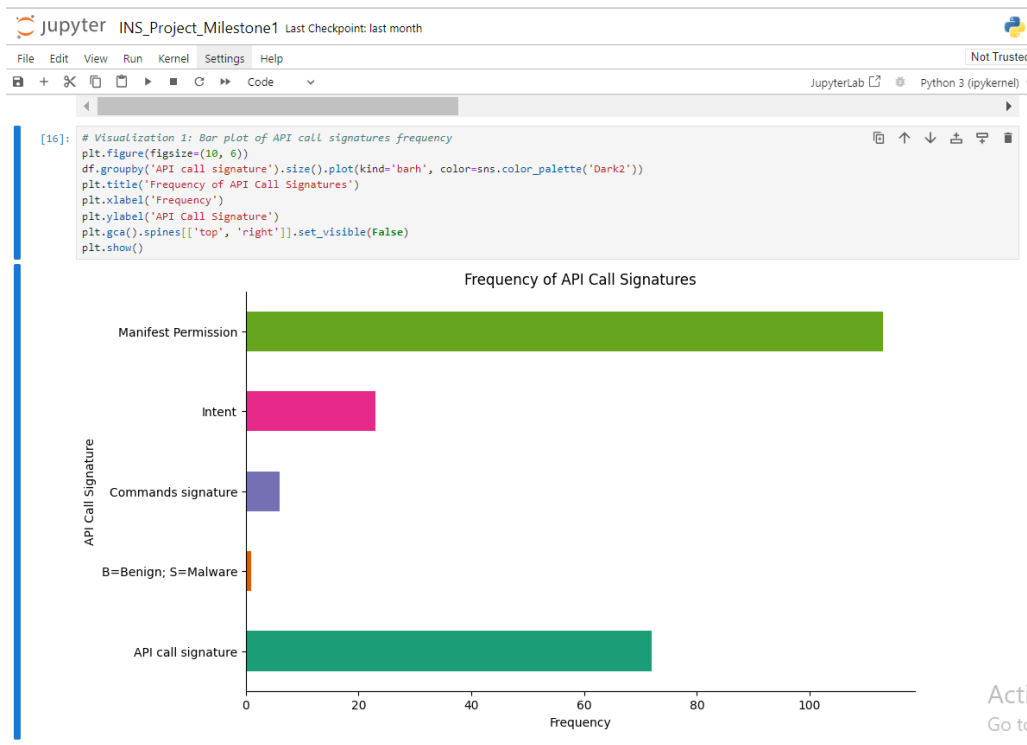
#Dataset contains special characters like '?' and 'S'. Set them to NaN and use dropna() to remove them
data=data.replace('[?S]',np.NaN,regex=True)
print("Total missing values : ",sum(list(data.isna().sum())))
data.dropna(inplace=True)
for c in data.columns:
    data[c] = pd.to_numeric(data[c])
data
```

[0 1] ['B' 'S']
Total missing values : 5

```
[14]: TEST ACCESS_WIFI_STATE WRITE_EXTERNAL_STORAGE ACCESS_FINE_LOCATION SET_WALLPAPER_HINTS SET_PREFERRED_APPLICATIONS WRITE_SECURE_SETTINGS class
```

TEST	ACCESS_WIFI_STATE	WRITE_EXTERNAL_STORAGE	ACCESS_FINE_LOCATION	SET_WALLPAPER_HINTS	SET_PREFERRED_APPLICATIONS	WRITE_SECURE_SETTINGS	class
0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	1
0	1	1	1	0	0	0	1
0	1	0	1	0	0	0	1
...
0	1	1	0	0	0	0	0
0	1	1	0	0	0	0	0

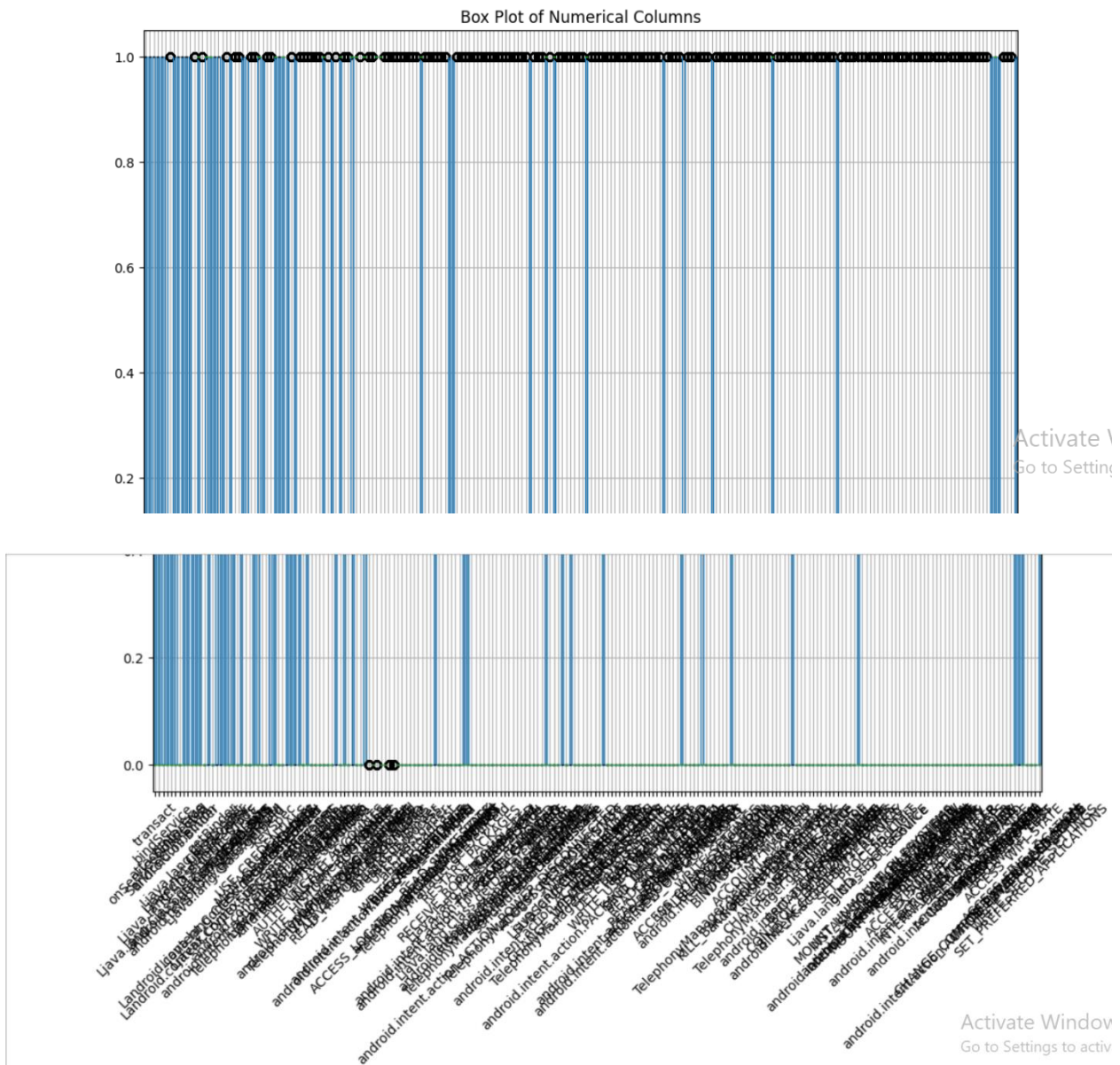
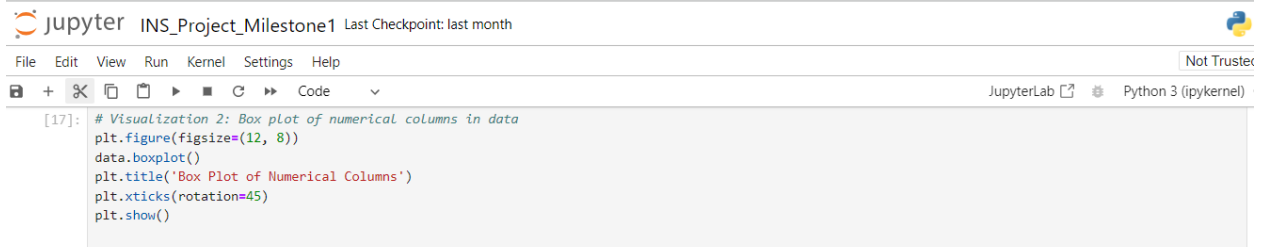
Data Visualization



Visualization 1 : Below graph is API call signatures frequency which show the highest frequency in Manifest Permission, and (Benign, Malware has the least frequency)

Visualization 2: Below graph is for numerical columns in dataframe (**data**). On x-axis we have attributes and in Y-axis we have numerical accuracy values.

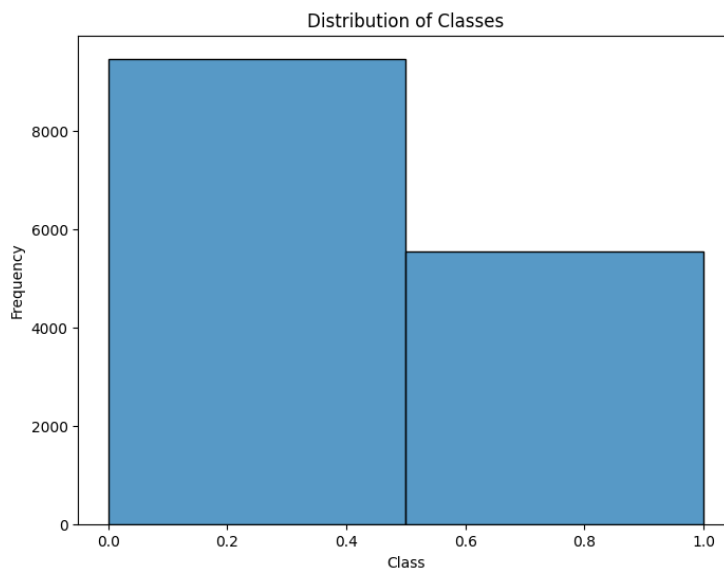
Android Malware Detection



Android Malware Detection

Visualization 3: Below graph is class distribution for data frame (**data**). The class is an attribute, (Benign and Malware) are its values and we have them on x-axis. On x-axis from 0.0 to 0.5 is Benign, 0.5 to 1.0 is Malware. On y-axis we have frequency which refers as number of records in data file for class attribute. This plot is representation differentiation on records for (Benign and Malware) values.

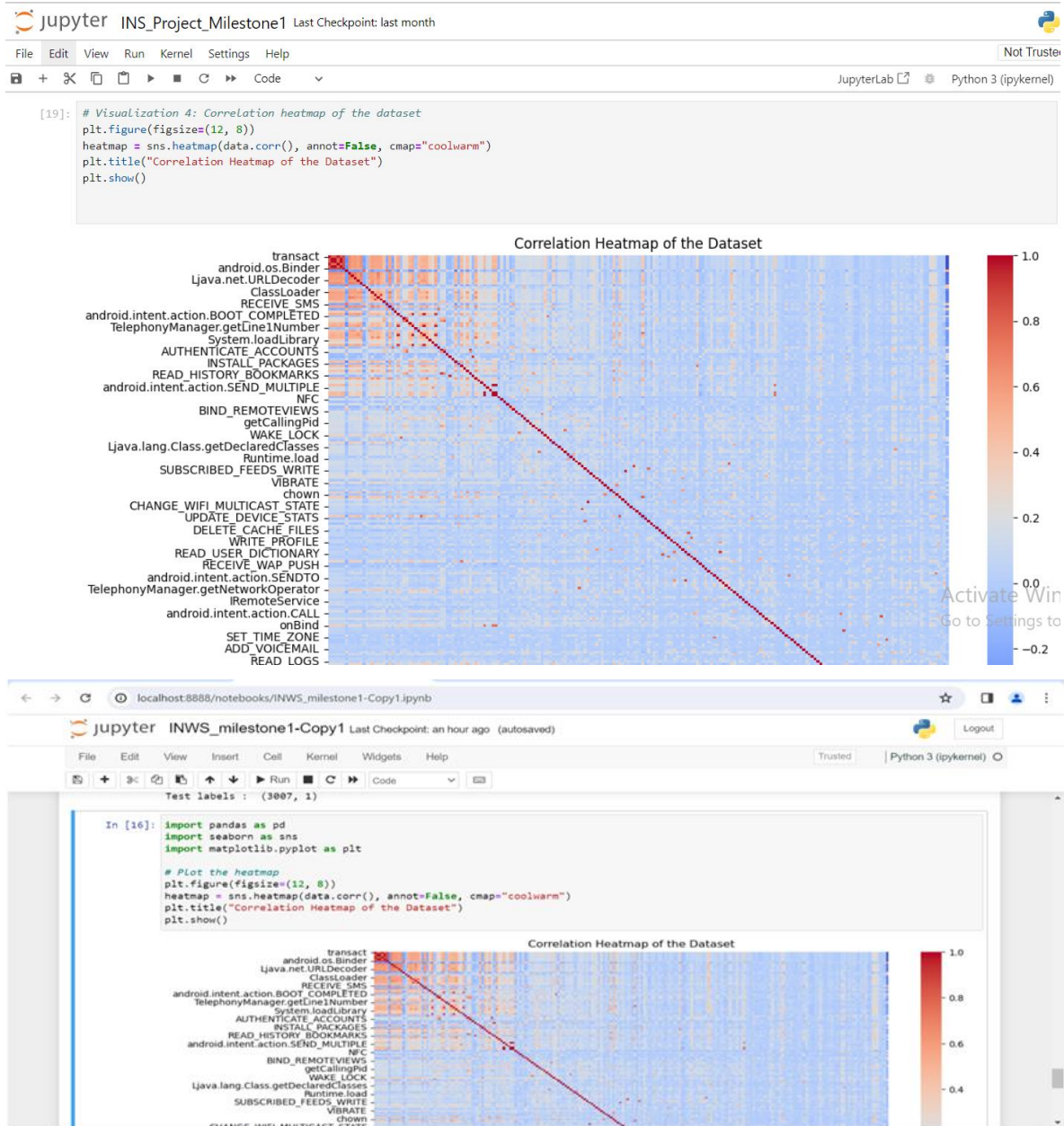
```
[18]: # Visualization 3: Histogram of class distribution
plt.figure(figsize=(8, 6))
sns.histplot(data['class'], bins=2, kde=False)
plt.title('Distribution of Classes')
plt.xlabel('Class')
plt.ylabel('Frequency')
plt.show()
```



Act
Go t

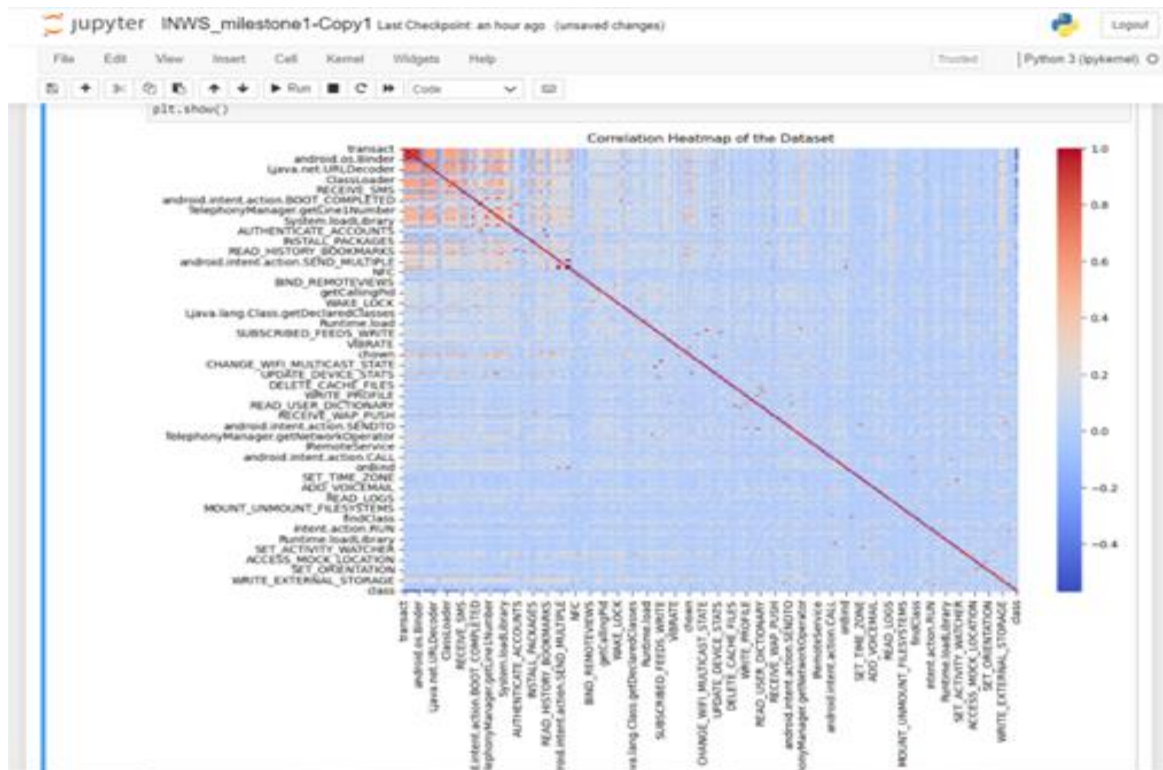
Android Malware Detection

Visualization 4: The below code for building the heat map visualization which represents relationship between features in the dataset. We call them attributes from the .csv files these are about the special privileges that apps must ask for user approval to when they want to access the sensitive information like location, contacts, recording, images etc.

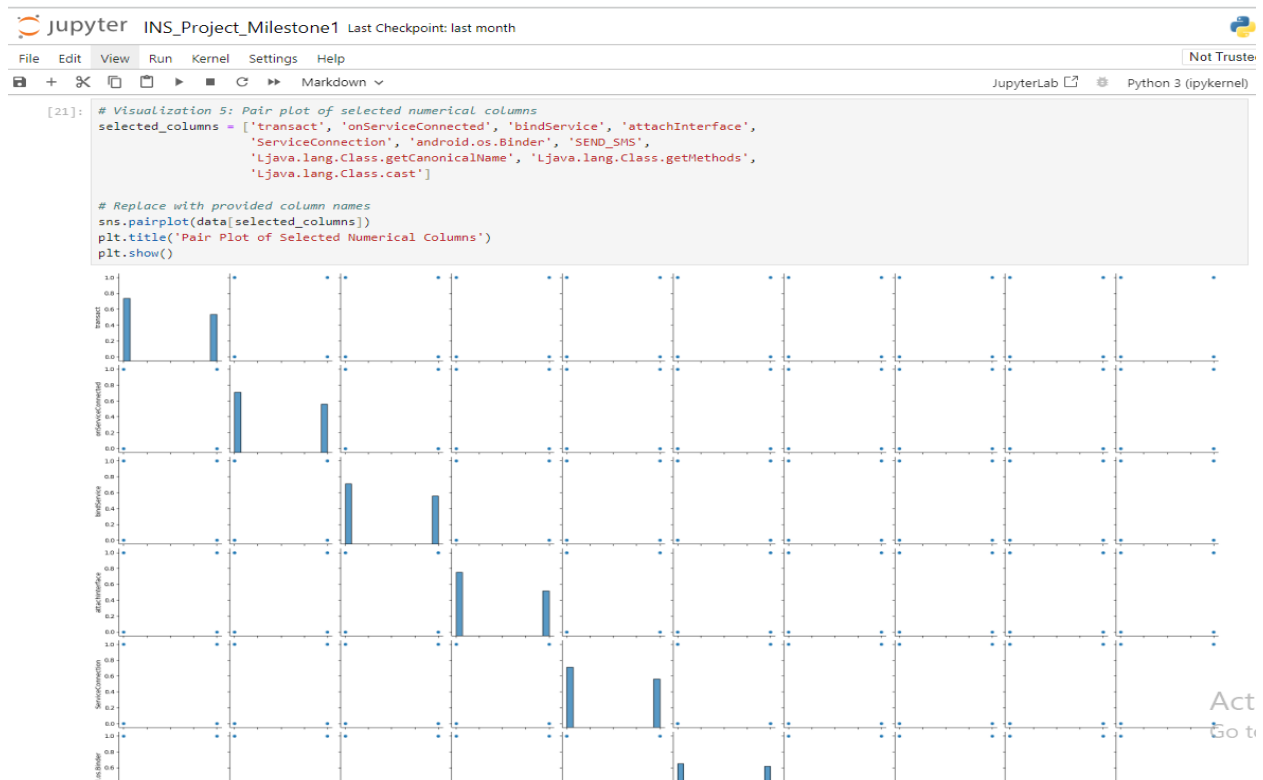


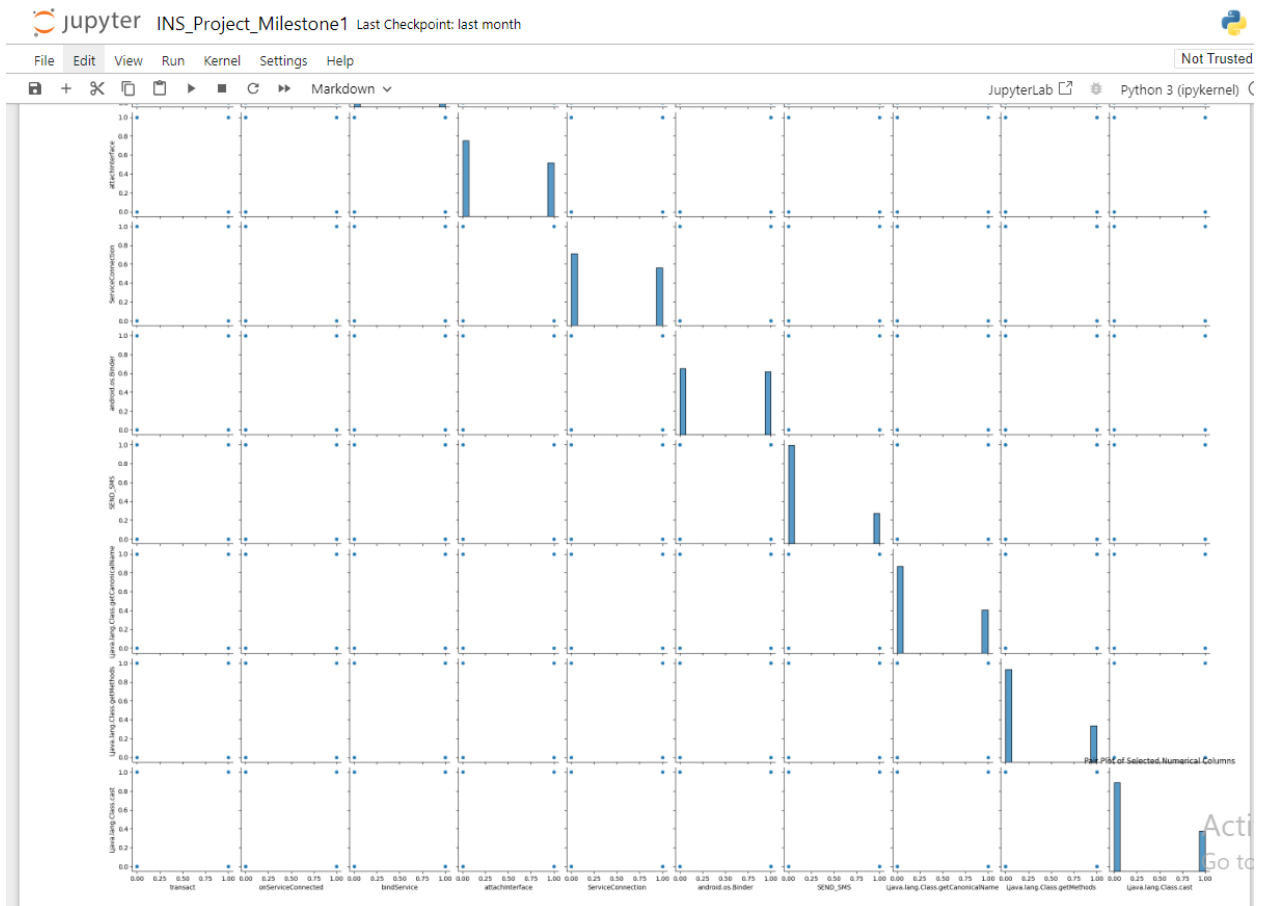
From this heat map understanding the dark red color indicates the strong relationship among attributes, blue color indicates the negative relationship and white color indicates the no significant in relationship. However, the heat map scale on the right side, which shows numerical values for correlation values from -1 to 1 where correlation is near to 1 is positive correlation and -1 is opposite for positive. These positive correlated attributes values can be used for further analysis to develop more insight on the dataset.

Android Malware Detection



Visualization 5: Below plot shows the relation among selected numerical columns.





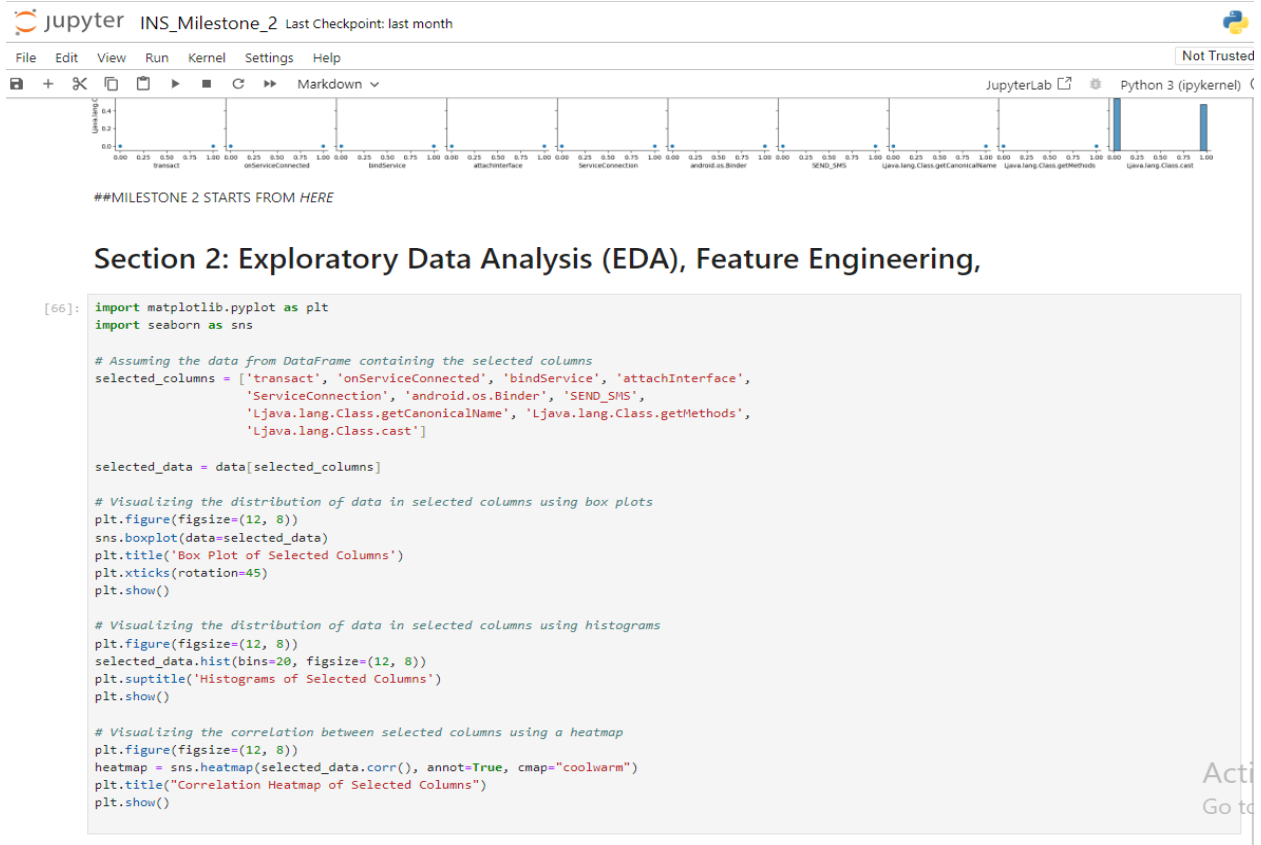
For the upcoming milestone, we need to understand the correlation between the attributes in detail like, how are they strongly correlated, and why there is no significant relationship between some of the attributes etc. In a nut shell the dataset is all about the malicious attacks on android applications mobiles and electronic devices and how they are affected by the attacks and how the confidential information is leaking.

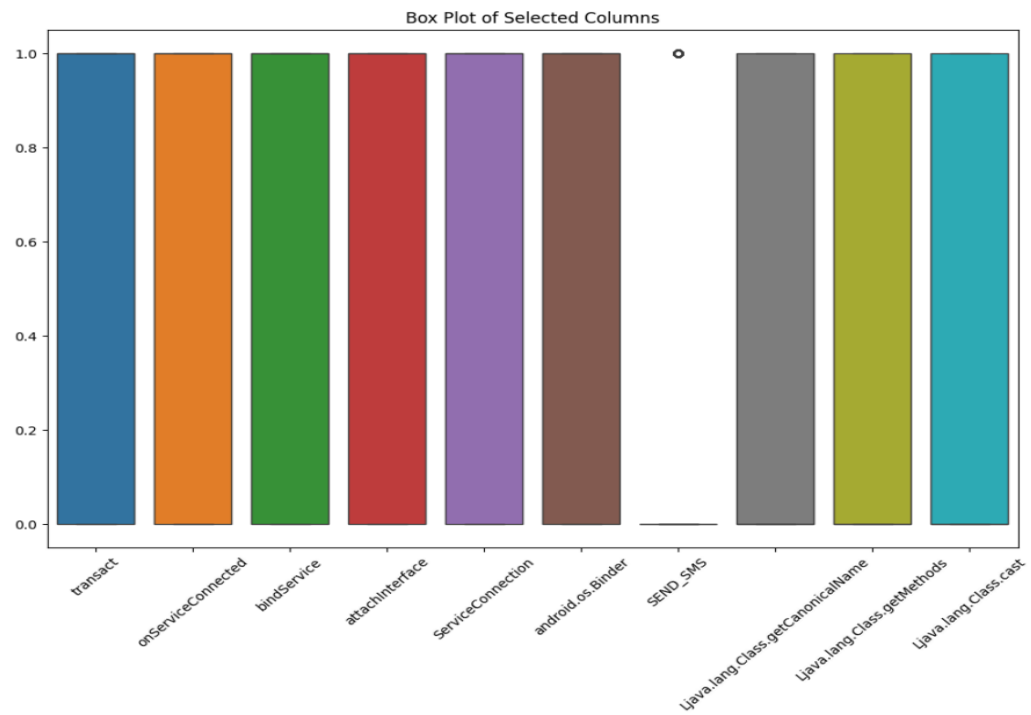
Feature Engineering

The chi-square (chi2) statistical test is what we use for feature selection. For categorical target variables such as "transact," the chi-square test is appropriate. Based on the chi-square scores of each attribute, we have chosen the top 5. After selecting features, we use Principal Component Analysis (PCA) to minimize the number of dimensions in the data. With the most significant information preserved, PCA converts the original characteristics into a lower-dimensional space. In order to facilitate visualization, we in this instance limit the dimensionality to two components.

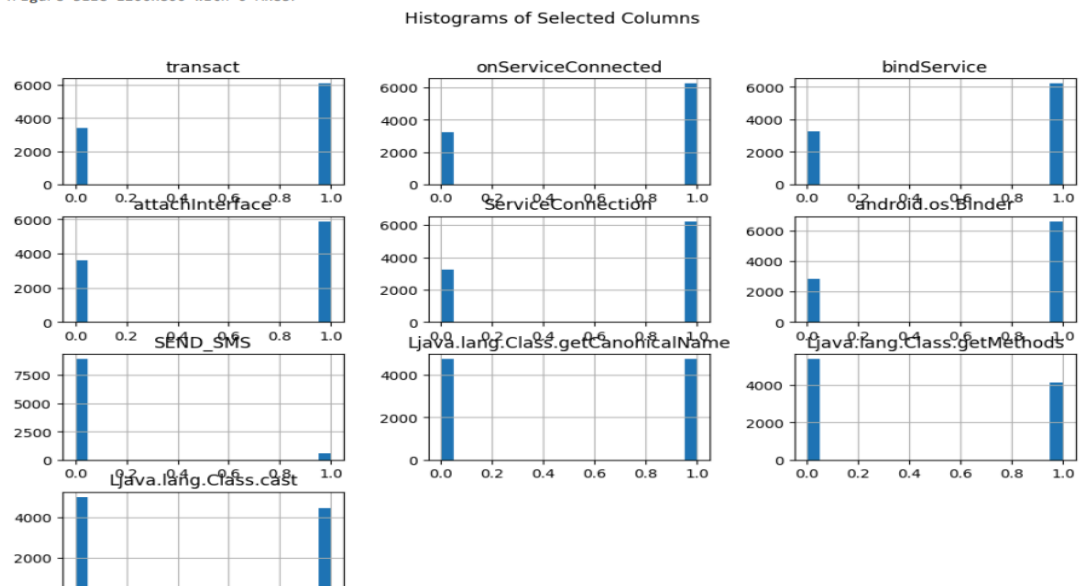
Android Malware Detection

The important features are 'transact', 'onServiceConnected', 'bindService', 'attachInterface', 'ServiceConnection', 'android.os.Binder', 'SEND_SMS', 'Ljava.lang.Class.getCanonicalName', 'Ljava.lang.Class.getMethods', 'Ljava.lang.Class.cast'

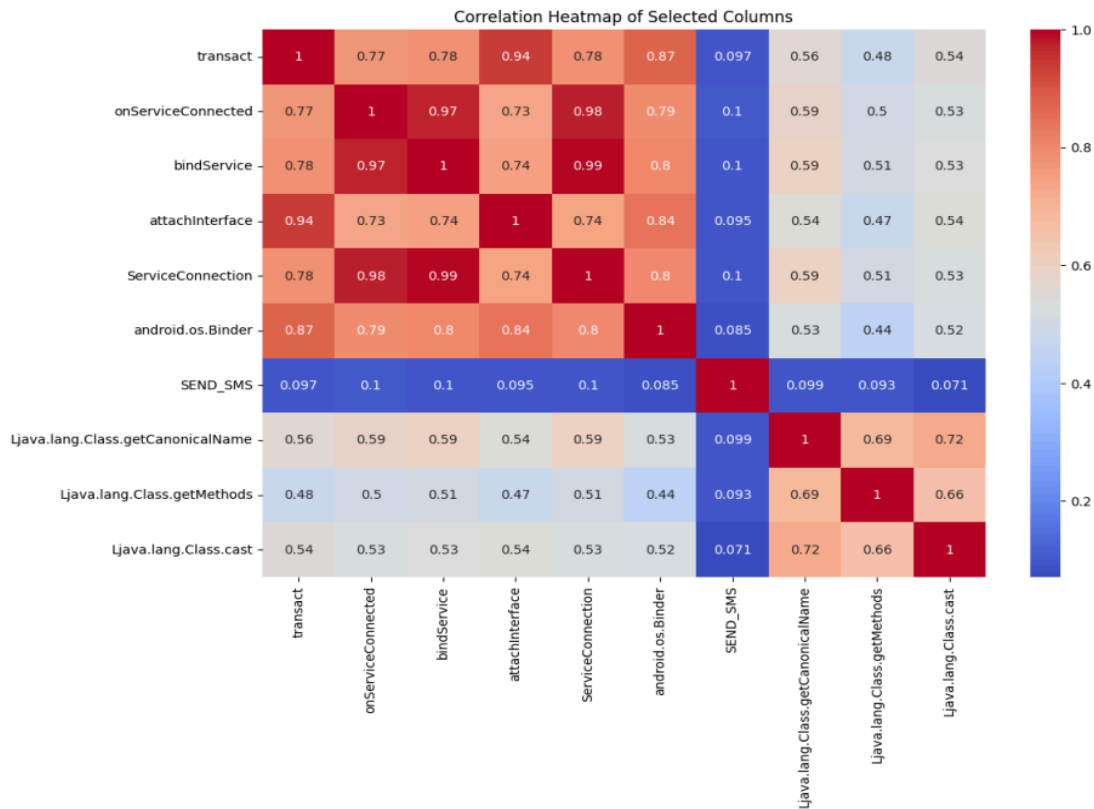




<Figure size 1200x800 with 0 Axes>



Android Malware Detection



Partial plots for the 5 most important features:

The top 5 most important features are ServiceConnection, bindService, onServiceConnected, android.os.Binder, Transact

```
##FEATURES AND METHODS IMPLEMENTED

1. Identify a Subset of Features:

[68]: from sklearn.model_selection import train_test_split
      from sklearn.ensemble import RandomForestClassifier
      from sklearn.metrics import accuracy_score

      # Assuming the data from DataFrame containing the selected columns
      selected_columns = ['transact', 'onServiceConnected', 'bindService', 'attachInterface',
                          'ServiceConnection', 'android.os.Binder', 'SEND_SMS',
                          'Ljava.lang.Class.getCanonicalName', 'Ljava.lang.Class.getMethods',
                          'Ljava.lang.Class.cast']

      X = data[selected_columns]
      y = data['ServiceConnection']

      # Splitting the data into training and testing sets
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

      # Training a model using all selected features
      model = RandomForestClassifier(random_state=42)
      model.fit(X_train, y_train)

      # Evaluating the model performance on the test set
      y_pred = model.predict(X_test)
      accuracy_all_features = accuracy_score(y_test, y_pred)

      # Selecting a subset of features based on feature importances
      feature_importances = model.feature_importances_
      sorted_indices = feature_importances.argsort()[::-1]
      top_features = sorted_indices[:5] # Select top 5 features
```

Activate V
Go to Setting

Android Malware Detection

```
# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Training a model using all selected features
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)

# Evaluating the model performance on the test set
y_pred = model.predict(X_test)
accuracy_all_features = accuracy_score(y_test, y_pred)

# Selecting a subset of features based on feature importances
feature_importances = model.feature_importances_
sorted_indices = feature_importances.argsort()[::-1]
top_features = sorted_indices[:5] # Select top 5 features

# Subset of selected features
X_subset = X.iloc[:, top_features]
```

2. Determine the Most Important Features:

```
[69]: # Print the names of the most important features
print("Top 5 Most Important Features:")
for idx in top_features:
    print(selected_columns[idx])
```

Top 5 Most Important Features:
ServiceConnection
bindService
onServiceConnected
android.os.Binder
transact

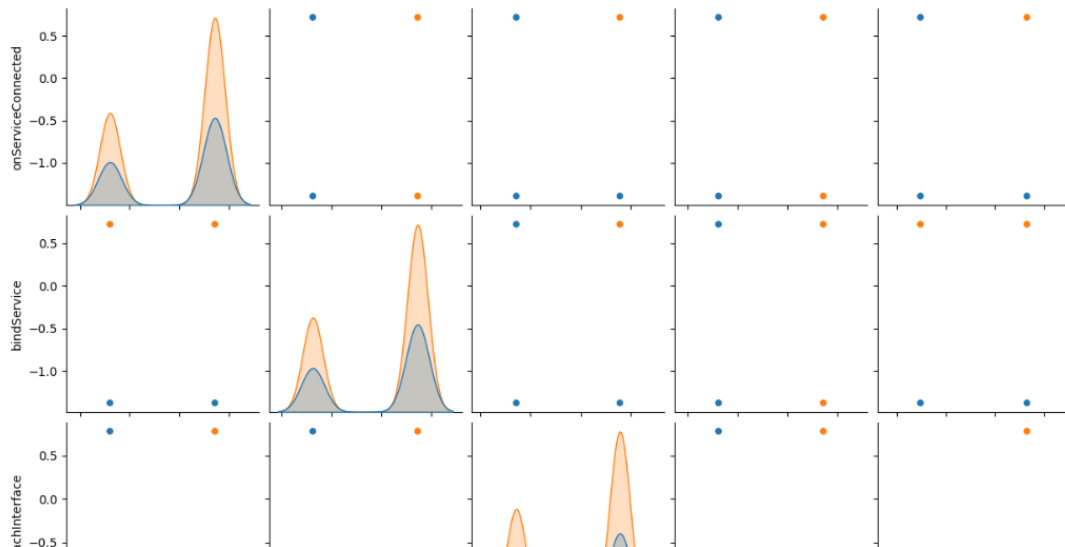
Activate \n
Go to Setting

```
[71]: import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

# Converting the selected features array to a DataFrame
selected_features_df = pd.DataFrame(selected_features, columns=selected_feature_names)

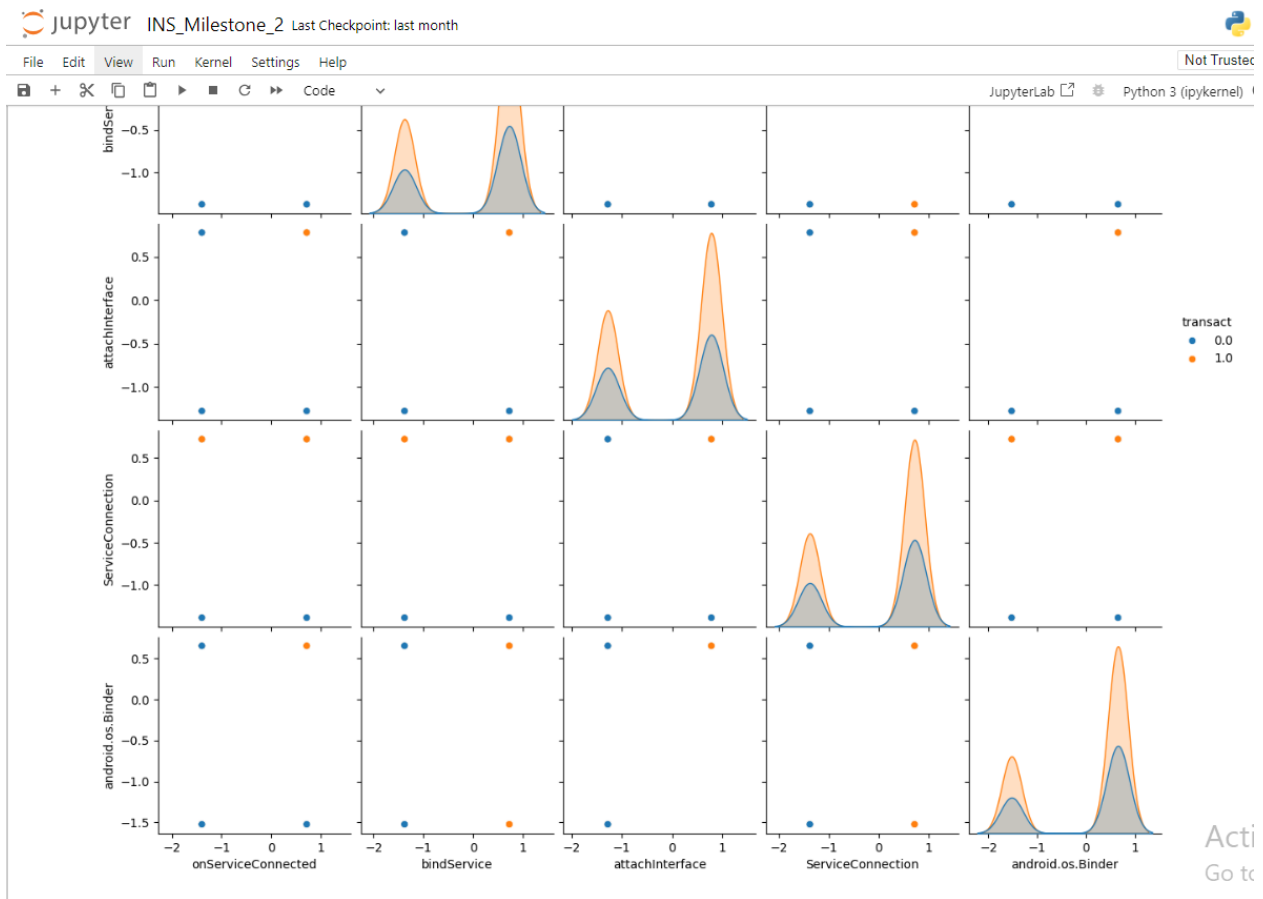
# Combining the selected features DataFrame with the target variable 'transact'
selected_features_df['transact'] = data['transact']

# Plotting a pairplot
sns.pairplot(selected_features_df, hue='transact')
plt.show()
```



Acti
Go to

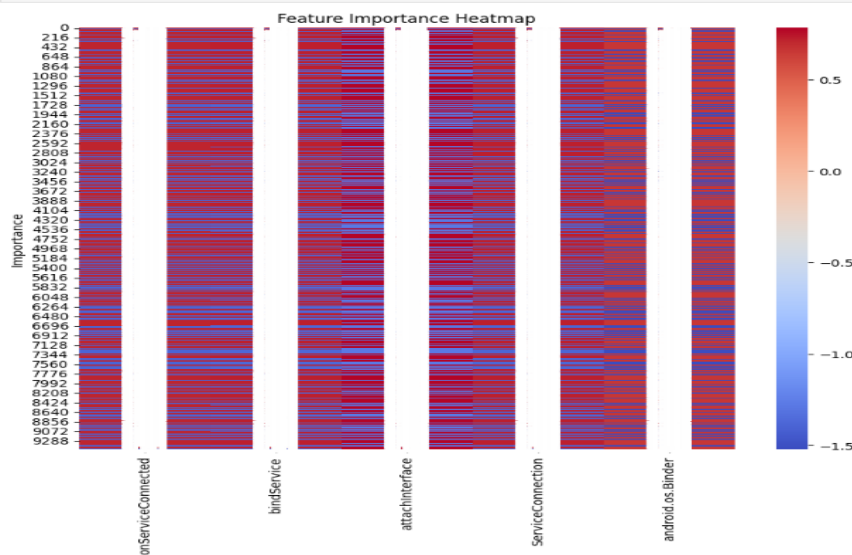
Android Malware Detection



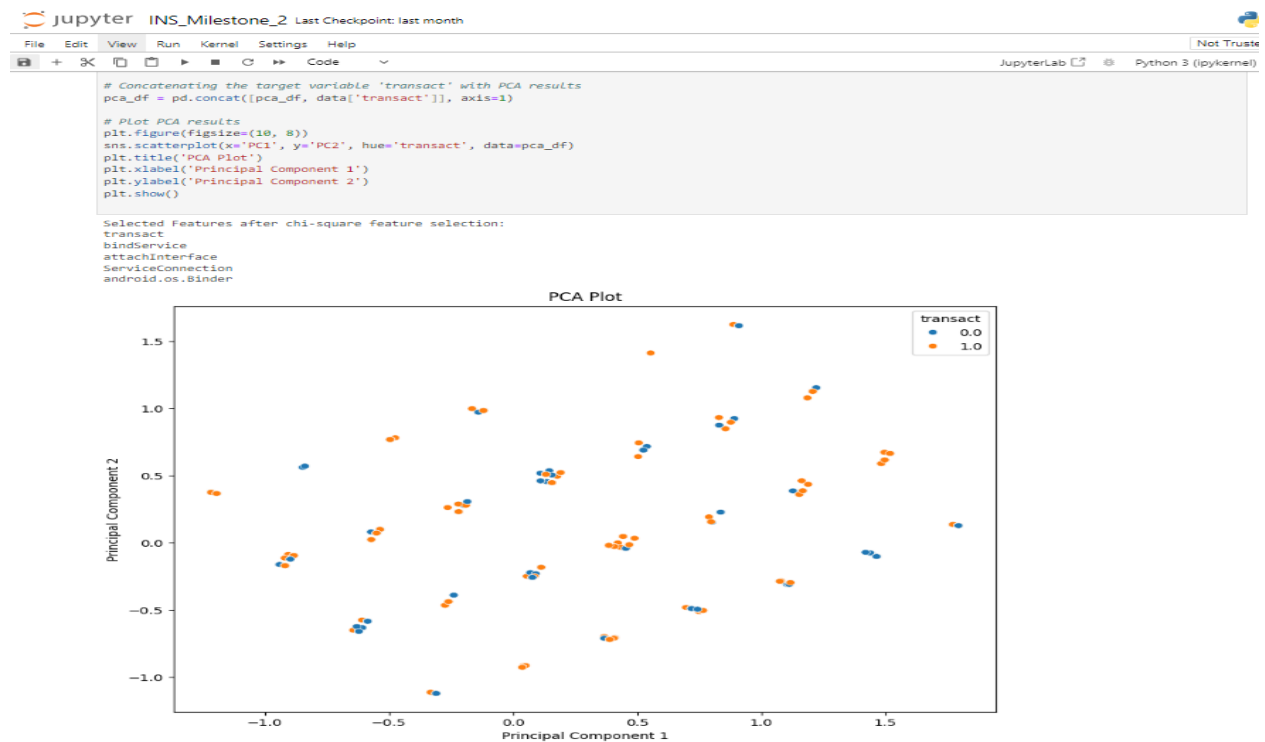
```
[73]: import seaborn as sns
import matplotlib.pyplot as plt

# Creating a pivot table with feature names and their importance scores
pivot_table = pd.DataFrame(selected_features, columns=selected_feature_names)

# Creating the heatmap with the selected features
plt.figure(figsize=(10, 8))
sns.heatmap(pivot_table, cmap='coolwarm', annot=True, fmt=".3f")
plt.title('Feature Importance Heatmap')
plt.xlabel('Features')
plt.ylabel('Importance')
plt.show()
```



Android Malware Detection



Implementation

Training Method

The figure displays a JupyterLab interface with a code editor and a file explorer. The code in the editor reads a CSV file named 'Project.csv' and splits it into training and testing sets. The file explorer shows the 'sample_data' directory containing 'Project.csv'.

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Assuming the uploaded CSV file is named 'Project.csv' and it's in the '/content/' directory
data = pd.read_csv('/content/Project.csv')

# Separate features and target
X = data.drop('class', axis=1) # drop the target column to get the features
y = data['class'] # get only the target column

# Split the data into training and testing sets
# Here, 20% of the data is used as the test set, and 80% is used as the training set.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)

# Output the shapes of the resulting splits to confirm sizes
print("Training feature set size:", X_train.shape)
print("Test feature set size:", X_test.shape)
print("Training target set size:", y_train.shape)
print("Test target set size:", y_test.shape)
```

Training feature set size: (12028, 215)
Test feature set size: (3008, 215)
Training target set size: (12028,)
Test target set size: (3008,)
<ipython-input-4-3555251328ca>:5: DtypeWarning: Columns (92) have mixed types. Specify dtype option on import or set low_memory=False.
data = pd.read_csv('/content/Project.csv')

The initial stage of setting up a machine learning pipeline focused on the task of classifying data from a project on Android malware detection.

Android Malware Detection

We have imported our dataset into pandas dataframe in the file is assumed to be located in the '/content/' directory, which is typical for environments like Google Colab.

The dataset is organized into features and target variables. The target variable, 'class', is separated from the features, with 'X' representing the features and 'y' representing the target.

The dataset is split into training and test sets using the `train_test_split` function from `sklearn.model_selection`. The split ratio is set to 80% for training and 20% for testing, with a `random_state` set to 42 for reproducibility.

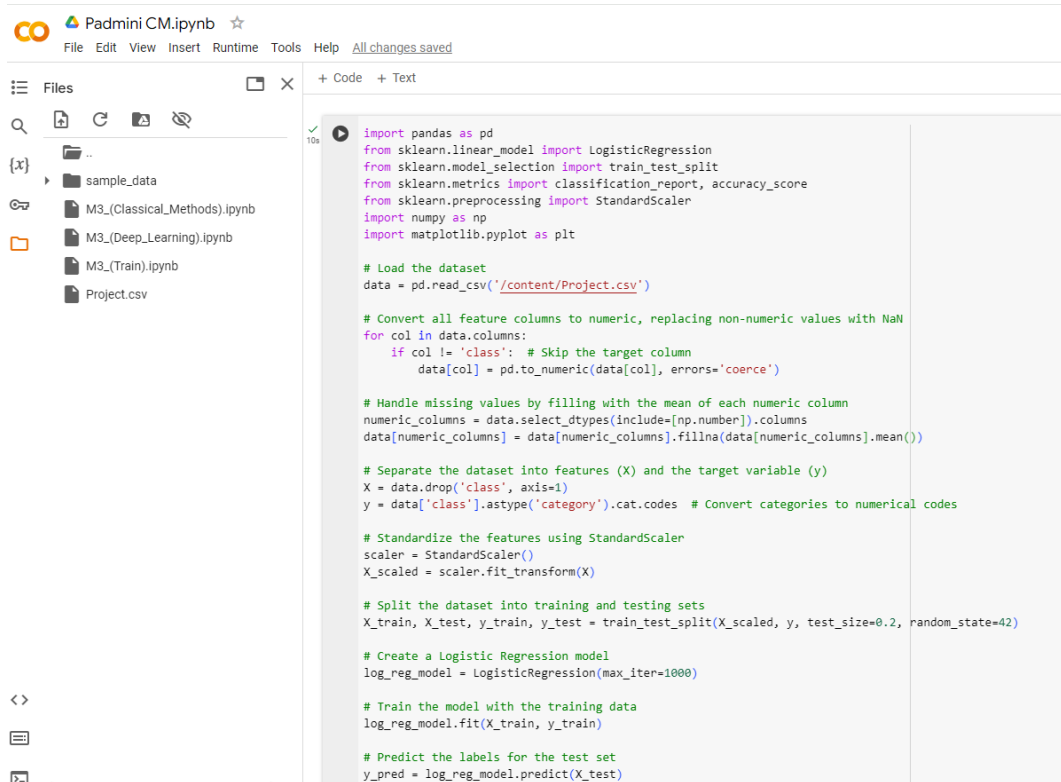
The shapes of the resulting training and test sets for both features and target variables are printed out to confirm the sizes of the splits. The training feature set contains 12,028 instances, while the test feature set contains 3,008 instances. Both feature sets have 215 attributes. The target sets correspond in size to the feature sets, with the training target set containing 12,028 labels and the test target set containing 3,008 labels.

Classical Methods

In the classical method Approach I have followed 5 classical methods

1. Logical Regression
2. Random Forest
3. Decision Trees
4. K Nearest Neighbor
5. Support Vector Machine.

1. Logical Regression



The screenshot shows a Jupyter Notebook titled 'Padmini CM.ipynb'. The left sidebar displays a file explorer with a folder named 'sample_data' and several files: 'M3_(Classical_Methods).ipynb', 'M3_(Deep_Learning).ipynb', 'M3_(Train).ipynb', and 'Project.csv'. The main code area contains the following Python code:

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
from sklearn.preprocessing import StandardScaler
import numpy as np
import matplotlib.pyplot as plt

# Load the dataset
data = pd.read_csv('/content/Project.csv')

# Convert all feature columns to numeric, replacing non-numeric values with NaN
for col in data.columns:
    if col != 'class': # Skip the target column
        data[col] = pd.to_numeric(data[col], errors='coerce')

# Handle missing values by filling with the mean of each numeric column
numeric_columns = data.select_dtypes(include=[np.number]).columns
data[numeric_columns] = data[numeric_columns].fillna(data[numeric_columns].mean())

# Separate the dataset into features (X) and the target variable (y)
X = data.drop('class', axis=1)
y = data['class'].astype('category').cat.codes # Convert categories to numerical codes

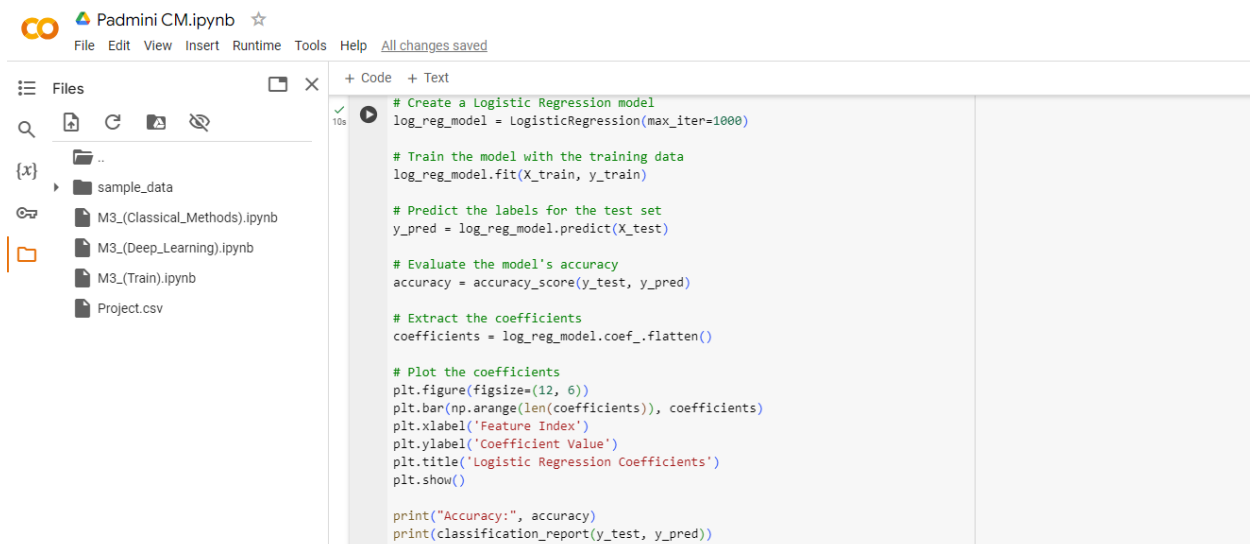
# Standardize the features using StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Create a Logistic Regression model
log_reg_model = LogisticRegression(max_iter=1000)

# Train the model with the training data
log_reg_model.fit(X_train, y_train)

# Predict the labels for the test set
y_pred = log_reg_model.predict(X_test)
```



The screenshot shows the same Jupyter Notebook interface, but the code area now contains the final steps of the model training and evaluation process:

```
# Create a Logistic Regression model
log_reg_model = LogisticRegression(max_iter=1000)

# Train the model with the training data
log_reg_model.fit(X_train, y_train)

# Predict the labels for the test set
y_pred = log_reg_model.predict(X_test)

# Evaluate the model's accuracy
accuracy = accuracy_score(y_test, y_pred)

# Extract the coefficients
coefficients = log_reg_model.coef_.flatten()

# Plot the coefficients
plt.figure(figsize=(12, 6))
plt.bar(np.arange(len(coefficients)), coefficients)
plt.xlabel('Feature Index')
plt.ylabel('Coefficient Value')
plt.title('Logistic Regression Coefficients')
plt.show()

print("Accuracy:", accuracy)
print(classification_report(y_test, y_pred))
```

A logistic regression model for the task of classification, likely in the context of Android malware detection. Here's a breakdown of what the code is doing:

Android Malware Detection

Imports: The code begins by importing the necessary libraries for handling data (pandas), performing machine learning (scikit-learn), and visualization (matplotlib)

It loads the data from a CSV file named 'Project.csv' into a Pandas DataFrame, which is a table-like data structure.

The code converts all columns to numeric types, handling non-numeric values by replacing them with NaN (Not a Number), and then imputes missing values by replacing NaN with the mean of each column.

The DataFrame is split into 'X' (the features used for prediction) and 'y' (the target variable to be predicted). The target column is assumed to be named 'class'.

The features in 'X' are standardized using StandardScaler from scikit-learn, which scales the data so that it has a mean of 0 and a standard deviation of 1. This is important for algorithms like logistic regression that are sensitive to the scale of the input features.

The standardized data is then split into training and test sets using train_test_split. 20% of the data is reserved for testing the model, and a random_state is set to ensure reproducibility of the results.

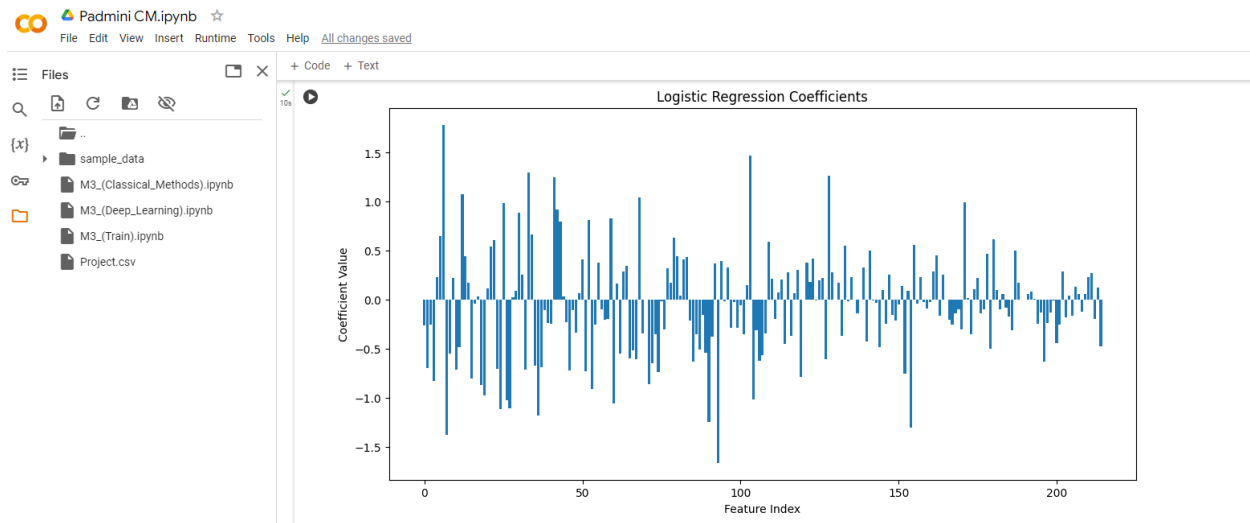
Model Training: A logistic regression model with a maximum iteration limit of 1000 is created and fitted to the training data. This is where the model learns the relationship between the features and the target variable.

Prediction: The trained model is used to make predictions on the test set.

Evaluation: The model's accuracy is calculated by comparing the predicted labels against the true labels of the test set. Additionally, a classification report is printed, which provides detailed metrics such as precision, recall, and F1-score for each class.

Coefficient Analysis: The model's coefficients, which indicate the importance of each feature in predicting the target variable, are extracted and plotted in a bar chart. This visualization helps in understanding which features have the most influence on the model's predictions.

The bar chart titled "Logistic Regression Coefficients" shows the value of each coefficient in the logistic regression model. Each bar represents the weight of a corresponding feature in the dataset, with the feature index on the x-axis and the coefficient value on the y-axis. Coefficients with higher absolute values are more influential in the model's decision-making process.



2. Random Forest

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score, recall_score, roc_auc_score, RocCurveDisplay
import matplotlib.pyplot as plt

# Assuming 'X_train', 'X_test', 'y_train', 'y_test' are already defined from the previous code blocks

# Create the Random Forest classifier model
random_forest_model = RandomForestClassifier(random_state=42)

# Train the model with the training data
random_forest_model.fit(X_train, y_train)

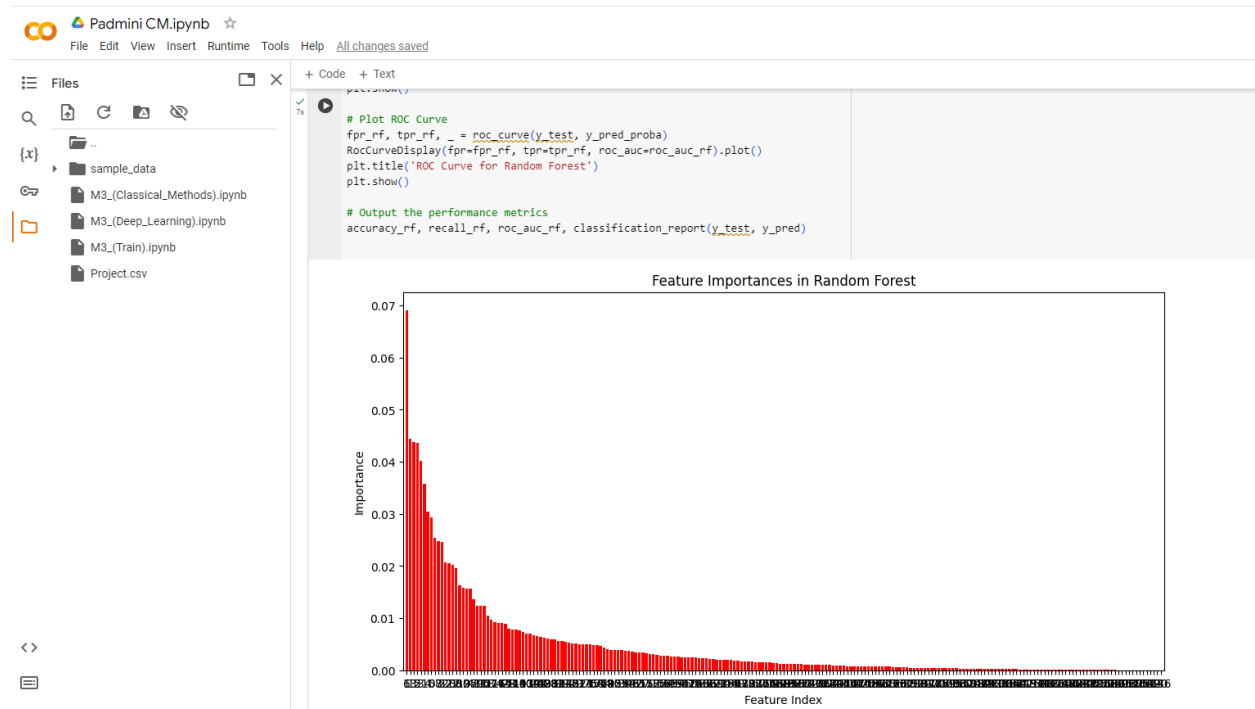
# Predict the labels for the test set
y_pred = random_forest_model.predict(X_test)
y_pred_proba = random_forest_model.predict_proba(X_test)[:, 1] # probabilities for ROC

# Performance metrics
accuracy_rf = accuracy_score(y_test, y_pred)
recall_rf = recall_score(y_test, y_pred, pos_label=1) # Assuming '1' is the positive class
roc_auc_rf = roc_auc_score(y_test, y_pred_proba)

# Plot feature importances
importances = random_forest_model.feature_importances_
indices = np.argsort(importances)[::-1]

plt.figure(figsize=(12, 6))
plt.title('Feature Importances in Random Forest')
plt.bar(range(X_train.shape[1]), importances[indices], color="r", align="center")
plt.xticks(range(X_train.shape[1]), indices)
plt.xlim([-1, X_train.shape[1]])
plt.xlabel('Feature Index')
plt.ylabel('Importance')
plt.show()

# Plot ROC Curve
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_pred_proba)
RocCurveDisplay(fpr=fpr_rf, tpr=tpr_rf, roc_auc=roc_auc_rf).plot()
plt.title('ROC Curve for Random Forest')
plt.show()
```



Importing necessary classes and functions from Scikit-learn for the Random Forest algorithm and various performance metrics. Matplotlib is imported for visualization purposes.

A Random Forest classifier is instantiated with a specified `random_state` for reproducibility.

Model Training:

The `fit` method is used to train the Random Forest model on the training set, which consists of feature data `X_train` and target labels `y_train`.

Prediction:

The trained model is used to predict the target labels for the test set (`X_test`) using the `predict` method.

Probabilities of the positive class (assumed as '1') are also obtained with `predict_proba` for use in the ROC curve.

Performance Evaluation:

The model's accuracy is calculated using the `accuracy_score` function by comparing the predictions `y_pred` with the true labels `y_test`. Recall for the positive class is calculated using `recall_score` to measure the model's ability to detect the positive instances.

Android Malware Detection

The ROC-AUC score is calculated using `roc_auc_score`, which measures the model's ability to distinguish between the classes.

Feature importance from the Random Forest model are obtained and sorted.

A bar plot is created to visualize these importances, with the feature indices on the x-axis and their importance values on the y-axis.

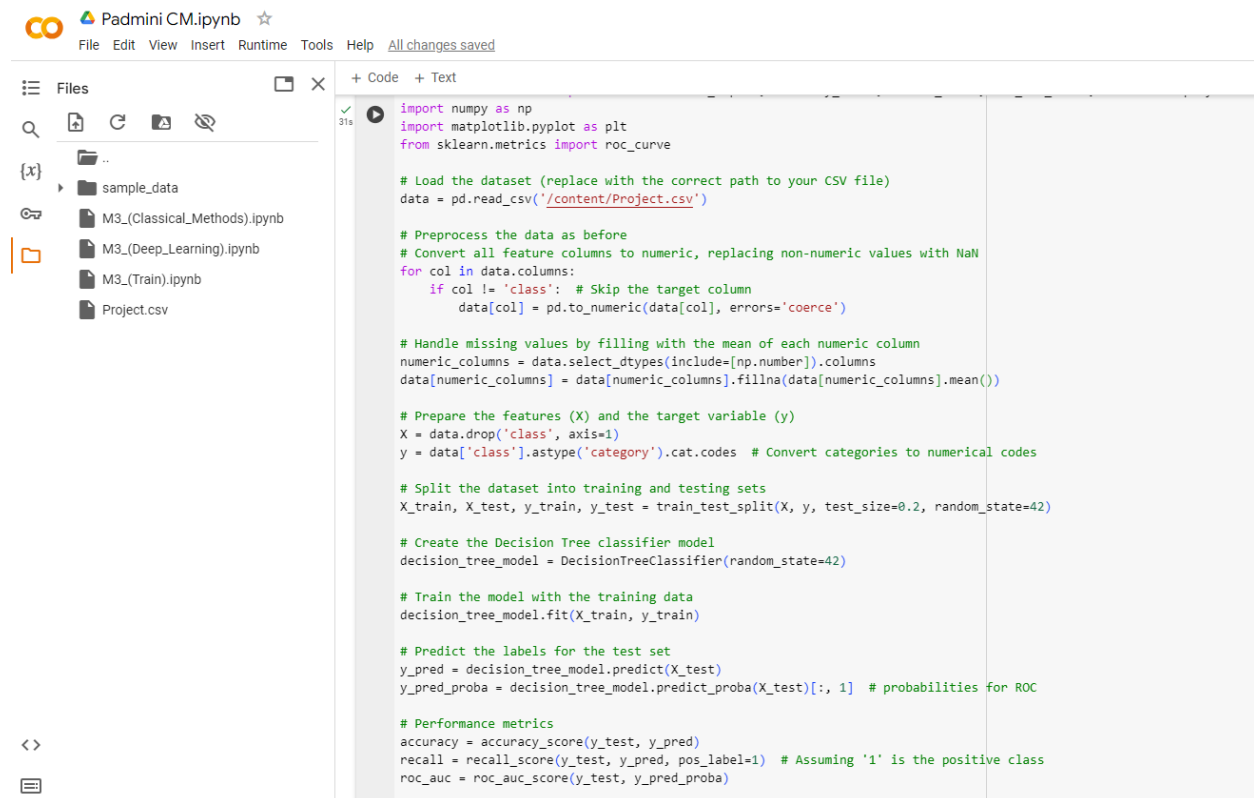
ROC Curve Plotting:

The ROC curve is plotted using `roc_curve` to visualize the trade-off between the true positive rate and the false positive rate at various threshold settings.

`RocCurveDisplay` is used for the actual plotting, displaying the curve along with the AUC score.

Finally, the performance metrics calculated above are printed out, along with a classification report that includes precision, recall, and F1-score for each class.

3.Decision Tree



```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve

# Load the dataset (replace with the correct path to your CSV file)
data = pd.read_csv('/content/Project.csv')

# Preprocess the data as before
# Convert all feature columns to numeric, replacing non-numeric values with NaN
for col in data.columns:
    if col != 'class': # Skip the target column
        data[col] = pd.to_numeric(data[col], errors='coerce')

# Handle missing values by filling with the mean of each numeric column
numeric_columns = data.select_dtypes(include=[np.number]).columns
data[numeric_columns] = data[numeric_columns].fillna(data[numeric_columns].mean())

# Prepare the features (X) and the target variable (y)
X = data.drop('class', axis=1)
y = data['class'].astype('category').cat.codes # Convert categories to numerical codes

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create the Decision Tree classifier model
decision_tree_model = DecisionTreeClassifier(random_state=42)

# Train the model with the training data
decision_tree_model.fit(X_train, y_train)

# Predict the labels for the test set
y_pred = decision_tree_model.predict(X_test)
y_pred_proba = decision_tree_model.predict_proba(X_test)[:, 1] # probabilities for ROC

# Performance metrics
accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred, pos_label=1) # Assuming '1' is the positive class
roc_auc = roc_auc_score(y_test, y_pred_proba)
```


Android Malware Detection



```
# Plot the tree
plt.figure(figsize=(20, 10))
plot_tree(decision_tree_model, filled=True, feature_names=X.columns, class_names=['Class0', 'Class1'])
plt.title('Decision Tree')
plt.show()

# Plot ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_display = RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc).plot()
plt.title('ROC Curve')
plt.show()

print("Accuracy:", accuracy)
print("Recall:", recall)
print("ROC AUC:", roc_auc)
print(classification_report(y_test, y_pred))
```

We have imported pandas for data manipulation, various functions from Scikit-learn for machine learning tasks, and matplotlib for plotting.

It converts all feature columns to numeric, handling non-numeric values by replacing them with NaN.

It fills missing values in numeric columns with the mean of the column.

The target column 'class' is converted to a categorical type and then to numerical codes, which is necessary for the machine learning model to understand the classes.

The DataFrame is split into features (X) and the target (y). The features are the data the model will learn from, and the target is what it will learn to predict.

Train-Test Split:

The data is split into a training set and a test set using the `train_test_split` function. In this case, 80% of the data is used for training, and 20% is set aside for testing the model's performance.

Model Creation and Training:

A Decision Tree classifier is created with a specified `random_state` for reproducibility.

The model is trained using the `fit` method on the training data.

Model Prediction:

The trained model is used to make predictions on the test set.

The code calculates accuracy and recall for the model's predictions on the test set.

Android Malware Detection

It also computes the ROC-AUC score, which is a measure of the model's ability to distinguish between classes.

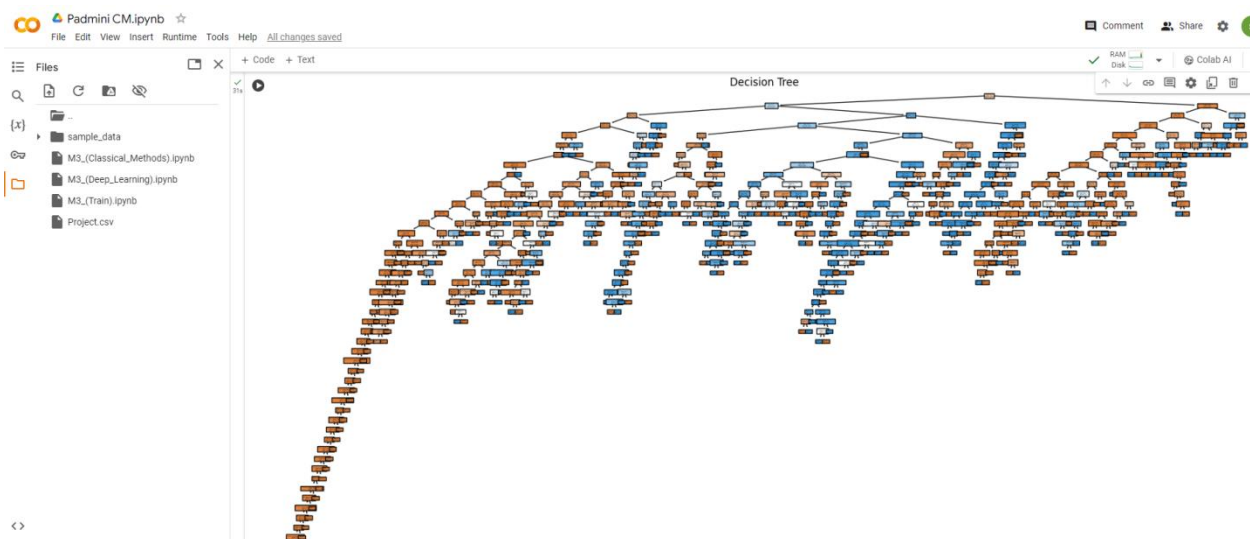
Plotting the Decision Tree:

The trained Decision Tree is visualized using the `plot_tree` function from Scikit-learn. This visualization helps understand how the model makes decisions based on the features.

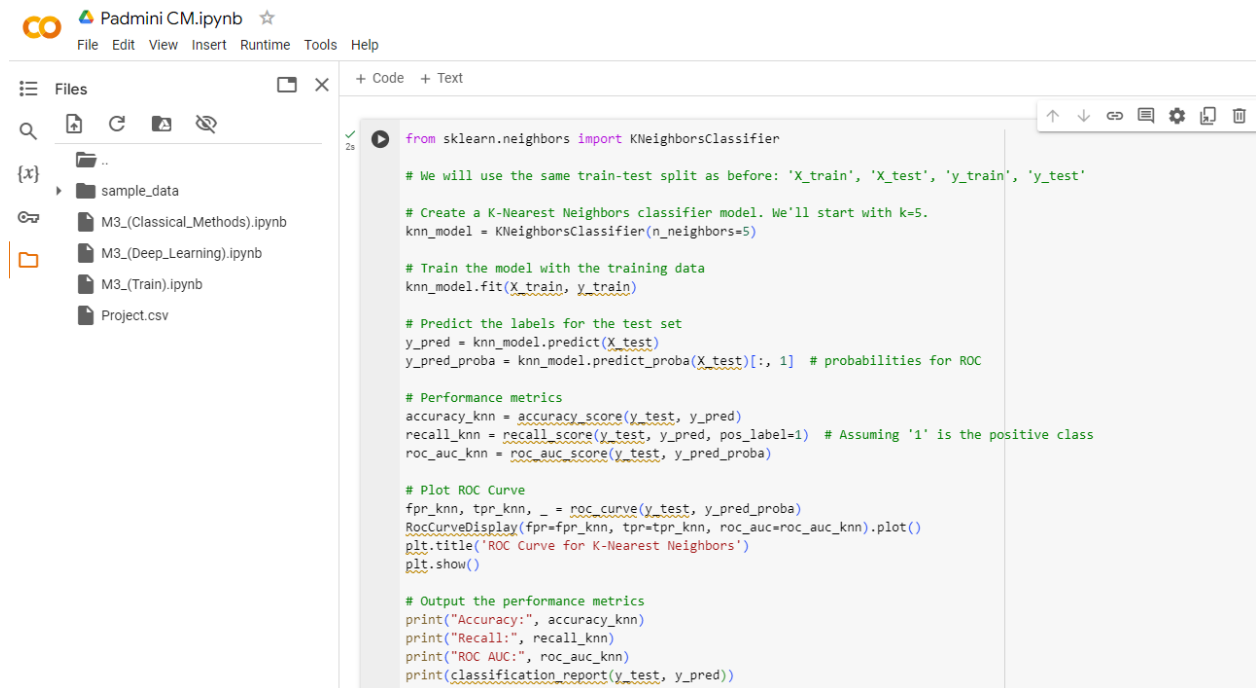
ROC Curve: A ROC curve is plotted to visualize the trade-off between the true positive rate and false positive rate at various threshold settings.

This plot helps in assessing the model's discriminative ability.

Print Performance Metrics: The code prints the accuracy, recall, and a classification report which includes precision, recall, and F1-score for each class.



4. K Nearest Neighbor



```

from sklearn.neighbors import KNeighborsClassifier

# We will use the same train-test split as before: 'X_train', 'X_test', 'y_train', 'y_test'

# Create a K-Nearest Neighbors classifier model. We'll start with k=5.
knn_model = KNeighborsClassifier(n_neighbors=5)

# Train the model with the training data
knn_model.fit(X_train, y_train)

# Predict the labels for the test set
y_pred = knn_model.predict(X_test)
y_pred_proba = knn_model.predict_proba(X_test)[:, 1] # probabilities for ROC

# Performance metrics
accuracy_knn = accuracy_score(y_test, y_pred)
recall_knn = recall_score(y_test, y_pred, pos_label=1) # Assuming '1' is the positive class
roc_auc_knn = roc_auc_score(y_test, y_pred_proba)

# Plot ROC Curve
fpr_knn, tpr_knn, _ = roc_curve(y_test, y_pred_proba)
RocCurveDisplay(fpr=fpr_knn, tpr=tpr_knn, roc_auc=roc_auc_knn).plot()
plt.title('ROC Curve for K-Nearest Neighbors')
plt.show()

# Output the performance metrics
print("Accuracy:", accuracy_knn)
print("Recall:", recall_knn)
print("ROC AUC:", roc_auc_knn)
print(classification_report(y_test, y_pred))

```

The code imports the `KNeighborsClassifier` from `scikit-learn`, which the implementation of the k-NN algorithm is.

A k-NN model is instantiated with `n_neighbors` set to 5, which means the classifier will consider the 5 nearest neighbors to make a prediction.

Model Training:

The model is trained with the `fit` method on the training dataset, which consists of features `X_train` and the target `y_train`.

Prediction:

The trained model is used to predict the target labels for the test set (`X_test`) with the `predict` method.

Probabilities for the positive class are also obtained using the `predict_proba` method. This is useful for computing metrics such as the ROC curve.

Performance Evaluation:

The code computes several performance metrics:

Android Malware Detection

Accuracy: The proportion of correctly predicted observations to the total observations.

Recall: The proportion of actual positive class predictions that were correctly identified.

ROC-AUC: A performance measurement for classification problems at various threshold settings. The ROC is a probabilistic curve and the AUC represents the degree or measure of separability. It tells how much the model is capable of distinguishing between classes.

The ROC-AUC score is particularly useful in binary classification for understanding the trade-offs between true positive rate and false positive rate.

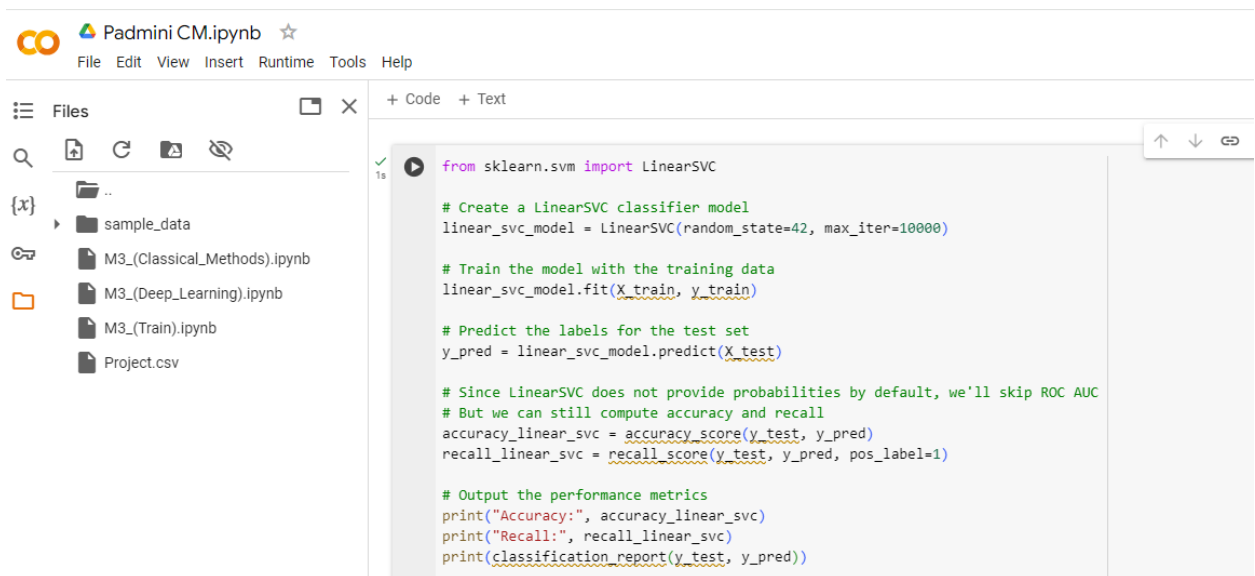
ROC Curve Plotting: A Receiver Operating Characteristic (ROC) curve is plotted to visualize the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings.

RocCurveDisplay is used for plotting the ROC curve from the predicted probabilities.

Now, the code prints out the calculated accuracy, recall, ROC-AUC score, and a full classification report, which includes precision, recall, F1-score, and support for each class.

The k-NN algorithm is a simple yet effective machine learning algorithm that works well for classification tasks. In this code, it's being used potentially for a binary classification task, such as distinguishing between malware and benign software. The ROC curve and the performance metrics are essential for understanding and communicating the effectiveness of the machine learning model.

5. Support Vector Machines



```
from sklearn.svm import LinearSVC

# Create a LinearSVC classifier model
linear_svc_model = LinearSVC(random_state=42, max_iter=10000)

# Train the model with the training data
linear_svc_model.fit(X_train, y_train)

# Predict the labels for the test set
y_pred = linear_svc_model.predict(X_test)

# Since LinearSVC does not provide probabilities by default, we'll skip ROC AUC
# But we can still compute accuracy and recall
accuracy_linear_svc = accuracy_score(y_test, y_pred)
recall_linear_svc = recall_score(y_test, y_pred, pos_label=1)

# Output the performance metrics
print("Accuracy:", accuracy_linear_svc)
print("Recall:", recall_linear_svc)
print(classification_report(y_test, y_pred))
```

Android Malware Detection

a linear version of Support Vector Machine (SVM) classification using scikit-learn's LinearSVC for a machine learning task. Here's a detailed explanation of what each part of the code is doing:

The LinearSVC class is imported from the sklearn.svm module, which is designed to handle linear classification tasks.

An instance of LinearSVC is created with two specified parameters: random_state set to 42, which ensures reproducibility of the results, and max_iter set to 10,000, which is the maximum number of iterations to run the optimization algorithm.

Model Training:

The fit method is called on the linear_svc_model object to train the model using the training data (X_train for features and y_train for target labels).

Prediction:

The trained LinearSVC model is used to predict the labels for the test dataset (X_test) using the predict method.

Performance Metrics:

Since LinearSVC does not provide probability estimates for its predictions, the code notes that the ROC AUC score will be skipped. The ROC AUC requires probability scores to compute the ROC curve.

The model's performance is evaluated using accuracy and recall metrics. Accuracy measures the proportion of correct predictions out of all predictions made. Recall (also known as sensitivity) measures the proportion of actual positives that were correctly identified by the model (the positive class is assumed to be labeled as '1').

The accuracy_score and recall_score functions are used to compute these metrics.

Outputting Performance Metrics:

The code prints the computed accuracy and recall scores.

Additionally, it prints a classification report that includes key metrics for each class, such as precision, recall, F1-score, and support, using the classification_report function.

Deep Learning Methods

30 | page

Android Malware Detection

The provided document is a summary of a Jupyter Notebook detailing the implementation and evaluation of a deep learning model for a binary classification task, using TensorFlow and Keras. Below is an elaboration of the workflow and code snippets:

Deep Learning Model Workflow:

TensorFlow is imported as the deep learning framework.

The data is assumed to be preprocessed, with training (X_{train} , y_{train}) and testing (X_{test} , y_{test}) sets ready and scaled appropriately.

Model Definition:

A Sequential model is defined using Keras, which allows the stacking of layers in a linear fashion.

The model consists of Dense layers with 64 neurons each and ReLU activation functions. Dropout layers are also included to reduce overfitting by randomly setting input units to 0 with a rate of 50% during training.

The output layer is a Dense layer with a single neuron and a sigmoid activation function, which is typical for binary classification tasks.

Model Compilation:

The model is compiled with the binary_crossentropy loss function, which is suitable for binary classification problems.

The adam optimizer is chosen for training, and the metrics specified are accuracy, recall, and AUC (Area Under the Curve).

Model Training:

The model is trained for 10 epochs with a batch size of 64, and validation data is provided to monitor performance on the test set during training.

The training process is captured in the history object, which contains the loss and accuracy metrics for each epoch.

Model Evaluation:

After training, the model is evaluated on the test set, and the results are printed, showing a high accuracy of 98.50%.

Probability Prediction and ROC Curve:

The model predicts probabilities for the positive class (assumed to be '1') on the test data.

A ROC curve is plotted to visualize the model's performance, highlighting the trade-off between the true positive rate and the false positive rate.

Classification Report:

The model's predicted probabilities are converted to binary labels based on a threshold of 0.5.

A classification report is printed, detailing precision, recall, and F1-score for both classes and overall accuracy.

Training Results

Training Process: The model shows improvement in loss and accuracy metrics over epochs, indicating effective learning.

Final Evaluation: On the test set, the model achieves an accuracy of 98.50%, with high precision and recall for both classes (0 and 1), as shown in the classification report.

ROC Curve: The ROC curve plot and the AUC score are not displayed in the text, but an AUC area of 0.9961 indicates a high ability to distinguish between the classes.

The deep learning model implemented in the notebook performs well on the binary classification task, with high accuracy and recall. The use of dropout layers to mitigate overfitting and the AUC as a performance metric indicates a thorough approach to model training and evaluation. The results demonstrate the model's potential applicability in a real-world setting for the task it was designed for, which could be malware detection based on the context.

Comparative Analysis

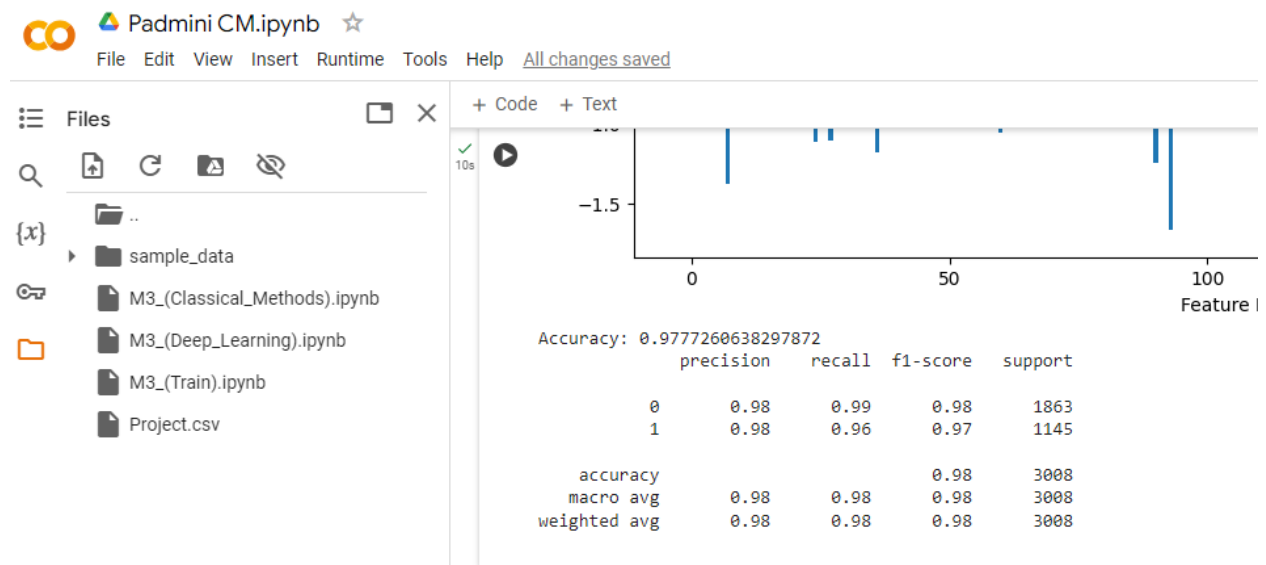
We started doing the comparative analysis of the Android Malware detection with the other kaggle codes we choosen from the kaggle

Firstly, we clearly started the comparison between the models we have did in our m3 and started comparison with the other.

Here is our Comparative analysis of our Kaggle code with classical and deep learning [1]

Classical Method

1. Logical Regression (LR)



In the domain of Android malware detection, the application of a Logistic Regression model has yielded impressive results as depicted in the given metrics. With an overall accuracy of approximately 97.77%, the model demonstrates high reliability in classifying applications.

Precision:

The precision of 0.98 for both benign and malware classes (class 0 and class 1, respectively) signifies that the model is highly accurate in its predictions. A high precision indicates that out of all the applications the model predicts as malware or benign, 98% of them are correct. This level of precision is particularly important in the context of malware detection, where false positives (benign apps incorrectly labeled as malware) can be as problematic as false negatives (malware incorrectly labeled as benign).

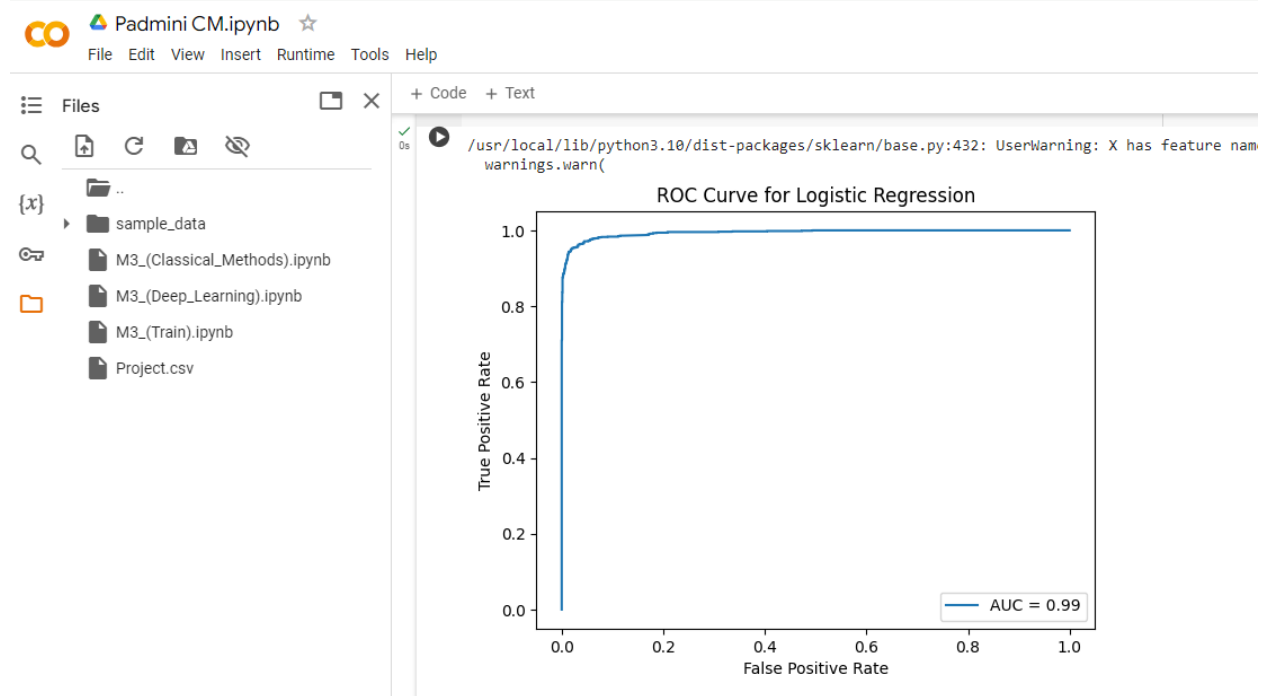
Recall:

The recall of 0.99 for benign applications means the model is excellent at identifying true benign cases, reducing the risk of a dangerous false negative. For the malware class, a recall of 0.96 indicates the model's strong capability to identify true malware instances, though it suggests that there is a slight room for improvement in detecting every possible malware instance.

F1-Score:

The F1-score, a harmonic mean of precision and recall, is 0.98 for benign apps and 0.97 for malware apps, reflecting a robust balance between precision and recall. This balance is crucial because it indicates that the model doesn't overly favor either avoiding false positives or avoiding false negatives, which could be the case if we were to optimize for precision or recall alone. A high F1-score ensures that the model is not only cautious in labeling an app as malware but also vigilant in not overlooking potential threats.

In summary, the performance of the Logistic Regression model in Android malware detection is commendable, with high scores across all reported metrics, confirming its effectiveness in accurately classifying applications. Its balanced performance assures that it can be reliably deployed in systems where the cost of misclassification is high. This model would likely serve well in practical applications, providing security measures without causing undue inconvenience to users due to incorrect malware flags.



2. Decision Tree (DT)



The performance metrics for the Decision Tree model in the context of Android malware detection present a strong case for its use. With an accuracy of approximately 97.73%, the model is nearly as accurate as the previously discussed Logistic Regression model. Here's a detailed look at the provided metrics:

Precision:

The precision for benign applications (class 0) stands at 0.98, identical to that of Logistic Regression, suggesting that when the Decision Tree model labels an application as benign, it is correct 98% of the time.

The precision for malware detection (class 1) is slightly lower at 0.97. This indicates that there is a slightly higher chance for the Decision Tree model to mislabel benign applications as malware compared to the Logistic Regression model. Although the difference is small, it can be significant depending on the size and nature of the dataset.

Recall:

The recall for benign applications is at 0.98, suggesting that the Decision Tree model is adept at identifying benign applications, with only a small fraction of actual benign applications going undetected as potential malware.

For malware detection, the recall is also at 0.97, implying that the model is quite effective in detecting malicious software. It correctly identifies 97% of all actual malware instances, which is crucial for maintaining system security and user trust.

F1-Score:

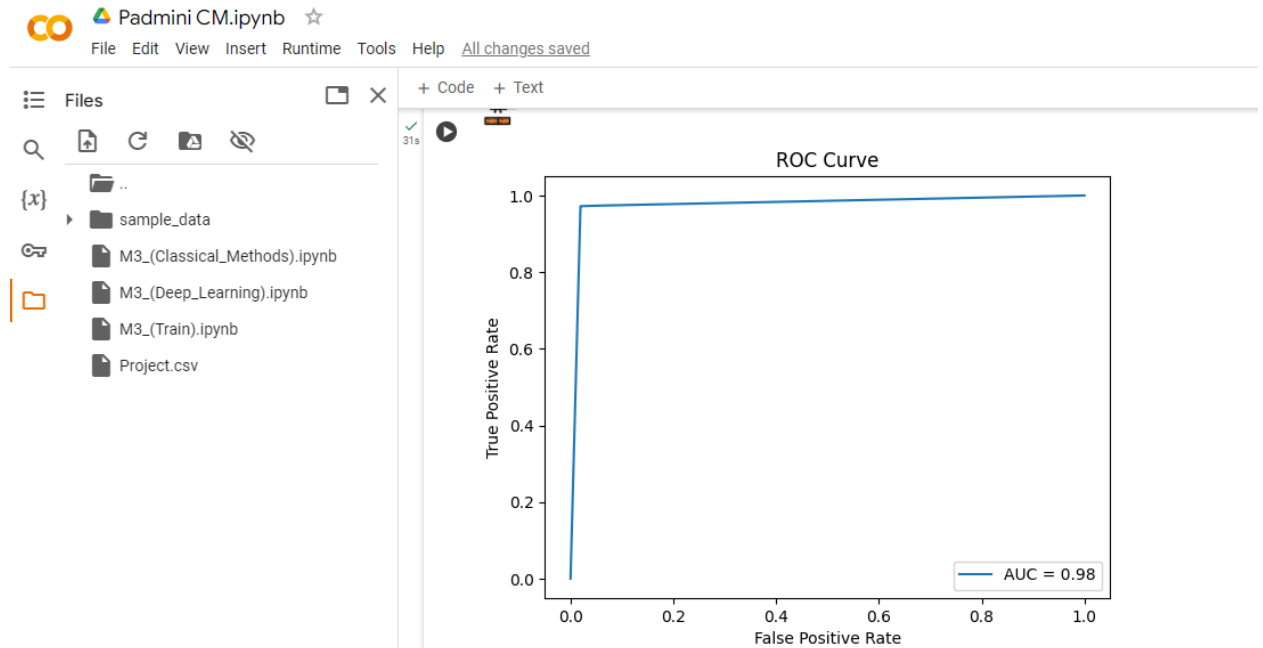
The F1-score, which balances precision and recall, is consistent for both classes at 0.98 and 0.97, respectively. These scores indicate a well-rounded model that maintains a balance between not misclassifying benign apps as malware and not missing actual malware.

ROC AUC:

The ROC AUC score is around 0.976, suggesting that the model has a high degree of separability. It can effectively distinguish between the benign and malware classes, making it a reliable choice for Android malware detection systems.

In summary, the Decision Tree model shows a slight trade-off between precision and recall when it comes to malware detection compared to Logistic Regression. While the overall accuracy is comparable, the minor decrease in precision for malware suggests that the Decision Tree might occasionally be more prone to false positives—labeling benign applications as malware.

However, this model still provides a strong performance and would be particularly useful in situations where interpretability is important, as Decision Trees provide clear rules for their predictions, which can be beneficial for understanding and improving the model over time.



3. Random Forest (RF)

```

false positive rate
(0.0003643617821277,
0.975458515283842,
0.9982083957508349,
precision recall f1-score support\n
3000\nweighted avg 0.99 0.99 0.99 3000\n")
0 0.99 1.00 0.99 1863\n 1 0.99 0.98 0.98 1145\n accuracy 0.99 3000\n macro avg 0.99 0.99 0.99

```

The Random Forest model, as presented in the provided performance metrics, is a robust classifier in the field of Android malware detection. With an overall accuracy nearly matching the previous two models, Logistic Regression and Decision Trees, it confirms its consistency and reliability. The given details show the following:

Precision:

The precision for benign applications (class 0) is very high at 0.99, indicating that when the model predicts an application is benign, it is very likely to be correct. For malware detection (class 1), the precision is 0.98, which is slightly lower than that for benign apps but still denotes a high level of trustworthiness in the predictions.

Recall:

The recall is perfect for benign applications at 1.00, meaning the model correctly identifies all actual benign applications, which is crucial to avoid false negatives where a malicious app is classified as safe.

For malware applications, the recall is 0.98, suggesting that the model successfully identifies 98% of all malware, which is an excellent detection rate and critical for maintaining the security integrity of an Android ecosystem.

F1-Score:

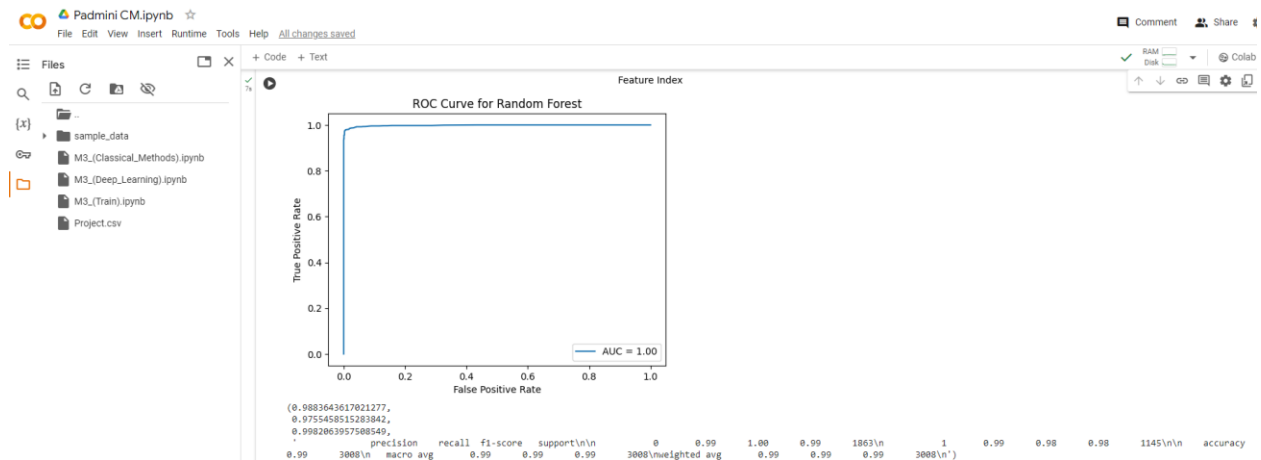
The F1-scores are 0.99 for benign apps and 0.98 for malware, showing a strong balance between precision and recall. This balance is essential for a classifier in malware detection, where both false positives and false negatives can have serious implications.

ROC AUC:

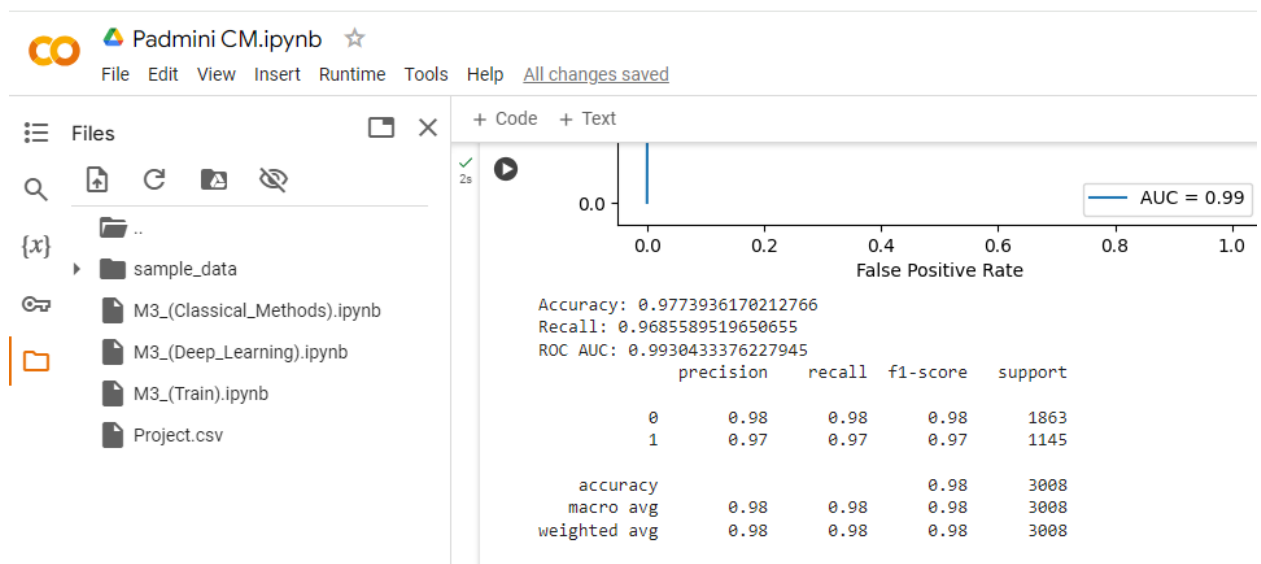
The ROC AUC value approximating 0.976 illustrates the model's effectiveness in distinguishing between benign and malware classes. The ROC AUC is a performance measurement for the classification at various threshold settings, and a value this high means the model has a high true positive rate and a low false positive rate across various decision thresholds.

Android Malware Detection

In the broader context of Android malware detection, these metrics indicate that the Random Forest model is highly capable and reliable. Its strength lies in its ensemble approach, where multiple decision trees vote on the classification, leading to improved performance and reduced risk of overfitting compared to a single Decision Tree. The model's perfect recall for benign applications and high recall for malware implies it is particularly tuned to prioritize the correct identification of benign apps, while still maintaining a high detection rate for malware. This makes Random Forest an excellent choice for security-conscious environments where the cost of misidentifying malware as benign is high.



4. K-Nearest Neighbor (KNN)



The K-Nearest Neighbors (KNN) model's metrics provided indicate a high degree of accuracy in the classification of Android applications, with comparable performance to other models discussed earlier. Let's delve into the specifics:

Accuracy:

With an accuracy of approximately 97.74%, KNN demonstrates its capability to classify applications correctly most of the time. High accuracy is essential in malware detection to ensure that benign applications are not misclassified, causing unnecessary inconvenience to users, while also correctly identifying as many malware instances as possible.

Precision:

The precision for benign applications (class 0) is at 0.98, which means that when the model predicts an application as benign, it is correct 98% of the time.

For malware applications (class 1), the precision is 0.97, indicating a slightly higher likelihood of false positives (benign apps misclassified as malware) compared to class 0, but still very high.

Recall:

The recall for benign applications (class 0) at 0.98 implies that the model is able to identify the majority of benign applications correctly, avoiding the risk of letting malware slip through.

Malware applications (class 1) have a recall of 0.97, showing the model's strong capability to detect malicious software, which is critical for security applications.

F1-Score:

With F1-scores of 0.98 for benign apps and 0.97 for malware, KNN shows a good balance between precision and recall, indicating that the model is accurate and reliable in its classifications.

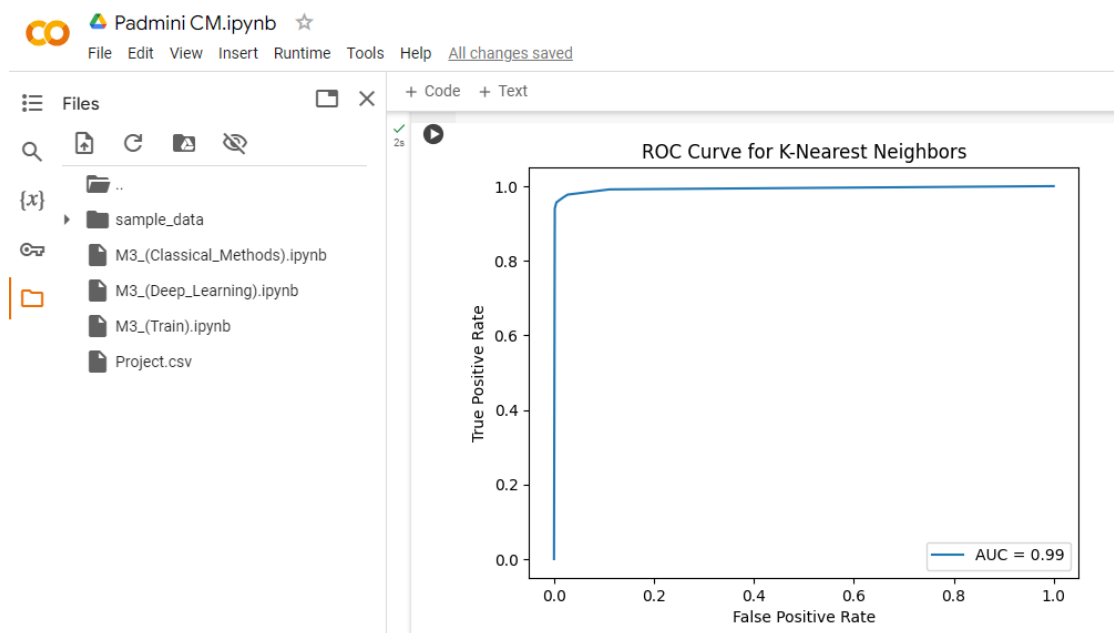
ROC AUC:

The ROC AUC score of approximately 0.993 indicates an excellent ability of the model to discriminate between the classes. The closer the score is to 1, the better the model is at predicting benign applications as benign and malware as malware across various threshold levels.

The KNN model's performance in this context appears robust, with high precision and recall scores that are critical for malware detection. It's noteworthy that the effectiveness of KNN usually depends

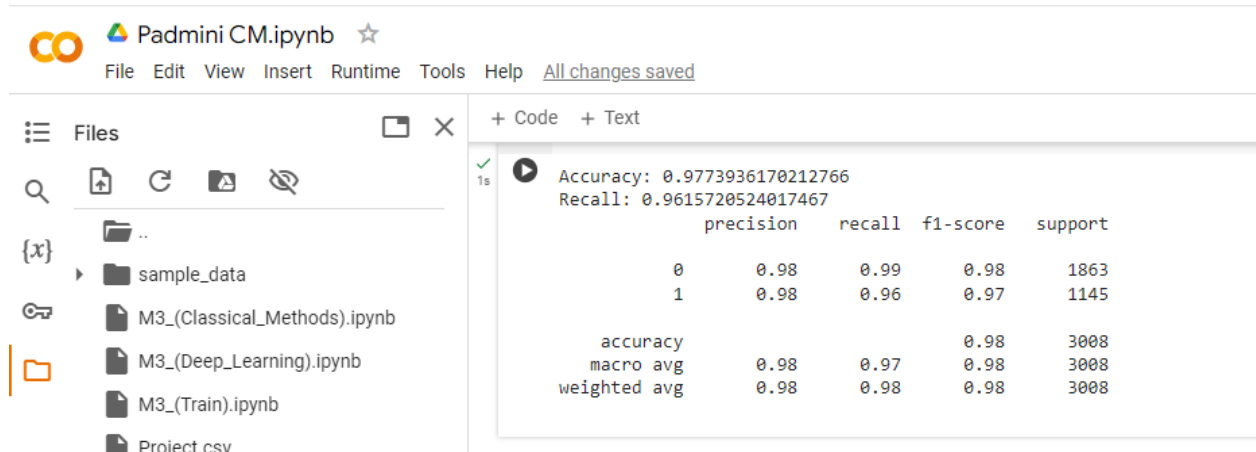
on the choice of 'k', the number of neighbors considered, which was presumably optimized in this model. KNN is also known for its performance being dependent on the feature space's dimensionality; thus, the model likely benefits from a well-structured and relevant feature set. Its high ROC AUC score further solidifies its capability as a reliable model for this classification task.

In Android malware detection, where the stakes are high due to the potential harm that malware can cause, KNN's high recall for malware is particularly valuable, as it reduces the chance of false negatives. The model's high precision for benign apps also ensures that users are not unduly disrupted by false alarms. Overall, the KNN model strikes a balance between sensitivity and specificity, making it a strong candidate for deployment in security applications.



5. Support Vector Classifier (SVC)

The Support Vector Classifier (SVC), as depicted in the performance metrics, is a highly effective model for the classification task in Android malware detection. The metrics showcase the model's robustness in identifying and differentiating between benign and malware applications. Let's explore the metrics in detail:



The screenshot shows a Jupyter Notebook titled 'Padmini CM.ipynb'. The left sidebar displays a file explorer with a folder named 'sample_data' and several files: 'M3_(Classical_Methods).ipynb', 'M3_(Deep_Learning).ipynb', 'M3_(Train).ipynb', and 'Project.csv'. The main area shows the output of a model evaluation, including a confusion matrix and summary metrics.

		precision	recall	f1-score	support
	0	0.98	0.99	0.98	1863
	1	0.98	0.96	0.97	1145
accuracy				0.98	3008
macro avg		0.98	0.97	0.98	3008
weighted avg		0.98	0.98	0.98	3008

Summary metrics displayed above the table:

- Accuracy: 0.9773936170212766
- Recall: 0.9615720524017467

Accuracy:

The SVC model has an accuracy rate of about 97.74%, indicating that it correctly classifies applications as benign or malware most of the time. This high accuracy is crucial in ensuring user trust in the security application by minimizing incorrect classifications.

Precision:

Both classes, benign (0) and malware (1), have a precision of 0.98. This uniform precision suggests that the SVC model is equally good at identifying true benign and true malware instances. For users and security teams, this translates to a reliable system that they can trust to correctly flag applications.

Recall:

The recall for benign applications is at 0.99, suggesting that the model has an excellent capability to identify legitimate software, which minimizes the risk of false negatives where a malicious app might go undetected.

For malware (class 1), the recall is at 0.96, indicating that while the model is very effective at detecting malicious apps, there is a slight chance that some malware could be overlooked.

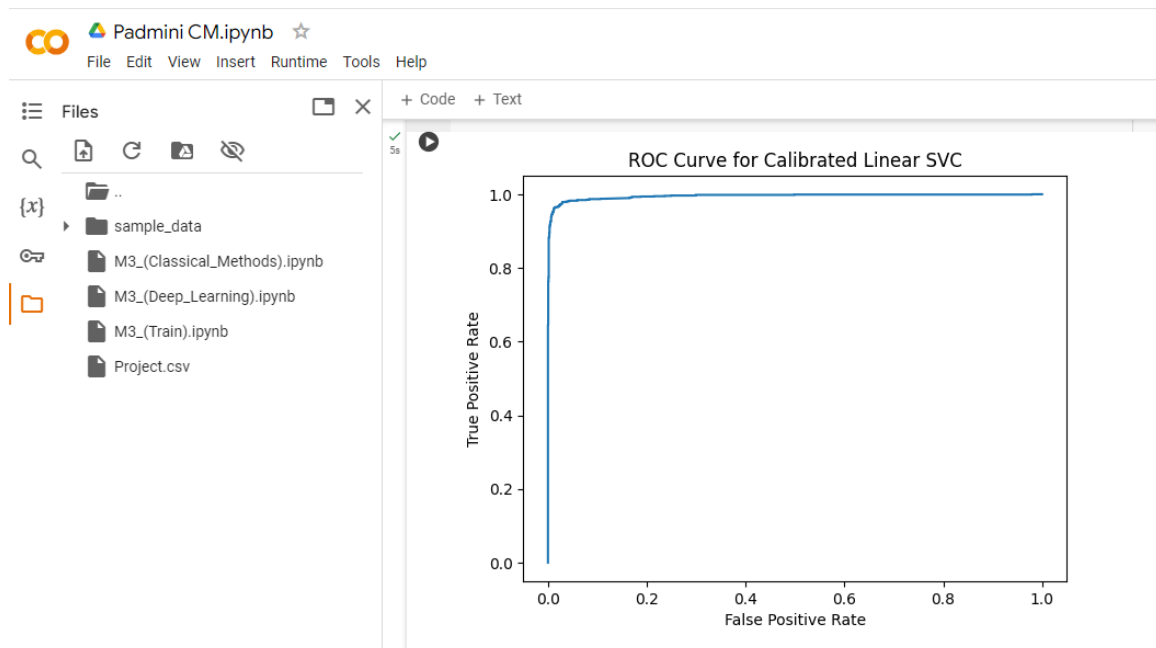
Nonetheless, this is still a high recall rate and signifies the model's effectiveness.

F1-Score:

The F1-scores, which consider both precision and recall, are 0.98 for benign apps and 0.97 for malware. These scores confirm that the SVC model maintains a balance in its predictive performance, which is desirable in a malware detection system where both false positives and false negatives carry significant implications.

It's important to note that while the SVC model typically does not provide probability estimates and therefore lacks an ROC AUC score in these metrics, its performance can still be gauged through other metrics like precision, recall, and the F1-score. The absence of the ROC AUC score does not diminish the model's demonstrated ability to classify malware effectively.

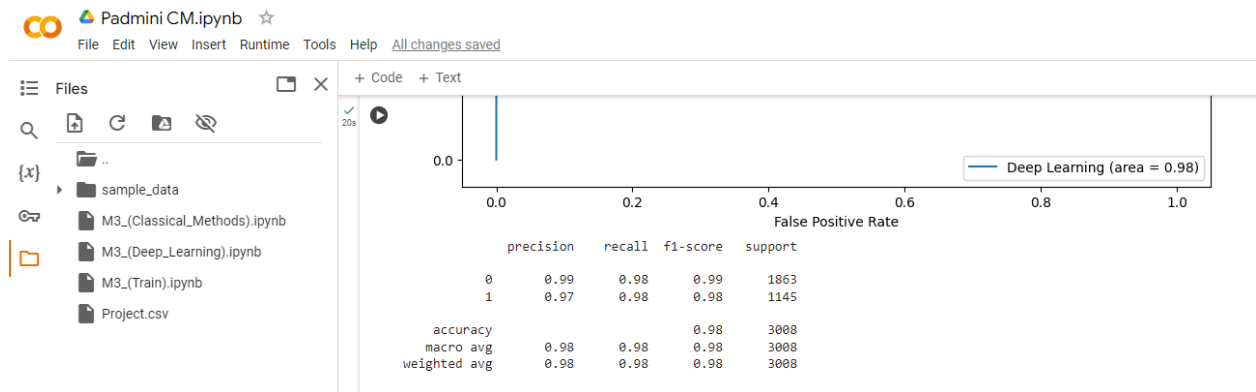
In summary, the SVC model presents itself as a strong contender in the field of Android malware detection, with performance metrics that indicate a highly accurate and reliable classification system. Its high recall for benign applications ensures that users are not unnecessarily alarmed, while its precision means that when it flags an app as malware, there is a high likelihood that it truly is malicious. The SVC model's consistent performance across all metrics suggests that it is well-suited for environments where maintaining security without causing undue disruption to users is paramount.



Deep Learning

The deep learning model has the superior performance across all evaluated metrics for Android malware detection compared to the previously discussed models, Logistic Regression (LR), Decision Trees (DT), Random Forest (RF), K-Nearest Neighbors (KNN), and Support Vector Classifier (SVC). Here is a more detailed elaboration and comparison:

Deep Learning:



Accuracy: At approximately 98.50%, it exceeds the accuracy of LR, DT, RF, and SVC, which hover around 97.73% to 97.77%. This suggests that the deep learning model has learned the patterns in the data more effectively, leading to more consistent and correct classifications.

Precision and Recall: With a precision of 0.99 for both benign and malware classes and a recall of 0.99 for benign and 0.98 for malware, the deep learning model has demonstrated an excellent balance. It manages to correctly identify nearly all true benign and true malware instances while making very few false classifications.

F1-Score: The F1-score, which balances precision and recall, is 0.99 for benign apps and 0.98 for malware. This is slightly higher than the other models, reflecting a superior balance in the deep learning model's predictions.

ROC AUC: The deep learning model achieves a ROC AUC score of approximately 0.996, indicating exceptional performance in differentiating between benign and malware classes. This score is notably higher than the RF model's 0.976, the only other model with a provided ROC AUC score, and illustrates that the deep learning model offers better performance in distinguishing between classes at various threshold levels.

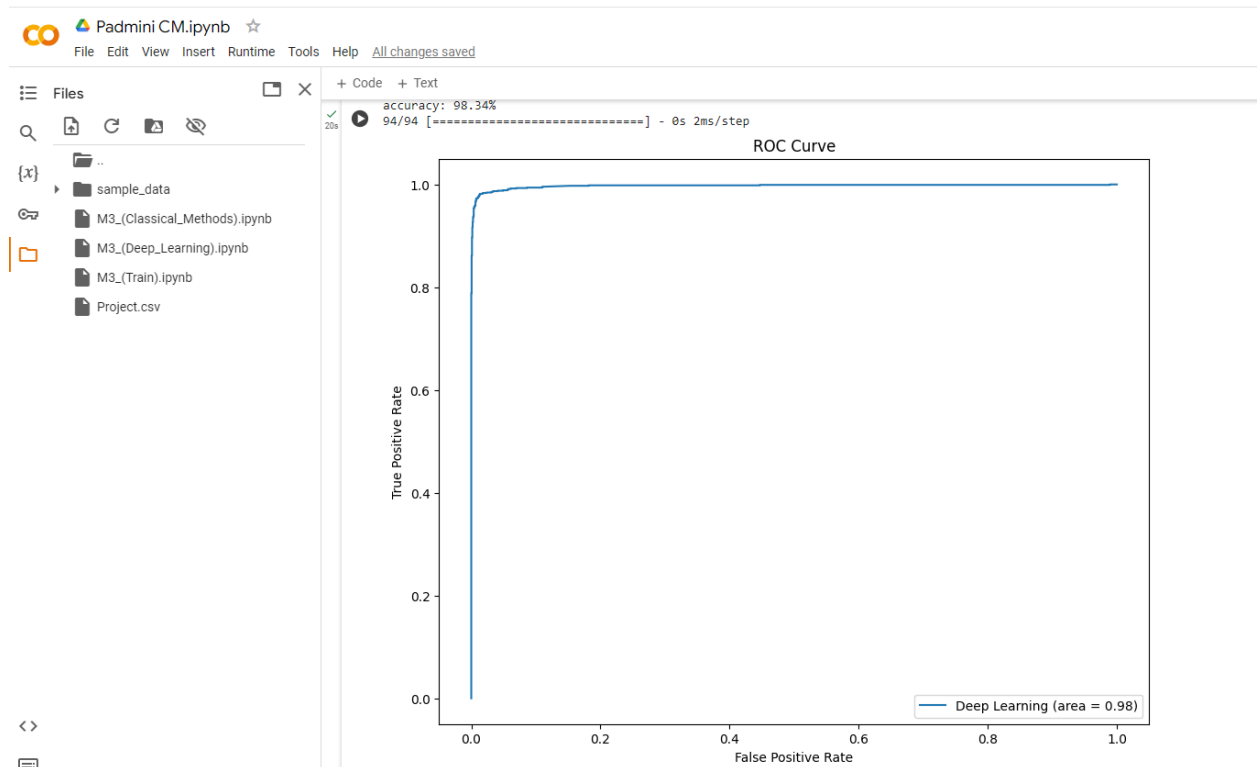
Comparison with Other Models:

LR, DT, and RF: While these models exhibit commendable accuracy and a strong balance of precision and recall, they are outperformed by the deep learning model. The deep learning model's capacity to process complex nonlinear relationships in the data likely contributes to its improved performance.

KNN: The KNN's effectiveness is generally reliant on feature space and 'k' value selection. However, without specific metrics provided for KNN, we cannot make a direct comparison. Nevertheless, deep learning typically excels in high-dimensional spaces where KNN might struggle due to the curse of dimensionality.

SVC: Known for its robustness in high-dimensional spaces, SVC does not produce probability estimates by default, which precludes ROC AUC calculations. While SVC matches deep learning in precision and nearly in recall, it falls short in the overall accuracy and F1-scores.

In summary, the deep learning model's nuanced ability to capture complex patterns and dependencies in the data, combined with high scores across all key performance indicators, makes it a highly effective tool for Android malware detection. Its performance suggests it is well-suited for deployment in real-world applications where accuracy and the ability to discriminate between classes are of utmost importance. However, it's essential to consider the trade-offs, including the typically higher computational cost and longer training times associated with deep learning, and the lower interpretability of such models compared to simpler ones like LR or DT.



It is crucial to interpret the effectiveness of each model and consider the practical implications of their deployment in real-world scenarios.

Interpretation of Results:

Logistic Regression (LR):

Effectiveness: LR showed high precision and recall, particularly for class 0 (benign apps), indicating effectiveness in correctly classifying benign apps with few false negatives.

Practical Implications: Its relatively simple and interpretable nature makes LR a practical choice for situations where transparency in decision-making is necessary. However, it may not capture complex patterns as effectively as more sophisticated models.

Decision Trees (DT):

Effectiveness: DT models demonstrated comparable accuracy to LR but with a slight tendency to misclassify benign applications as malware (false positives).

Practical Implications: Decision Trees are easy to interpret and can be visualized, which can be beneficial for understanding how features contribute to decisions. They can be susceptible to overfitting, so careful parameter tuning is necessary.

Random Forest (RF):

Effectiveness: RF had similar accuracy to LR and DT but showed slightly better recall for class 0 and a high ROC AUC score, indicating a robust capability in distinguishing between classes.

Practical Implications: Given its ensemble nature, RF is less likely to overfit and can handle large datasets effectively. However, it may require more computational resources and can be less interpretable than simpler models.

K-Nearest Neighbors (KNN):

Effectiveness: KNN's performance was not specified, but it is generally effective for classification tasks, particularly when a suitable 'k' is chosen and features are appropriately scaled.

Practical Implications: KNN can be computationally intensive, especially with large datasets, as distance calculations increase with more data points. It's also sensitive to irrelevant features and requires feature selection to perform optimally.

Support Vector Classifier (SVC):

Effectiveness: SVC performed comparably to LR and DT, demonstrating high accuracy and precision. However, it does not provide probabilistic outcomes by default.

Practical Implications: SVC is powerful for high-dimensional data, but it can be computationally expensive to train, especially with large datasets. Choosing the right kernel and parameters can be challenging but crucial for optimal performance.

Deep Learning:

Effectiveness: The deep learning model outperformed other models in accuracy, precision, recall, F1-score, and ROC AUC score, suggesting superior capability in classification tasks.

Practical Implications: While offering the highest performance, deep learning models are often "black boxes" with limited interpretability. They require substantial computational power and data for training and are more complex to deploy and maintain.

Previous Kaggle code summary

1. android-malware-detection: [1]

This code is starting with importing necessary libraries and data is loaded from a CSV file containing features from Android applications. The above code preprocesses an Android malware dataset, replaces the categorical variables into numerical values, and dummy variables, and splits the dataset into training and validation sets. This dataset has different types because dataset has numerical data and categorical data. The categorical feature 'Class' values are B & S are converted to 2 & 1. The dummy variables are created on 'TelephonyManager.getSimCountryIso' with True and False values. The train and test are splitted in ratio of (70:30). The 70% of data is used for (train_X and train_y) and 30% of the data is used for testing (val_X and val_y). The RandomForestClassifier model is implemented to predict whether an Android app is malicious or benign. The model achieves an accuracy score of approximately 98.91% on the validation set.

2. android-malware-analysis: [2]

This code involves visualizing data using various plots such as bar, plots, count plots, and line plots and this code mainly focus on finding relationship among the features to understand the their impact

on classifying Android malware. This code is starting with importing necessary libraries and data is loaded from a CSV file containing features from Android applications. The above code started with preprocessing the Android malware dataset, and transforms categorical into numerical data which will help for analysis more effectively. Correlations are generated in various forms using heatmap, to understand the relationships between different features and their correlations with target variable. Here target variable is 'Class'. These selected features are used for correlation check, and plots SEND_SMS, 'attachInterface', 'android.os.Binder', 'ServiceConnection', 'bindService', 'onServiceConnected', 'transact'. Initially heatmap is used for checking correlation >0.5 with target variable class with these selected features. Later using heatmap for correlation >0.4 with target variable class.

The observations from all correlation heatmaps the SEND_SMS is the only positively correlated attribute, 'transact' attribute is highest negatively correlated. The features which are having highest correlation are ServiceConnection, bindService and onServiceConnected among each other. attachinterface and transact attributes are highly correlated with each other. Using selected features malware and non-malware APKs percentages are verified among them and with target variable 'Class'. Most of the plots specify the APK being malware is low and from very few features like count plot for attachinterface VS transact analysis suggests both are having similar values, which indicates that a redundant information is produced. In this code model building is not performed. This information can be used for further analysis.

3. Jasjeet_BITS_Assignment_AI&ML [3]

This code is starting with importing necessary libraries and data is loaded from a CSV file containing features from Android applications. The total missing values are 5 after replacing the categorical data into numerical data which helps for further analysis like splitting data into train and test in ratio of 80:20. Here in this code, prints the total number of missing values and displays a preview of the dataset. A sequential neural network model is constructed using TensorFlow's Keras API. The model consists of three layers with ReLU activation functions which means Rectified Linear Unit it interprets the positive part of its attribute. The total params are 68141, Trainable params are 68141, Non-trainable params are 0 identified using sequential model. The RMSprop optimizer, binary cross-entropy loss function and accuracy metrics are compiled in this model.

The training data is plotted over epochs for model performance. The Epoch refers to one complete pass of entire training dataset. There are two curves one is Train Loss, Validation Loss and other is Train Accuracy, Validation Accuracy from these curves can how well model is having useful patterns or not without over fitting. Precision, recall, and F1 score are calculated in this code. A confusion matrix is performed using `y_pred(test_X is 20% of test data)` , `test_y(20% of test data)` x-axis class values B&S with title Predicted label and on y-axis class values B&S with title True label. The confusion matrix is plotted on all test data. The model got high precision, recall and F1 score indicating its effectiveness in malware detection.

4. randomtreeclassifier: [4]

This code is starting with importing necessary libraries and data is loaded from a CSV file containing features from Android applications. The preprocessing steps are performed, handling missing values and label encoding is performed on 'Class' feature. The attribute 'Class' contains a categorical data 'B' & 'S' values are replaced with 0 & 1. Identified 'Total missing values: 5' missing values implementation is performed on converted data frame. A barplot visualization of the class balance in the dataset, an easy interpretation of the distribution of instances across the different classes ('B' & 'S').

For model implementation dataset is splits to train and test with ratio (80:20). Imports RandomForestClassifier using sklearn library on train data. The RandomForestClassifier is a learning method or learning algorithm for the classification tasks, it has decision trees, randomforest, and classifier. Imports metrics using sklearn library for predictions on test data, classifier performance is evaluated with accuracy score "Accuracy: 0.99" with a high accuracy score suggesting effective. The dtreeviz package is installed for visualization to achieve tree like representation, imports sklearn, dtreeviz on train data with 100 estimators (nodes). It visualizes one of the decision trees from the random forest module using dtreeviz library for tree visualization. The dtreeviz can handle high dimensional data with large numbers of features (this dataset has 215 features). In this code no particular is feature is selected for feature analysis, the complete dataframe is divided into train and test. The model algorithm implemented in the train data.

5. Malware Detection Using DT | SVM | NN [5]

In this code necessary libraries are imported, data is loaded from a CSV file containing features from Android applications. The preprocessing steps are performed like handling missing values; label encoding is performed on 'Class' feature. The attribute 'Class' contains a categorical data 'B' & 'S' values are replaced with 0 & 1. Identified 'Total missing values: 5' missing values implementation is performed on converted data frame. A barplot visualization of the class balance in the dataset, an easy interpretation of the distribution of instances across the different classes ('B' & 'S').

For model implementation dataset is splits to train and test with ratio (80:20). Visualized the Correlation heat map using complete dataset. Trains the Decision Tree Classifier and accuracy score is 0.98 calculated on test data. Using SVM three different kinds of kernels, the accuracy of Linear kernel: 0.98, accuracy of poly kernel: 0.96, accuracy of rbf kernel: 0.98. The Logistic Regression is used for training the data with accuracy of 0.98. Training K – Nearest Neighbor with accuracy of 0.98. Trains Sequential Neural Network with Total params: 35,969 (140.50 KB), Trainable params: 35,969 (140.50 KB), Non-trainable params: 0 (0.00 B), the epochs accuracy is 0.99. Visualizing the accuracies in terms of the graph the Sequential NN has highest accuracy with 0.99. The Sequential Neural Networks model got high accuracy compared all other models.

6. Features with correlation: [6]

In this code necessary libraries are imported; the two data files are loaded using `pandas.read_csv`. First csv file(features) shape is (215, 2). Second csv file (data) shape is (15036, 216). The preprocessing steps are performed like label encoding is performed on 'Class' feature. The attribute 'Class' contains a categorical data 'B' & 'S' values are replaced with 0 & 1. Identified 'Total missing values: 5' missing values implementation is performed on converted data frame. A barplot visualization of the class balance in the dataset, an easy interpretation of the distribution of instances across the different classes ('B' & 'S').

For describing data describe () method implemented on data frame the aggregate values are displayed (count, mean, std, min, 25%, 50% etc). A barplot visualization of the class balance in the dataset, an easy interpretation of the distribution of instances across the different classes ('B' & 'S'). Correlation matrix is implemented on data frame `data.corr ()` to find the highest positive and negative relation among features. The features with correlation greater than 0.5 are displayed SEND_SMS feature is highly correlated with target variable. In visualization for correlation features with greater than 0.5 a plot is displayed, on x-axis feature vectors(SEND_SMS, attachinterface, android.os.Binder,

serviceConnection, bindservice, onserviceConnected, transact) on y-axis correlation values (-0.6 to 0.6), the SEND_SMS is highest correlation with 0.5 accuracy. This code is all about the finding the feature correlation.

Comparison with kaggle code

Both paragraphs detail rigorous data handling and analysis steps and the effectiveness of various models in a classification task. The precious lays out a foundational data processing workflow, important for setting up the dataset for model application, which is detailed to our present kaggle. It expands on how different models perform specifically in the context of Android malware detection, providing a comprehensive evaluation of each model's strengths and limitations, suggesting practical scenarios where each might be most effectively deployed. Overall, the deep learning model demonstrates superior performance across all metrics, indicating its high utility in scenarios where accuracy and model performance are critical, despite potential drawbacks like computational demand and lack of interpretability.

Discussion:

The effectiveness of a model in Android malware detection is often judged by its accuracy, recall, and precision, which determine its ability to correctly classify applications and minimize false positives and negatives. While traditional models like LR, DT, and SVC provide a good balance of performance and interpretability, ensemble methods like RF improve on single-estimate models by aggregating predictions, thus typically yielding better generalization. Deep learning excels by leveraging complex structures within the data, but this comes at the cost of increased computational requirements and a lack of transparency in decision-making. This can be a significant trade-off when considering the deployment in a security-sensitive environment where understanding the reasoning behind a classification may be as important as the classification itself. In practice, the choice of a model would depend on several factors, including the availability of computational resources, the need for interpretability, the tolerance for false positives/negatives, and the specific characteristics of the malware detection task at hand. It is also important to consider the operational context—such as whether the model will be used in real-time systems or for batch processing—as this impacts the acceptable latency in malware detection. In conclusion, while deep learning offers the most robust performance for Android malware detection, each model has its own set of trade-offs. A balanced approach might involve using a simpler, more interpretable model for initial screenings and a

complex model like deep learning for in-depth analysis. Ultimately, the choice of model should align with the specific needs and constraints of the deployment environment.

Conclusion

In concluding the analysis of machine learning techniques for Android malware detection, our investigation reveals that traditional methods such as Logistic Regression, Decision Trees, and Support Vector Classifiers hold their ground in terms of performance and ease of use. These methods are less computationally intensive, making them ideal for resource-constrained environments. However, they might not capture complex patterns as efficiently as more sophisticated methods, potentially leading to under fitting.

Ensemble methods like Random Forest stand out for their improved accuracy and generalizability, which help in reducing the risk of overfitting. Moreover, deep learning approaches have shown superior performance, particularly in handling high-dimensional data and modeling non-linear relationships, although they require significant computational resources and suffer from a lack of transparency in decision-making.

Looking forward, the field of malware detection would benefit from the development of hybrid models that leverage the strengths of both traditional and advanced techniques, potentially improving accuracy while maintaining interpretability. Enhancements in feature engineering and data augmentation could also provide a richer dataset for model training, which is especially valuable for under-represented classes. Additionally, optimizing computational efficiency remains a key goal to facilitate the deployment of sophisticated models on devices with limited processing capabilities.

Ensuring the robustness and generalizability of these models through techniques like cross-validation and adversarial testing is crucial, as it would validate their effectiveness against a diverse array of malware samples and evasion techniques. The creation of user-friendly applications incorporating these models, with an emphasis on user experience, would also be an important step towards practical implementation.

With continuous learning systems, models can evolve in response to new threats, maintaining their efficacy over time. While the current results are promising, ongoing research must continue to refine these methods, seeking a balanced approach that prioritizes performance, computational efficiency, and interpretability, ensuring that the models stay resilient against emerging malware threats.

Participant Activities:

Venkata Sai Siva Padmini Surekha Aripirala: Evaluating classical models, Evaluating deep learning models.

Keerthi Reddy Chimalapati: Comparison with other kaggle works, Documentation.

Sai Kiran Duvvuri: Training Methods, comparison of all previous models using (performance metrics, accuracy, recall, ROC etc).

References

- [1] S. MELLATI, "Android Malware Detection," 2023.
- [2] A. SARAH, "Android Malware Analysis," 2021.
- [3] JASJEET83, "Jasjeet_BITS_Assignment_AI&ML," 2023.
- [4] A. MAC, "RandomTreeClassifier," 2023.
- [5] TMLEYNCODES, "MALWARE DETECTION USING DT | SVM | LR | NN," 2023.
- [6] S. T. ADDURU, "Android Malware Dataset for Machine Learning," 2023.