CSE 422- Artificial Intelligence
Lab Section 04

Project Title: Detection of Potential Malware in Android Devices

Submitted By-

S M Rakib Hasan (22241038)
Nasar Ahmed Rashed (18101313)
Nayema Ahmed (20101469)
Shahriar Hasan Mickey (21141013)

Table of Contents

1. Introduction

Smartphones have become widely available in recent years and are available to people all over the world, causing security problems. According to the study method, uploading an app to the Android app store is less restrictive than the iOS App Store. Hackers have used Google Play for years to implement a complex backdoor capable of collecting several types of sensitive information. There are a large number of Android applications that are developed and submitted to the Play Store. Therefore, a powerful malware detection system is needed to detect malware. This is important for apps published on the Play Store. Android malware is similar to many other types of malware on desktops and laptops targeting Android phones and tablets. Malware or code designed to damage the user's device, e.g., trojans, adware, ransomware, spyware, viruses, or phishing applications, is called mobile malware. More than 140 million new malware samples have been discovered, according to the latest statistics from 2015. In the 2019 survey Made by researchers at Check Point, they found it interesting that the number of cyber attacks targeting smartphones and other devices has increased by 50 percent compared to last year. Many applications passed the first screening test during approval, despite the fact that they contained malicious content. Malware is difficult to distinguish. Therefore, a strong real-time Android malware detection system that can quickly analyze malicious contents, the contents that are difficult to detect before the malware damages the device, is very crucial to be made and used into real tile implementation and that is the focus of this project. CICMaldroid has been used for this work, which is relatively new and consists of the latest malware samples that can be successfully detected. In addition, the work is strong in high dimension data, which makes it work effectively when used as a new and corresponding independent dataset. The prototype application is demonstrated in this study, which has several areas to improve in the future, e.g., it can be made more dynamic and functional with real-time data.

2. Dataset Description

The dataset that is utilized for this work is the CICMalDroid 2020. This study aims to create a malware detection system for Android applications that is effective enough to recognize malware in real time. These were assembled from various sources, including the VirusTotaladministration, the Contagio security blog, AMD, and other datasets from ongoing investigations. Compared to other publicly accessible datasets, the samples in this one are recent, sophisticated, and dynamic. They were collected between 2017 and 2018. In this manner the dataset is separated into 5 particular classifications, which are Adware, Banking malware, SMS malware, Riskware, and Harmless.

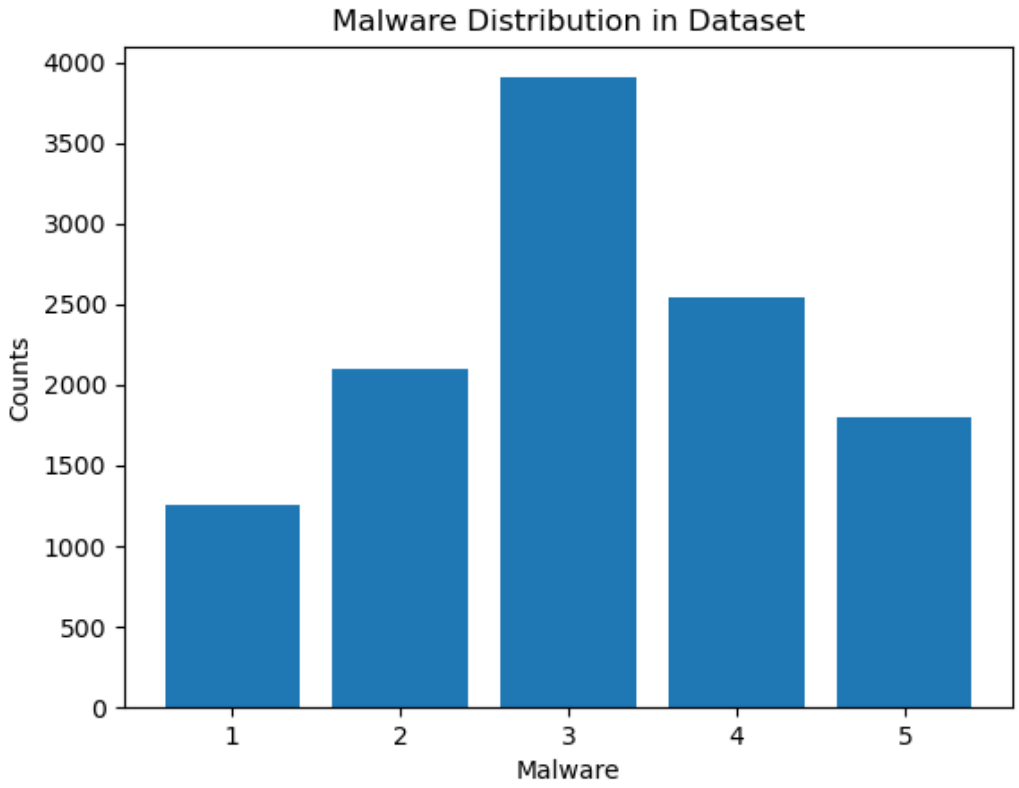

Fig. Malware Distribution in Dataset

CopperDroid, a VMI-based unique investigation framework, was utilized to progressively break down the information to consequently reproduce low-level operating system explicit and undeniable level Android Specific ways of behaving of Android tests.

13,077 out of 17,341 samples were successful, while the remaining ones were unsuccessful due to memory allocation issues, invalid APK files, and timeouts. 3,904 SMS malware samples, or nearly 34% of the total, were discovered while analyzing samples from various groups. In expansion, 2,546 Riskware were found which is almost 22% of the absolute examples. Adware involves 1253 which is nearly11% of the absolute examples. Banking has 2,100 which is around 18% lastly Harmless is 1,795 which is 15.5% around.

In CopperDroid, the APK documents were assessed, and the runtime nature was kept in log records. CopperDroid's output analysis results are in JSON format, making them easy to parse and use with other supplementary data. Statically extricated information like expectations,consents, and administrations, recurrence counts for different document types, muddling occasions, and touchy Programming interface summons are among the review's outcomes. The review involved powerfully noticed ways of behaving, which were parted into three classifications:

- **Framework calls**
- **Folio calls**
- **Composite ways of behaving.**

The dataset's 470 features have been broken up into three sections. Among these 7.4% of the information has a place with the Composite Ways of behaving area, 29.6% addresses the Programming interface and Framework calls bunch and 63.0% of the information falls under the Folio Calls area.

Source URL- https://www.unb.ca/cic/datasets/maldroid-2020.html.

Reference-

Samaneh Mahdavifar, Andi Fitriah Abdul Kadir, Rasool Fatemi, Dima Alhadidi, Ali A. Ghorbani; **Dynamic Android Malware Category Classification using Semi-Supervised Deep Learning**, The 18th IEEE International Conference on Dependable, Autonomic, and Secure Computing (DASC), Aug. 17-24, 2020.

Samaneh Mahdavifar, Dima Alhadidi, and Ali A. Ghorbani (2022). **Effective and Efficient Hybrid Android Malware Classification Using Pseudo-Label Stacked Auto-Encoder**, Journal of Network and Systems Management 30 (1), 1-34.

3. Dataset Preprocessing

Preprocessing involves preparing raw data for analysis by transforming it into a more useful and efficient form. This step helps ensure that the data is accurate, consistent, and suitable for analysis. [**NOTE: Our dataset has no NULL values**]. The preprocessing steps that we took are as follows:

Feature Selection

One of the initial steps in our preprocessing pipeline was feature selection. The dataset had a large number of features, which could potentially lead to overfitting and increased computational complexity. To mitigate these issues, we employed an ANOVA-based feature selection method. This method ranks the features based on their F-values with respect to the target variable and selects the top features that exhibit the most significant relationships.

After applying the ANOVA-based feature selection, we reduced the feature space from 470 to 120 features, retaining those features that contributed the most to explaining the variance in the target variable. This selection process aimed to improve model performance and reduce dimensionality while retaining the most relevant information.

Handling Categorical Data

Our dataset contained categorical variables, which needed to be transformed into numerical representations for model compatibility. We addressed this by utilizing appropriate categorical encoding techniques. However, due to the prior conversion of categorical values to numerical ones, and no requirement for further encoding or deletion of rows, no additional processing was necessary in this regard.

Dealing with Missing Values

The initial dataset was thoroughly examined for missing values. Fortunately, no missing values were detected, which saved us from having to deal with imputation techniques. The absence of missing values ensured that our models could be trained on complete data without introducing bias or error.

4. Feature Scaling

Feature scaling is a crucial step in the preprocessing pipeline, especially when working with machine learning algorithms that are sensitive to the scale of input features. In our project, we employed the StandardScaler to normalize the features and ensure that they all had comparable scales.

The **StandardScaler** is a popular method for feature scaling, as it transforms the data such that it has a mean of 0 and a standard deviation of 1. This scaling technique is particularly useful when the features have different ranges or units, which could otherwise lead to biased model training.

We applied the StandardScaler to our dataset after the feature selection step. This ensured that the reduced set of 120 features were all standardized, allowing our machine learning models to perform optimally without being influenced by the varying scales of the features.

We implemented the StandardScaler from the scikit-learn library. After fitting the scaler on the training data, we applied the learned scaling parameters to both the training and testing datasets. This ensured that our models were evaluated on standardized data to provide accurate performance metrics.

By following this practice of fitting the scaler to the training data and only transforming the test data, we simulated the real-world scenario where the model is exposed to new, unseen data. The model should be evaluated on how well it generalizes to this new data, and using the learned scaling parameters from the training data helps maintain the separation between the training and testing processes, preventing data leakage and ensuring accurate performance evaluation.

5. Dataset Splitting

To evaluate the performance of our machine learning models accurately, we divided the dataset into training and testing subsets. The train_test_split function from the scikit-learn library was used for this purpose. Our dataset contained a total of 11,598 data points.

Stratified Splitting

Given the importance of maintaining class distribution integrity in both the training and testing sets, we performed a stratified split. Stratified splitting ensures that the proportion of different classes in the original dataset is preserved in the training and testing subsets. This is particularly important when dealing with imbalanced datasets to prevent one class from being underrepresented in either the training or testing data.

Split Ratio

We chose an 80-20 split ratio, with 80% of the data allocated to the training set and 20% to the testing set. This ratio strikes a balance between having sufficient data for model training and a reasonable amount of data for evaluating model generalization.

Benefits of Stratified Splitting

Stratified splitting offers several benefits:

- **Preserved Class Distribution**: By ensuring that the class distribution remains consistent across the training and testing sets, we prevent bias and improve model evaluation on diverse data.
- **Mitigated Risk of Overfitting**: The model is less likely to overfit when evaluated on testing data that closely resembles real-world distribution.

Final Split

After applying stratified splitting, we obtained a training set with 9278 and a testing set with 2320.

6. Model Training and Testing

We trained and tested our dataset on five different models. The training and testing details are as follows:

1. **Naive Bayes**:

We trained on Naive Bayes classifier and got the following results on the test set:

- Naive Bayes Classifier Accuracy: 0.5892
- Naive Bayes Classifier Precision: 0.6936
- Naive Bayes Classifier Recall: 0.5892
- Naive Bayes Classifier F1-Score: 0.5462

Overall, it performed poorly. The confusion matrix is shown here:

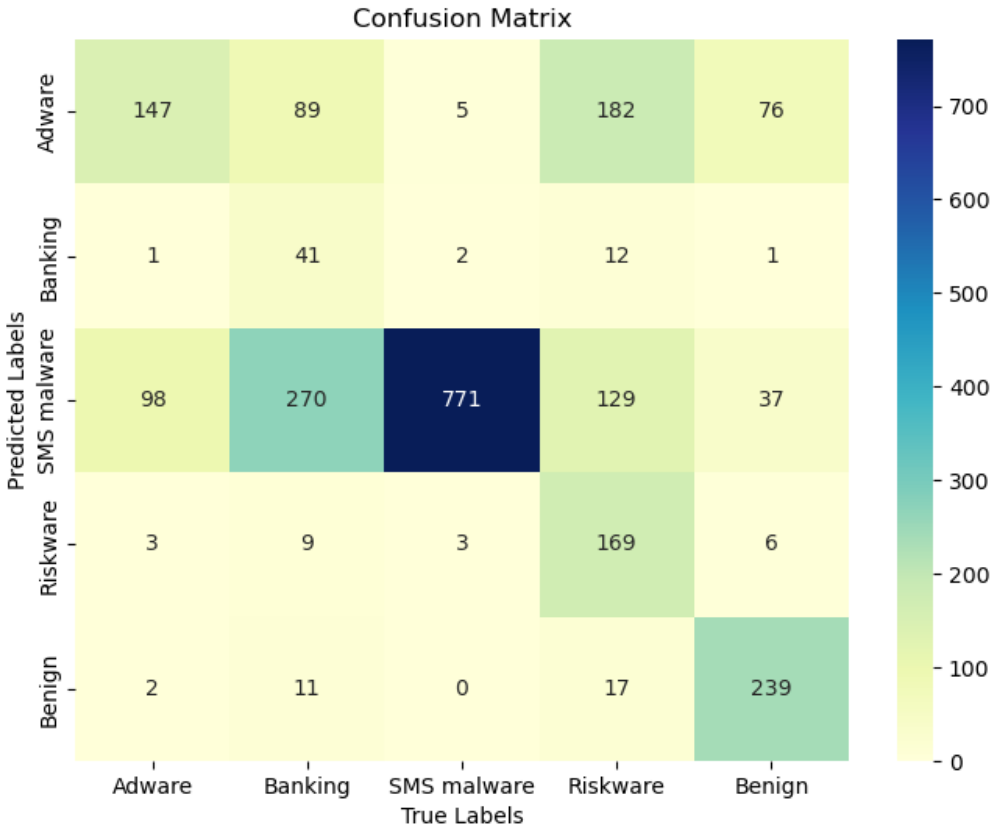

Fig. Confusion Matrix in Heatmap of Naive Bayes

2. **Random Forest Classifier**:

We trained our dataset on the random forest classifier with 300 "n_estimators" and random state 42. We got the following results:

- Random Forest Classifier Accuracy: 0.9422

- Random Forest Classifier Precision: 0.9431
- Random Forest Classifier Recall: 0.9422
- Random Forest Classifier F1-Score: 0.9421

It performed fairly well. Its confusion matrix is given as-

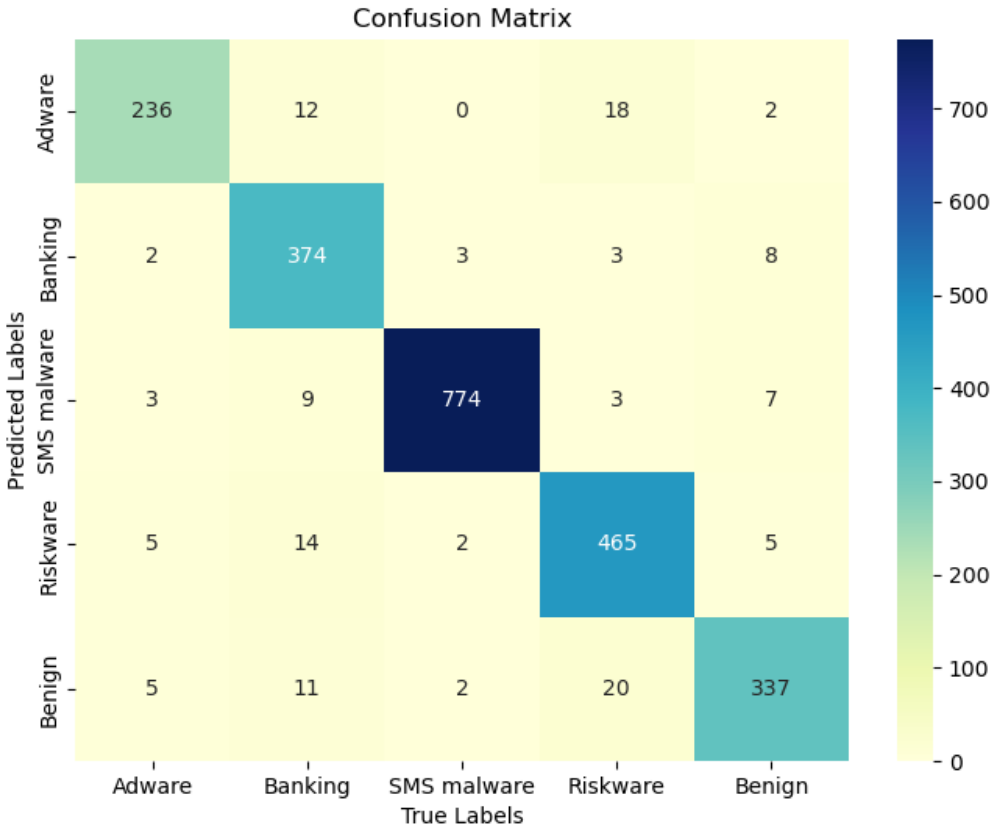

Fig. Confusion Matrix in Heatmap of Random Forest Classification

3. Logistic Regression:

After training and testing on logistic regression with one million iterations, we got the following results:

- Logistic Regression Accuracy: 0.8078
- Logistic Regression Precision: 0.8079
- Logistic Regression Recall: 0.8078
- Logistic Regression F1-Score: 0.8033

It also performed quite well. The confusion matrix is shown below:

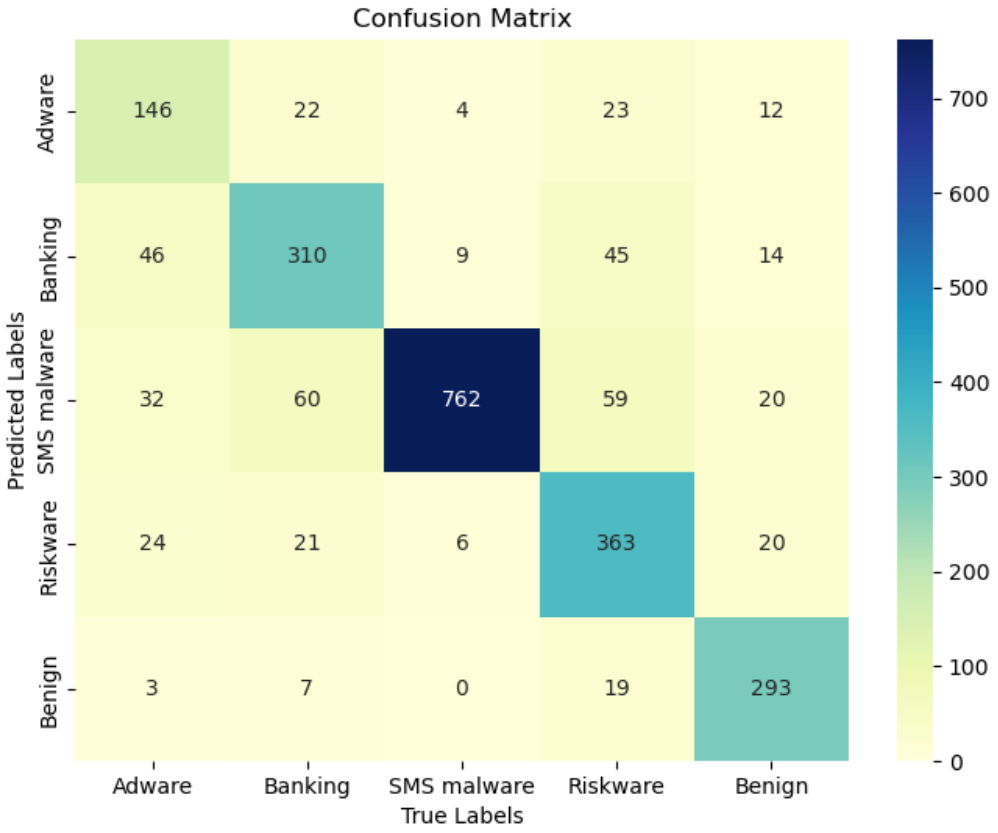

Fig. Confusion Matrix in Heatmap of Logistic Regression

4. **Support Vector Machine:**

Afterwards, we trained and tested on the support vector classifier with linear kernel and random state 42. We got the following results on the testing data:

- SVM Classifier Accuracy: 0.8263
- SVM Classifier Precision: 0.8264
- SVM Classifier Recall: 0.8263
- SVM Classifier F1-Score: 0.8234

The confusion matrix is as follows:

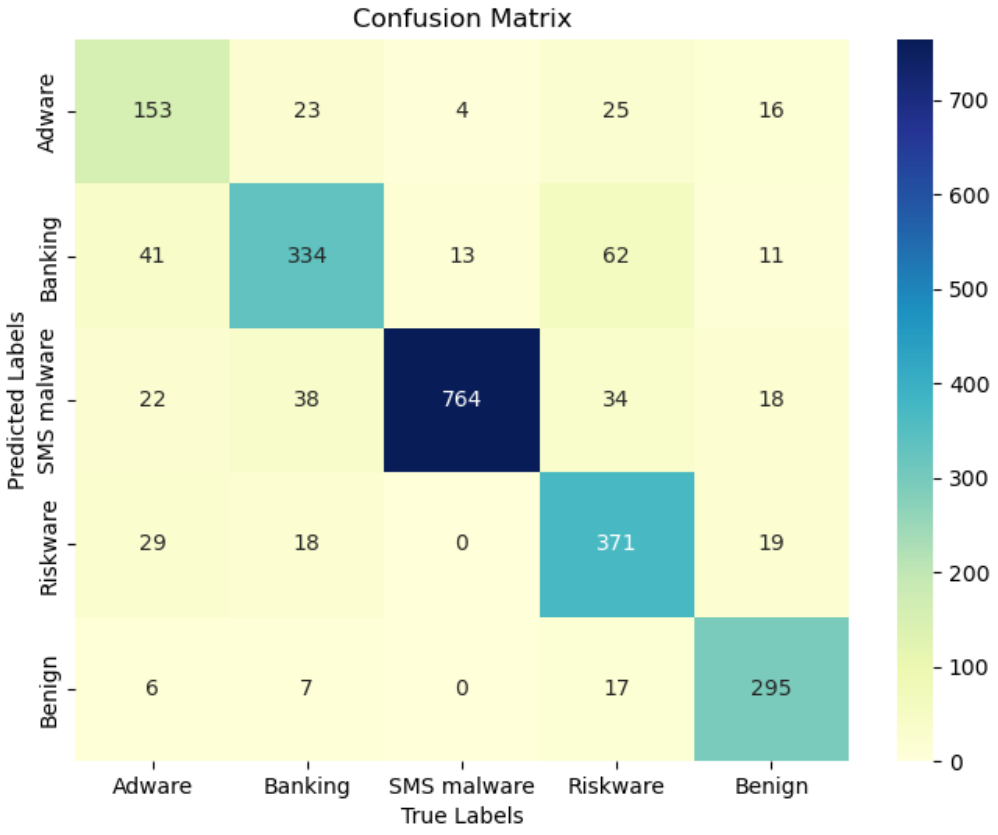

Fig. Confusion Matrix in Heatmap of Support Vector Machine

5. KNN Classifier:

Finally, we trained and tested on the KNN classifier. For choosing the optimal number of K, we tried 6 values using cross-validation and then trained on the most optimal K value. We got the following results:

- Best k: 3
- K-Nearest Neighbors Classifier Accuracy: 0.9009
- K-Nearest Neighbors Classifier Precision: 0.9045
- K-Nearest Neighbors Classifier Recall: 0.9009
- K-Nearest Neighbors Classifier F1-Score: 0.9011

The confusion matrix is given as-

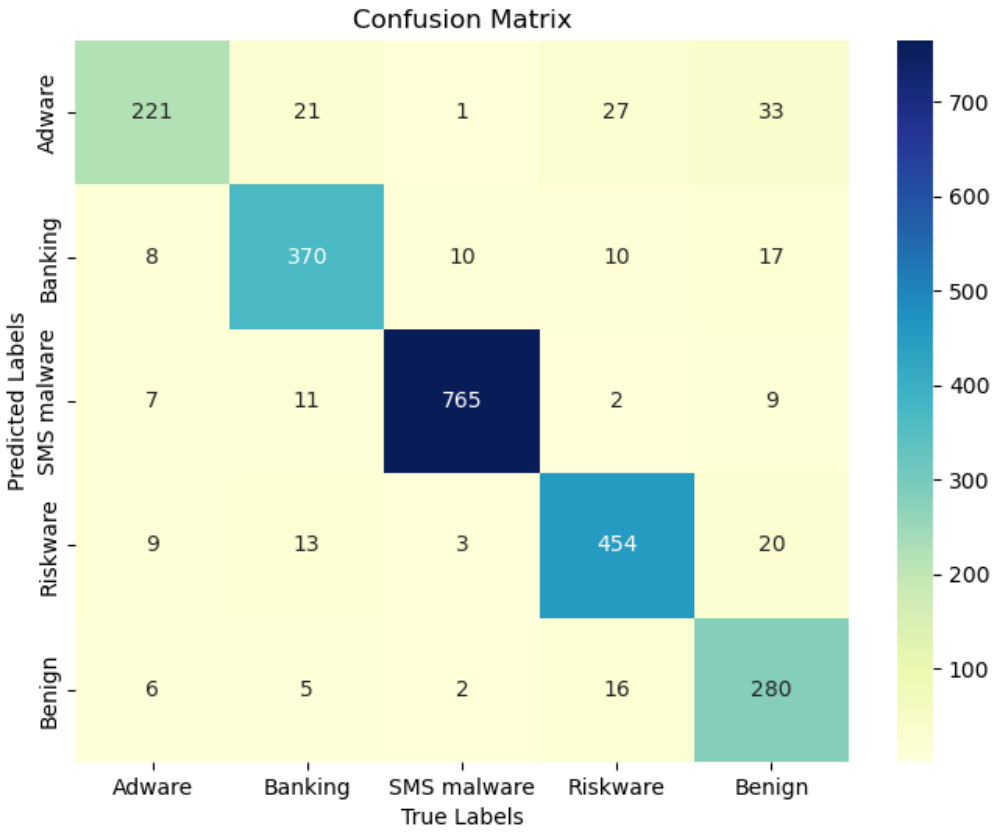

Fig. Confusion Matrix in Heatmap of KNN Classification

7. Comparison Analysis:

After training and testing on all the models, we found the following results:

Models	Accuracy	Precision	Recall	F1-Score
Logistic Regression	0.807759	0.807916	0.807759	0.803350
SVM	0.826293	0.826378	0.826293	0.823396
Random Forest	**0.942241**	**0.943053**	**0.942241**	**0.942148**
KNN	0.900862	0.904487	0.900862	0.901129

Table: Comparison of metric scores of different models

From the table, we see Random Forest Classifier performed best on the testing data while naive bayes classifier had the least scores in testing. Moreover, KNN Classifier have also achieved good scores. The metric scores are plotted as follows for visual comparison:

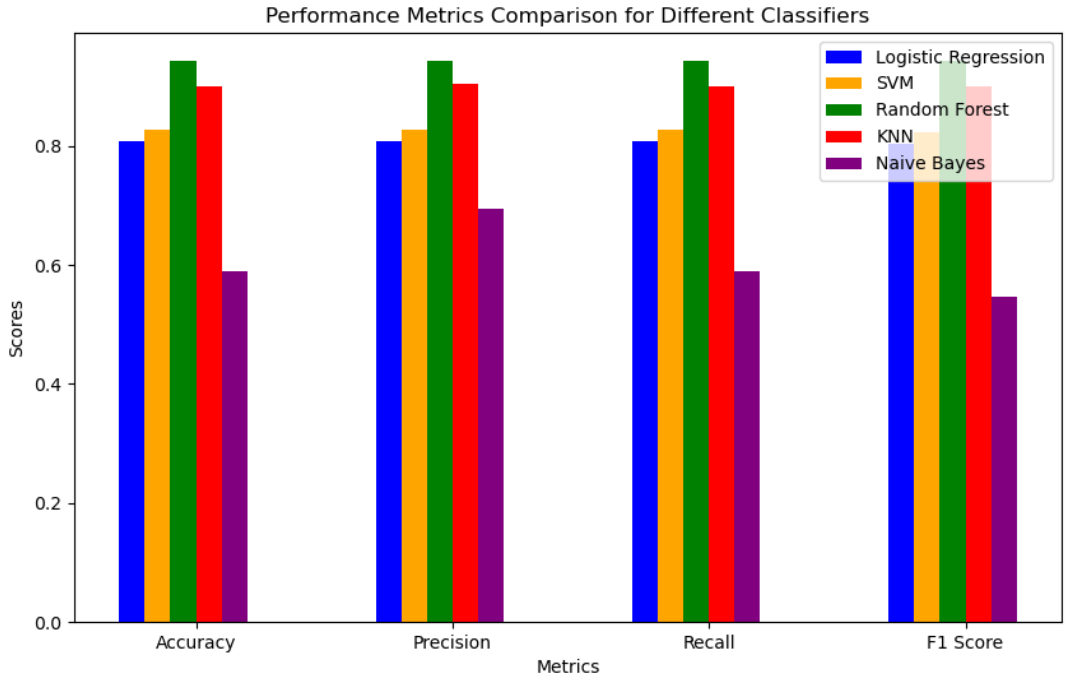

Fig. Performance Matrix Comparison for Different Classifiers

So the Random Forest classifier could detect potential malware from the data with better accuracy. We can clearly visualize the differences from the following comparison plot:

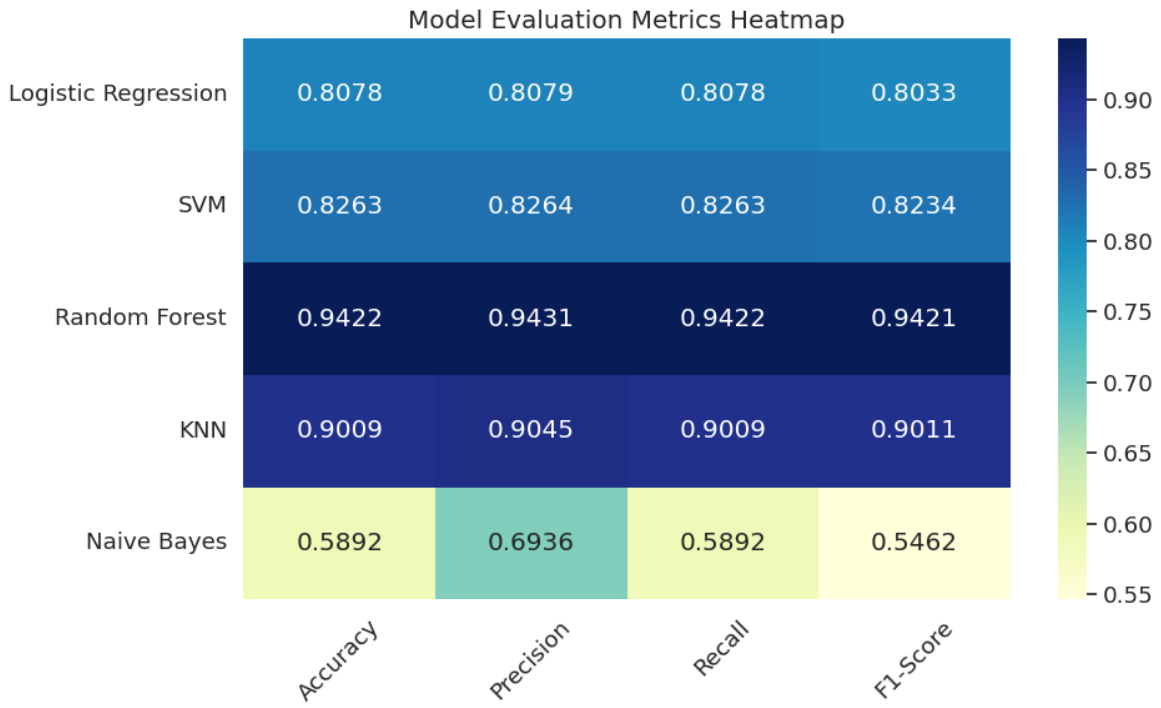

Fig. Model Evaluation Metrics Heatmap

8. Conclusion:

The importance of effective malware detection has grown significantly in ensuring the security and privacy of mobile devices connected to networks. This need arises due to the constant evolution and strengthening of malicious software, particularly in the realm of Android mobile phones, which are experiencing a substantial rise in user numbers. While significant progress has been made in intrusion detection to safeguard privacy and protect electronic devices, malicious actors are also advancing rapidly, presenting serious challenges to network security. As a result, the demand for stronger security measures has become urgent. Additionally, the influx of sophisticated malware into personal devices is happening at an accelerated rate, surpassing the capabilities of existing intrusion detection systems known for their sluggishness and inefficiency. Introducing real-time malware detection holds promise in addressing these issues due to its dynamic and swift nature, although this field remains relatively unexplored.

Our project steps in to provide an extra layer of security by incorporating more advanced features into the current security framework, addressing gaps in the research on real-time malware detection. Future improvements could involve infusing the developed application with dynamic attributes. Notably, our current study concentrates on identifying malware in Android applications, but there is potential to expand the scope to include applications supported by the iOS platform, utilizing relevant datasets.