4-27-2022

Android Malware Detection using Machine Learning Techniques

Mohamed Salem Alhebsi
msa4302@rit.edu

RIT

Android Malware Detection using Machine Learning Techniques

by

Mohamed Salem Alhebsi

A Capstone Submitted in Partial Fulfilment of the Requirements for the Degree of Master of Science in Professional Studies: Data Analytics

Department of Graduate Programs & Research

Rochester Institute of Technology

RIT Dubai

Date (April 27, 2022)

RIT

Master of Science in Professional Studies: Data Analytics

Capstone Approval

Students' Name: Mohamed Salem Alhebsi

Capstone Title: Android Malware Detection using Machine Learning Techniques

Capstone Committee

Dr. Sanjay Modak	**Chair of Committee**	**17/May/2022**
Dr. Ioannis Karamitsos	**Member of committee**	**16/May/2022**

Acknowledgments

I would like to express sincere gratitude to the committee chair Dr. Sanjay Modak and my supervisor Dr. Ioannis Karamitsos for providing their guidance, comments, and suggestions throughout the course of this project. I offer my appreciation for the learning opportunities provided by the committee.

I also thank all of my course instructors throughout the program, without whose guidance and support it would not be possible to attempt to solve such a complex analytical problem of malware detection. In particular, the Intro to Data Mining course served as a foundation for learning the basic concepts related to data exploration and machine learning.

Lastly, I express my appreciation to the researchers who have been working on the problem of malware detection. From the papers I studied while implementing this project, it was clear that a lot of work has been done in this area, and I learned a lot about the concepts and techniques used in this field. In addition to the researchers, I should also thank those who compile and maintain labeled malware and benign applications databases, which are valuable inputs to projects like this.

Abstract

Android is the world's most popular and widely used operating system for mobile smartphones today. One of the reasons for this popularity is the free third-party applications that are downloaded and installed and provide various types of benefits to the user. Unfortunately, this flexibility of installing any application created by third parties has also led to an endless stream of constantly evolving malware applications that are intended to cause harm to the user in many ways.

In this project, different approaches for tackling the problem of Android malware detection are presented and demonstrated. The data analytics of a real-time detection system is developed. The detection system can be used to scan through installed applications to identify potentially harmful ones so that they can be uninstalled. This is achieved through machine learning models.

The effectiveness of the models using two different types of features, namely *permissions* and *signatures*, is explored. Exploratory data analysis and feature engineering are first implemented on each dataset to reduce a large number of features available. Then, different data mining supervised classification models are used to classify whether a given app is malware or benign. The performance metrics of different models are then compared to identify the technique that offers the best results for this purpose of malware detection. It is observed in the end that the signatures-based approach is more effective than the permissions-based approach. The kNN classifier and Random Forest classifier are both equally effective in terms of the classification models.

Keywords: Android, malware, benign, permissions-based, signatures-based.

Table of Contents

List of Figures

List of Tables

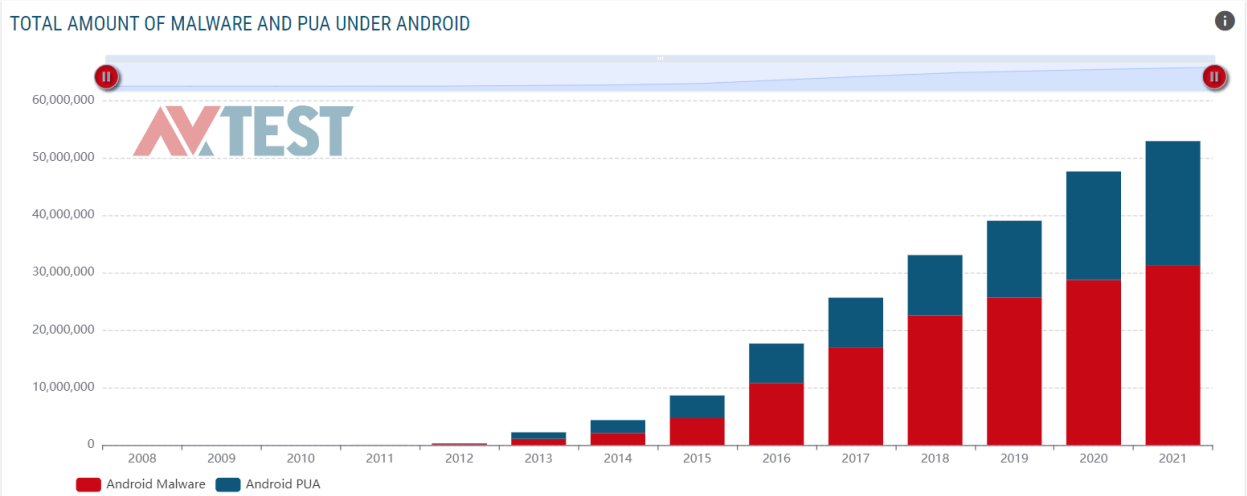
Chapter 1

1.1 Background Information

The first Android smartphone was launched in September 2008, and shortly thereafter, smartphones powered by the new open-source operating system were everywhere. In 2021, almost 12 new enhanced versions of Android were released, and it is the most widely used mobile operating system in the world, with an 84% share of the global smartphone market [1].

With this level of adoption coupled with the open-source nature of Android applications, security attacks are becoming more and more ubiquitous and seriously threaten the integrity of Android applications. Statistics show that more than 50 million malware and potentially unwanted applications (PUA) have been identified for Android [2].

Figure 1: Number of Android Malwares Per Year



Researchers have been studying the nature of malware applications for many years and have categorized them into different families [3]

- *Trojans*: These appear as benign apps and aim to steal the user's confidential information without the user's knowledge.
- *Backdoors*: These exploit root grant privileges and aim to gain control over the device and perform any operation without the user's knowledge.
- *Worms*: This malware creates copies of itself and distributes them over the mobile device's networks.
- *Spyware*: These appear as benign apps designed to monitor the user's confidential information, such as messages, contacts, location, bank information, etc., for undesirable consequences.

- *Botnets*: A botnet is a network of compromised Android devices controlled by a remote server.
- *Ransomware*: This malware prevents users from accessing their data by locking the mobile phone until a ransom amount is paid.
- *Riskwares*: These are legitimate that malicious authors exploit to reduce the device's performance or harm their data.

Standard approaches to detecting malware can be of two types – Static and Dynamic.

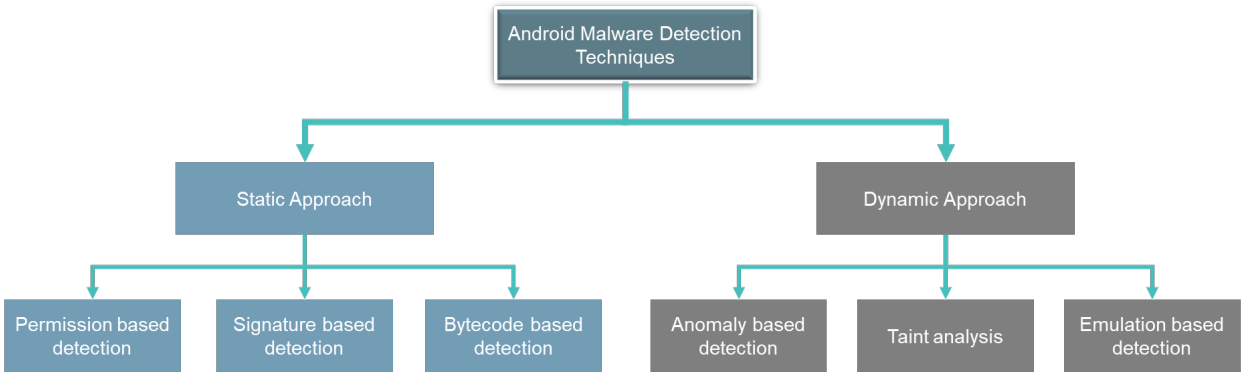
- **Static Approach**:

  In this approach, the functionalities and maliciousness of an application can be checked by disassembling and analyzing its source code without actually executing the application.
- **Dynamic Approach**:

  In this approach, the application is examined during execution and can help identify undetected malware by static analysis techniques due to code obfuscation and encryption of the malware.

These approaches can be further sub-divided based on the method of anomaly detection. Some of these sub-divisions are shown in the diagram below –

Figure 2: Android Malware Detection Techniques



1.2 Project Definition and Goals

The goals and objectives of the project are two-fold:

1. to achieve good accuracy in detecting malware from samples of benign and malware applications using multiple approaches, and
2. to compare the results of different approaches and algorithms and provide recommendations on the best strategy for malware detection

Specifically, I focused on two static approach methods – the Signature-based and Permission-based detection methods. These two methods are described below.

- **Permission-based malware detection:**

  In Android smartphones, the permissions granted to an application are essential in governing the access rights given to that application. E.g., At the time of installation, the user can grant an

application the permission to send information over the internet or access contacts. These permissions are assumed to be needed for the application to perform its designed functions. However, many times, applications request permissions that are not required for their functionality.

By analyzing the combinations of permissions requested by benign and malware applications, it is possible to identify whether an application is malware or not. This can be achieved by training machine learning classification models with data from known malware and benign applications.

- **Signature-based malware detection:**

  This is a method that is commonly used by commercial antimalware products. In this method, signatures are generated for the various API calls that the application will make. By identifying patterns of such signatures and comparing these signatures with existing malware families' signatures, it is possible to detect whether an application is benign or malware.

Project Definition:

In this project, I demonstrate both permission-based and signature-based methods using two different datasets.

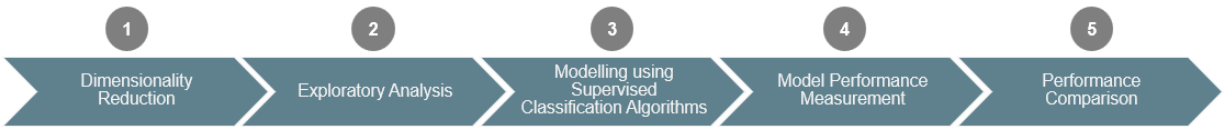
By using publicly available labeled data sources, different classification models are built to distinguish between malware and benign applications. The first data source contains information on permissions granted to the applications, while the second data source contains information on API call signatures of the applications.

Various data mining models are trained, and their performance metrics, such as precision and recall, are analyzed and compared. This comparison is first made for different classification models within an approach. Then, the best results for each approach are compared to understand which of the two systems is better for detecting malware.

1.3 Analysis Methodology

The step-by-step methodology followed in this project is illustrated below.

Figure 3: Project Methodology



Dimensionality Reduction:

The datasets used in the project had a large number of features, so it was necessary to reduce the dimensions before using classification algorithms.

The dimensions are reduced using a combination of three methods –

- ***Frequency Counts:***

Features that contain only one unique value are removed because they cannot help differentiate between the classes

- ***Correlations:***
  If there are any features that are highly correlated to each other, then only one of those features is retained.
- ***Chi-Square Test:***
  This statistical test for categorical variables is conducted to check which of the features are more associated with the class variable. The top 20 of such highly associated features are selected.

Exploratory Analysis:

In this step, visualizations are used to deep dive into the data.

- The data is split into two classes – malware and benign.
- Each feature is explored individually, and I try to see the distribution of values of the feature when split by the class variable.
- This step helps in identifying the potentially differentiating features. The features whose distributions differ significantly between malware and benign classes will likely be the most important features.

Modeling using Supervised Classification Algorithms:

- The dataset is first broken up into training and testing datasets.
- The training set is used to train the classification algorithm, while the testing set is used to validate the results seen on the training set and address overfitting issues.
- Three different supervised classification algorithms are used for each approach to perform the classification into benign and malware classes.
- These algorithms are – kNN, Logistic Regression, and Random Forest.

Figure 4: Supervised Classification Algorithms

Model Performance Measurement:

- Performance measurement of the classification process is first done using the confusion matrix.

Figure 5: Confusion Matrix Illustration

		Predicted label	
		Benign	Malware (Intrusion)
True label	Benign	TN	FP
	Malware (Intrusion)	FN	TP

- Additional performance measures are also calculated and used as the basis for comparison.
    - **Precision:** measures how many of the applications the model classifies as malware are true malware – should be as high as possible
    - **Recall:** measures how many of the malware samples in the dataset are correctly classified as malware – should be as high as possible
    - **F1-score:** It is the harmonic mean between precision and recall and measures the overall accuracy of the model – it should be as high as possible.

Performance Comparison:

After all the models are built, results are compared to identify the best model and approach.

- A recall score is selected for comparison.
- In a malware detection problem, it is more important to identify all the true malware correctly than to ensure that all the malware identified is actually true malware. Therefore, recall is more important than precision.
- A comparison of recall scores is made first between the three classification algorithms within each approach. This helped in identifying the best algorithm for each approach.
- A comparison of recall scores is then made between the two approaches. This helped in identifying the best approach for the purpose of malware detection.

1.4 Limitations of the Study

Limitations of using the permission-based method:

- Differences between permissions requested by benign applications and malware applications are small, and thus challenging to get excellent performance

Limitations of using the signature-based method:

- Since the signature-based method compares the signature of an application with that of existing malware, it cannot detect unknown malware types
- This method can be evaded by code obfuscation, method renaming, string encryption techniques, etc., because it can only identify existing malware and fails against unseen variants.

Limitations of the overall project:

- Due to limited resources, only 20 features are finally used in modeling. Although possibly the best 20 features are selected through feature selection methods, it is possible that a few important may have been missed.
- The project is highly dependent on the datasets created by external researchers. The authenticity and robustness of the data are not verified. Consequently, we should be careful when generalizing the results of the project.
- The project is limited to the two Android malware detection approaches – permissions-based and signatures-based. There could be other approaches that will give better performance than the ones considered in this project.

1.5 Sources of Data

The raw permissions dataset is obtained from the following source -
Mahindru, Arvind (2018), "Android Malware and Normal permissions dataset," Mendeley Data, V5, doi: 10.17632/958wvr38gy.5
The signatures data is obtained from Malgenome - Android Malware Genome Project, which is a repository of thousands of Android applications pre-classified into Malware and Benign.
Overview and descriptions of these datasets are further provided in Chapter 3.

Chapter 2

2.1 Literature Review

After conducting a thorough literature review of the research in the area of Android malware detection, I observed different types of objectives of such research. Many research articles are focused on surveying existing methods of solving the malware detection problem. These articles do a systematic review of different techniques that other researchers have used for this purpose and compare the results.

Liu et al. (2020) review in detail different approaches and research status from different perspectives like sample acquisition, data preprocessing, feature selection, machine learning algorithms, and performance evaluation. Finally, Odusami et al. (2018) review existing malware detection methods, including static and dynamic analysis approaches, describe the strengths and weaknesses of each approach, and conclude that machine learning-based methods show the best detection accuracy and thus are promising for the future.

Some studies are focused on choosing the correct feature set. The features used are just as crucial to the end outcome of the malware detection exercise as the techniques and algorithms used to perform the detection. So, these studies provide valuable insight into the right feature set to use. Wen et al. (2017) use a combination of features from both static and dynamic analysis, then apply PCA to reduce the dimensionality of data and use SVM to perform the classification of applications into benign and malware classes. Roy et al. (2020) build a feature extraction module that performs static analysis to map each API call to certain features. Then, feature vectors are generated, and dimensionality is reduced, following which classification algorithms are used to differentiate between benign and malicious applications. Daoudi et al. (2021) convert the bytecode of the application into grey-scale vector images and use 1-dimensional Convolutional Neural Networks to detect malware. This approach circumvents the need for creating comprehensive hand-crafted features and uses the raw bytecode of the application for analysis. Jiang et al. (2020) study the permissions frequently used by malicious applications and identify permissions they call dangerous fine-grained permissions, which better differentiate benign and malicious applications. These features are then used in machine learning models to perform the classification.

Other studies focus more on optimizing the detection algorithm than the feature set. These studies are focused on improving the detection performance by choosing the right machine learning algorithm and/or using various techniques to enhance the performance of traditional algorithms. Rathore et al., 2021 use multiple types of static analysis features and compare the performance of different machine learning and deep learning techniques, both supervised and unsupervised. They found that the baseline Random Forest model without any feature reduction achieved the best performance. Shao et al., 2021 extract features from the Android application package, use the relief feature selection method to select features,

use different sampling strategies to address the class imbalance, and improve traditional ensemble bagging algorithms to achieve the best performance.

There are also studies in the literature that use a combination of multiple techniques instead of applying one technique to improve results. These studies tend to develop a complex approach but can possibly lead to better results. Yerima et al. (2018) use a combination of machine learning algorithms for increased the performance of the detection system. They first perform classification using base classifiers and then re-classify the base classifier predictions using ranking-based algorithms to achieve the final prediction. Almin et al. (2015) analyze permissions requested by applications at the time of installation and perform a combination of clustering and classification to detect malware.

In a study conducted by Syrris and Geneiatakis (2021), the authors start by appreciating how much the Android operating system has, over the years, been advancing in enhancing its robustness. The robustness is associated with the advanced technologies, significant community support, and availability of tons of resources on the internet. However, all these privileges come at a cost on source platforms in which security is compromised. In this regard, malicious applications find a way to bypass some security protocols. In addressing this problem, Syrris and Geneiatakis (2021) state that there are several approaches that can be used to leverage machine learning to detect malware through the help of static analysis data. According to Kumar et al., 2018 statistical analysis and feature extraction are the two main methodologies that support machine learning in malware detection processes.

In research that was conducted by Li et al. (2018), the authors indicated that new malicious Android applications are introduced into the mobile ecosystem every ten seconds. This statistic is worrying, and there are chances of interfering with the mobile ecosystem growth globally if something is not done. Further, the authors acknowledge that there is a need for this problem to be addressed before it affects the integrity of the Android software engineering processes. In combating the problem, the authors acknowledge the need to have a scalable malware detection approach based on the dynamics of the mobile ecosystem and the development of Android applications.

The advancements in terms of technology in developing android operating systems have created more opportunities like the existence of e-commerce, among others. However, it has led to more challenges like cyber-attacks. Among the challenges posed by android devices and the mobile ecosystem as a whole, malicious applications have undoubtedly taken the lead (Christiana et al., 2020). The malware has also continued to advance in terms of sophistication and intelligence such that it has become hard for them to be easily detected through the existing systems. For instance, signature-based systems used for malware detection have become inefficient in detecting advanced malware applications (Christiana et al., 2020). As a result, machine learning techniques are now at the top in dealing with this challenge.

Cybercrimes are rapidly increasing on Android-based devices because of their wide usage across the globe. This increase has made it possible for malicious individuals to engineer malicious applications for their gains and at the expense of the users. In this research, the authors concur with Christiana et al. (2020) that the deployment of machine learning techniques is the option needed for curbing malicious android applications. Malicious applications can be classified using machine learning models to differentiate between benign and malicious android applications (Sharma et al., 2020). Additionally, a comparative analysis aimed at calculating the computational time necessary to detect malicious applications is a requirement necessary in machine learning techniques.

The challenge posed by this mechanism is based on the fact that some large bundle applications cannot be easily scaled hence creating the need for Significant Permission IDentification (SigPID). This approach has an efficiency of about 93.62 percent of malware detection in a particular dataset, making it the best method to detect malware. SigPID uses permission usage to analyze the increasing number of humanoid malware. It is not necessary for the engineers to analyze all humanoid permissions for them to detect the existence of malware (Assisi et al., n.d). Instead, mining the permission information is critical in determining the most important permissions that can easily lead to the classification of malicious and benign applications, hence complementing machine learning in malware detection. Kyaw and Kham (2019) reiterate that a scalable malware detection method that yields optimum results is the use of permission analysis through the SigPID approach. Malware applications are thus identifiable through the analysis of the permission behavior. Additionally, pruning procedures are therefore necessary for identifying the most significant permissions that will provide the desired results through the multilevel pruning methodology.

Android applications are readily available because of the comprehensive community support and other open sources that make it easier for malware engineers to develop more malicious applications. Android devices have been the target for malware applications because of the worldwide reception of the android devices (Kumar et al., 2018). For the purpose of reaching high levels of accuracy when detecting malware, a small subset of specific features should be considered. Furthermore, Android is actively implementing new security controls, including the use of a unique user ID (UID) and system permissions for every application (Rana et al., 2018). Therefore, the use of machine learning classifiers has become one of the best approaches for detecting any android malware in the mobile ecosystem and other android devices.

Fallah and Bidgoly (2019) research demonstrated the need to benchmark machine learning algorithms before the associated techniques can achieve the required level of efficacy. The basis of this approach is identifying the family of particular malicious applications. Moreover, the authors demonstrate the need to use combined techniques to get optimum results that are consistent across the platforms based on the selected datasets or classifications of the malware. In this context, the authors recommend using

machine learning and network-based detection techniques. In the case of machine learning, the detection process should work in both the unsupervised and supervised machine learning methods to get viable results that will be used in the decision-making process. However, this research does not clearly demonstrate how machine learning algorithms will handle the new variants of malware that have not been tested through the existing algorithms. This means that for the machine learning techniques to be effective, the algorithms have to be updated every often to capture and detect new families of malware.

The rising attacks on smartphones result from Android being the most used OS. In this context, the authors give a reason why it has been so easy for attackers to use malicious Android applications to launch attacks. The prompts posed to the user when installing the applications that require them to accept all sorts of necessary permissions before the installation are the main issue. Before a user gives the needed permissions, some applications fail to install, leaving the user with no option but to provide the permissions (Singh et al., 2022). Consequently, some Android applications are not approved by the associated organizations and might be misused in collecting user data that might eventually be misused. For this reason, the application of machine learning algorithms has increasingly been used in detecting Android malware. Android classification algorithms like decision trees, vector machines, and random forests form the basis of machine learning success in detecting malware on Android devices.

Considering the above types of literature available, I try to combine two types of studies in this project. The project's goal is to determine which feature set works best and explore different detection algorithms to determine which works best. It should be noted that the combination of multiple detection techniques is not in the scope of this project.

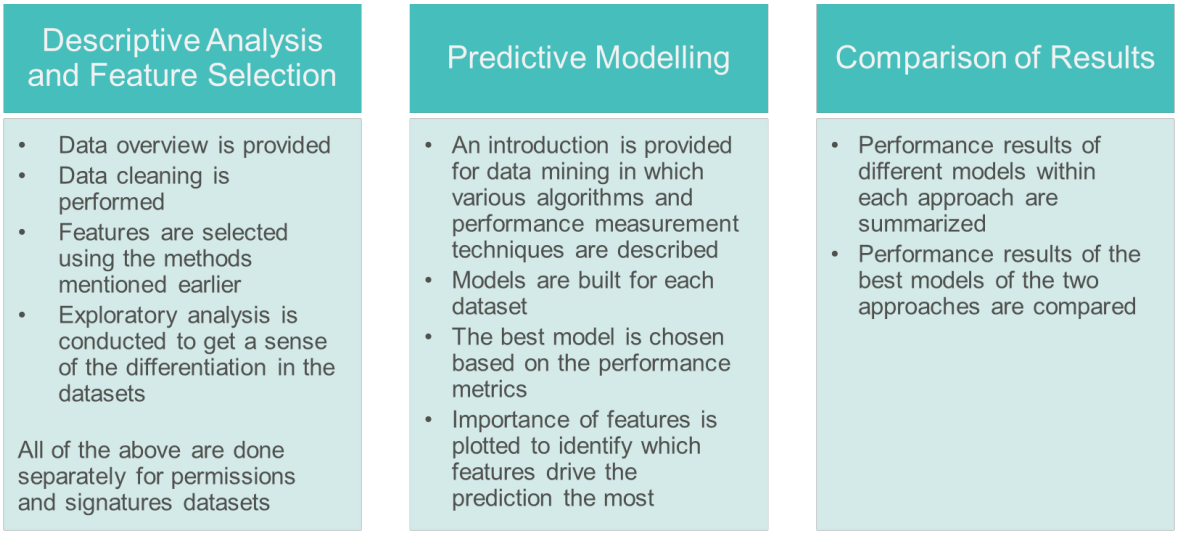
Chapter 3

3.1 Analysis Structure

The detailed analysis is described in the next section. A step-by-step explanation is systematically provided, and the code is also shown along with outputs and results.

The analysis is split into three main sections – Descriptive Analysis, Predictive Modelling, and Comparison of Results. These sections are described briefly in the diagram below.

Figure 6: Structure of the Analysis



3.2 Detailed Analysis

This section describes each analysis and includes the python codes that were used to generate these analyses. As mentioned in the previous section, it is split into two sub-sections – one for Descriptive Analysis and one for Predictive Modelling.

Before deep-diving into the analyses and results, let us first load the libraries required for analysis, visualization, and predictive modeling.

The *panda's* library is used to access the datasets, and visualization libraries like matplotlib and seaborn are used to generate graphs. The *sklearn* library will be used for predictive modeling and related operations.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import itertools
```

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import StratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
```

3.2.1 Descriptive Analysis and Feature Selection

In the following sections, the permissions dataset and the signatures dataset are explored separately. Descriptive analysis is conducted, the number of features in each dataset is reduced through feature selection techniques, and the datasets are prepared for running classification algorithms.

3.2.1.1 Permissions Data

Let's read the permissions data.

```
data_1 = pd.read_csv('dataset_1.csv') # dataset_1 is the permissions data
```

Data Description

Let’s start by seeing how many rows and columns the permissions data has.

```
print('The permissions data has {} row and {} columns'.format(data_1.shape[0], data_1.shape[1]))
```

```
The permissions data has 28849 row and 176 columns
```

There are 176 columns in the dataset. The vast majority of these are permissions-related columns.

Let's see the first few rows of the data.

```
pd.options.display.max_columns = 5
data_1.head()
```

	Package	Category	...	your_personal_information_write_to_user_defined_dictionary	class
0	net.iconchanger	Personalization	...	0	malware
1	com.Jaaru.TruthorDareFree	Comics	...	0	malware
2	com.jrtstudio.iSyncr	Music & Audio	...	0	malware
3	com.ftbsports.fmrm	Sports Games	...	0	malware
4	com.sillycube.android.Rushing	Sports Games	...	0	malware

5 rows × 176 columns

We can see above that each dataset row is identified by the package name (**'Package'**). Information on the category of the app (**'Category'**) is also available. The last column of the dataset (**'class'**) indicates whether the app is malware or benign.

The remaining columns are all permissions related, and they are binary variables - 1 represents the app requests for that permission, and 0 illustrates that the app does not request that permission.

Some permissions column names are shown below.

Table 1: Examples of Permissions-related features

Examples of permissions-related features
default_access_all_system_downloads
default_modify_google_settings
default_change_screen_orientation
default_directly_install_applications
default_read_phone_state_and_identity
default_write_contact_data
hardware_controls_take_pictures_and_videos
your_messages_receive_sms
your_personal_information_read_contact_data

Data Cleaning

- **Missing Values Check:**

  The data should not contain any missing values. If it does, either the missing data should be removed, or some type of missing value imputation needs to be performed.

  This is checked as follows -

```
print('The maximum number of missing values in any column is: {}'.format(data_1.isnull().sum().max()))
```

```
The maximum number of missing values in any column is: 0
```

Therefore, there are no missing values in the data, so no further action is needed.

- **Type Conversion:**

  The data types of the columns should be appropriate for the analysis that will be conducted.

  All variables in the permissions data, except for app identifiers like 'Package' and 'Category,' are read as categorical variables. For performing classification operations, it is necessary for the ***class*** variable to be binary (1 and 0). This type of conversion is done as follows -

```
data_1['class'] = data_1['class'].map({'malware': 1, 'benign': 0})
```

- **Feature Selection**

Since the dataset has 176 columns, not all can be used in predictive modeling. The law of parsimony suggests that the best solution to a problem is the one that uses the least number of features. More features will also cause performance issues with many data mining algorithms. So, some ways of reducing the number of features should be implemented.

Frequency Counts

One way of reducing the number of features is to identify the ones that will not impact the results. There could be some variables that have only one unique value and thus will not help distinguish between malware and benign. That is, these features are 0 for all rows or 1 for all rows.

In the following chunk, the value counts of each column in the dataset are calculated, and identify the columns that are all 0 or all 1.

```
freq_cols = data_1.loc[:,[x for x in data_1.columns.tolist() if x not in ['Package', 'Category', 'class']]].apply(
    lambda col: col.value_counts(normalize=True)).transpose()

select_cols = freq_cols.loc[np.logical_and(pd.isnull(freq_cols[0]) == False,
                                           pd.isnull(freq_cols[1]) == False),:].index.tolist()

print('There are {} columns that are all 0 or all 1'.format(len(freq_cols) - len(select_cols)))
```

```
There are 10 columns that are all 0 or all 1
```

These columns cannot be removed from the original dataset.

```
data_1_select = data_1.loc[:, select_cols + ['class']]
```

Only ten columns were removed, and thus there are still too many features in the data.

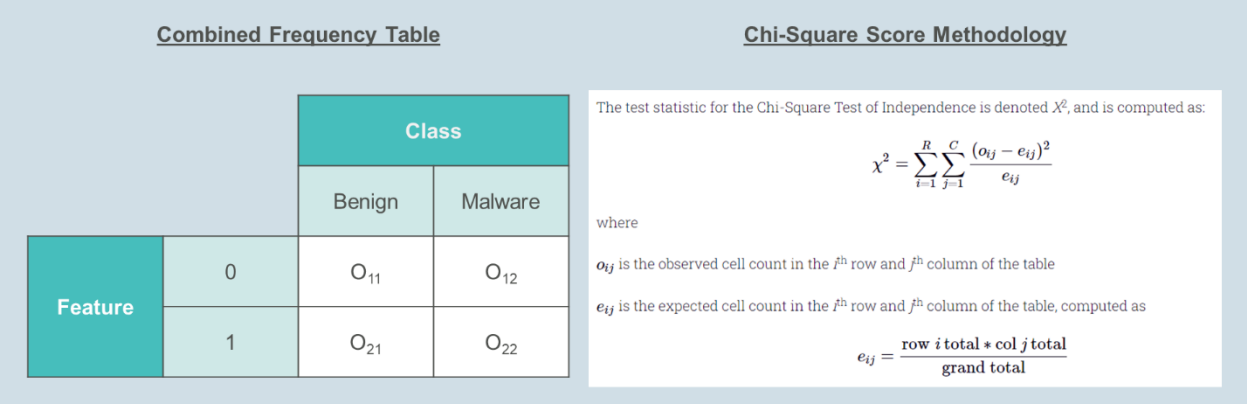
Chi-Square Test

The goal of feature selection is to retain the features that differentiate the classes with the most features using a chi-square test analysis.

The Chi-square test is used for categorical features in a dataset. By calculating Chi-square scores between each feature and the target ***'class'*** variable, it is possible to identify the top differentiating features. For this project, the top 20 features with the best Chi-square scores were selected.

The Chi-square test of independence works by comparing the combined frequency table of the two categorical variables. A chi-square score is calculated using the following methodology -

Figure 7: Chi-Square Test Illustration



If the chi-square score is greater than the critical chi-square value for the chosen significance level, we can say that the feature has some association with the class variable.

By applying this to all the features, the **top 20** features were selected as per their chi-square score with the class variable. The following code achieves this.

```
chi2_features = SelectKBest(chi2, k=20)
X_kbest = chi2_features.fit_transform(data_1_select.loc[:,
                                                        [x for x in data_1_select.columns.tolist() if
                                                         x not in ['Package', 'Category', 'class']]], data_1_select['class'])
```

The top 20 features chosen through this process are as follows –

```
select_cols = [data_1_select.columns.tolist()[i] for i in chi2_features.get_support(indices=True)]
data_1_select = data_1_select.loc[:, select_cols + ['class']]
select_cols
```

```
['default_audio_file_access',
 'default_access_the_cache_filesystem',
 'default_access_to_passwords_for_google_accounts',
 'default_control_location_update_notifications',
 'default_discover_known_accounts',
 'default_force_device_reboot',
 'default_interact_with_a_device_admin',
 'default_permission_to_install_a_location_provider',
 'default_read_phone_state_and_identity',
 'default_write_contact_data',
 'hardware_controls_record_audio',
 'system_tools_format_external_storage',
 'your_accounts_access_all_google_services',
 'your_accounts_contacts_data_in_google_accounts',
 'your_messages_read_sms_or_mms',
 'your_messages_receive_sms',
 'your_messages_send_sms-received_broadcast',
 'your_personal_information_add_or_modify_calendar_events_and_send_email_to_guests',
 'your_personal_information_read_calendar_events',
 'your_personal_information_write_contact_data']
```

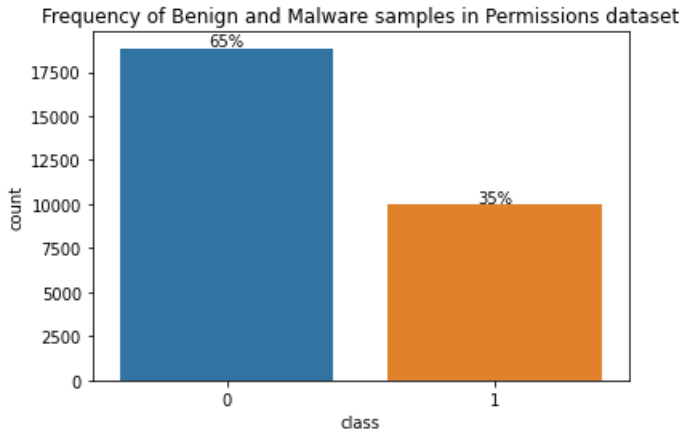
Only the above features are used for further modeling and analysis.

- **Exploratory Analysis**

In this section, I first check the split between malware samples and benign samples in the data. There should be a sufficient number of each type of samples to run data mining algorithms. This is checked as follows.

```
y = data_1_select.loc[:, 'class']
total = float(len(y))
ax = sns.countplot(y)
for p in ax.patches:
    height = p.get_height()
    ax.text(p.get_x()+p.get_width()/2.,
            height + 100,
            '{:1.0f}%'.format(height/total * 100),
            ha="center")
ax.set_title('Frequency of Benign and Malware samples in Permissions dataset')
```

Figure 8: Class Distribution of Permissions Data



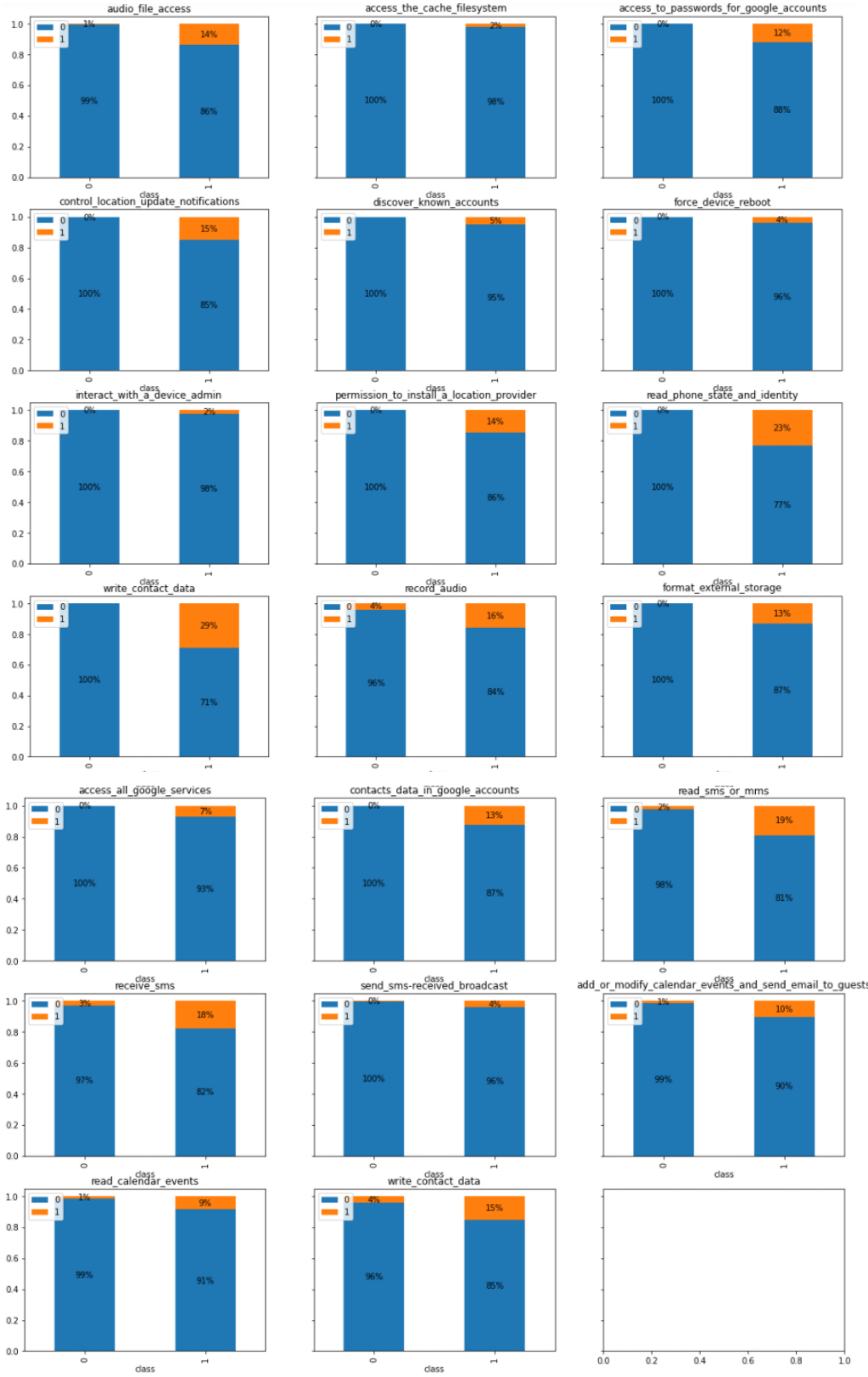
Therefore, 65% of the data is in the 'Benign' class, and 35% is in the 'Malware' class. There are sufficient samples in each class to proceed with the analysis.

Now, I look at the split of 'Yes' and 'No' for each feature when broken down by Benign and Malware samples. This will help visually identify features that most differentiate malware and benign apps.

```
classes = ['benign', 'malware']
titles = [x.replace('default_','').replace('hardware_controls_','').replace('network_communication_','').
          replace('phone_calls_','').replace('services_that_cost_you_money_','').
          replace('system_tools_','').replace('your_accounts_','').
          replace('your_location_','').
          replace('your_messages_','').
          replace('your_personal_information_','') for x in select_cols]
titles = [x if 'storage_modify' not in x else 'storage_modify' for x in titles]
fig, axs = plt.subplots(int(np.ceil(len(select_cols)/3)), 3, sharey=True, figsize=(18,30))

for i,ax in enumerate(axs.flat):

    try:
        plt_data = data_1_select.groupby('class')[select_cols[i]].value_counts(normalize=True).unstack()
    except IndexError:
        continue
    # perc_0 = np.array(plt_data[0])

    # ax.bar(classes, np.array((1,1)), label='Yes')
    # ax.bar(classes, perc_0, label='No')

    plt_data.plot(kind='bar', stacked=True, ax=ax)

    ax.set_title(titles[i])
    ax.legend(loc='upper left')

    for c in ax.containers:
        labels = ['{}%'.format(int(round(v.get_height()*100,0))) if v.get_height() > 0 else '' for v in c]
        ax.bar_label(c, labels=labels, label_type='center')

plt.show()
```

The results are shown below.

Figure 9: Permissions Data Exploratory Analysis Results



From the above plots, it can be observed that in terms of absolute difference in percentages, the features *'read_phone_state_and_identity'* and *'write_contact_data'* differentiate between malware and benign samples the most.

This means that 23% of malware apps request *'read_phone_state_and_identity'* permission and 0% of benign apps do so. Therefore, if an app requests this permission, we can predict that it is malware.

In subsequent sections, when we do predictive modeling, we can expect that these features with high differences will be the key drivers of prediction.

3.2.1.2 Signatures Data

For the predictive modeling, the signatures data was used to test the effectiveness of the signatures-based method. The original dataset contained hundreds of different types of features. For this project, only the API signatures related features were selected. Similar to the permissions data exploration, I first read this dataset and then conducted the descriptive analysis as well as applied feature selection methods.

```
data_2 = pd.read_csv('dataset_2.csv')
```

Data Description

Let's see how many rows and columns the signatures dataset has.

```
print('The signatures data has {} row and {} columns'.format(data_2.shape[0], data_2.shape[1]))
```

```
The signatures data has 3799 row and 73 columns
```

The first few rows of the data are shown below –

```
pd.options.display.max_columns = 8
data_2.head()
```

	transact	onServiceConnected	bindService	attachInterface	...	Ljava.lang.Class.getResource	defineClass	findClass	class
0	0	0	0	0	...	1	0	0	S
1	0	0	0	0	...	0	0	0	S
2	0	0	0	0	...	0	0	0	S
3	0	0	0	0	...	0	0	0	S
4	1	1	1	1	...	1	0	0	S

5 rows × 73 columns

Therefore, the dataset contains 73 columns, of which 72 are API signature related, whereas 1 is the class variable that indicates whether the record is malware or benign.

API signatures are basically parts of the source code of the Android application. These signatures for a particular app will contain all the possible operations and interactions of the app with the Android phone. Each column in this dataset represents one such function in the app source code.

As can be seen in the first few rows above, all the signature-related variables are binary. 1 indicates that the source code of the app contains that signature function, and 0 indicates that it does not have that signature function.

Similar to earlier, some names of signature columns are shown below to get a sense of the type of features available.

Table 2: Examples of Signatures related features

Examples of signatures-related features
transact
android.content.pm.PackageInfo
android.telephony.SmsManager
TelephonyManager.getSimSerialNumber
HttpPost.init
Ljava.lang.Class.getCanonicalName
System.loadLibrary
Ljava.net.URLDecoder
sendMultipartTextMessage

As these column names show, these signatures are part of the android app code and are used to perform specific operations. There could be some signatures that are not common in benign apps but are common in malware apps. Thus, these signatures provide a way of detecting whether an app is malware or benign.

Data Cleaning

- **Missing Values Check:**

  Like with the permissions data, the number of missing values in the dataset is checked as follows:

```
print('The maximum number of missing values in any column is: {}'.format(data_2.isnull().sum().max()))
```

```
The maximum number of missing values in any column is: 0
```

  Therefore, there are no missing values, and the data appears to be clean. No action is needed.

- **Type Conversion:**

  Like earlier, let's convert the **'class'** variable into a binary variable instead of a categorical variable. This will help later in performing the classification.

  Unlike the permissions data, the class variable in the signatures data contains categories 'S,' and 'B.' 'S' stands for Suspect and 'B' stands for Benign. So, I convert them to binary accordingly.

```
data_2['class'] = data_2['class'].map({'S': 1, 'B': 0})
```

- **Feature Selection**

As seen above, the data has 72 features. The aim of this section is to reduce this number to 20 features like in the permissions data analysis.

I first calculate the value counts of each column in the dataset to see which the top 10 most frequently occurring features are for malware apps.

```
freq_cols = data_2.loc[:,[x for x in data_2.columns.tolist() if x not in ['class']]].apply(
    lambda col: col.value_counts(normalize=True)).transpose()

freq_cols.sort_values(1, ascending=False).head(10)
```

	0	1
Binder	0.108450	0.891550
IBinder	0.114767	0.885233
android.os.IBinder	0.114767	0.885233
Ljava.lang.Object.getClass	0.182153	0.817847
android.content.pm.PackageInfo	0.194525	0.805475
onBind	0.214793	0.785207
HttpUriRequest	0.382996	0.617004
TelephonyManager.getDeviceId	0.385101	0.614899
Ljava.lang.Class.forName	0.396420	0.603580
Ljava.lang.Class.getMethod	0.414846	0.585154

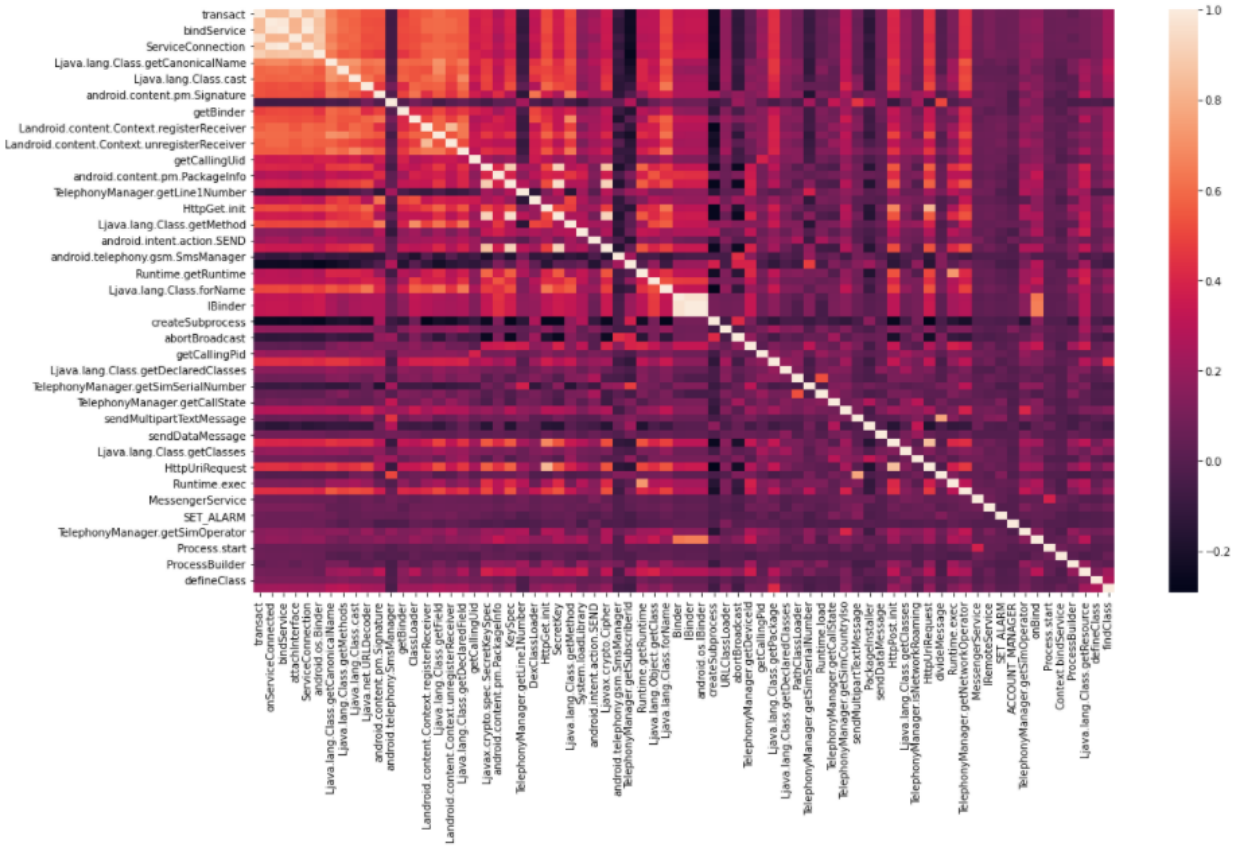
Correlation

It can be observed above that some of the features are similarly named, for example, *'Binder'* and *'IBinder*.' We can look for correlations in the data to see if any features that are highly correlated to others can be removed.

Below is a correlation heatmap of all the features in the dataset. Lighter colors in the grid indicate a higher correlation.

```
select_cols = [x for x in data_2.columns if x != 'class']
cor = data_2.loc[:, select_cols].corr()
plt.figure(figsize=(18,10))
sns.heatmap(cor, annot=False)
```

Figure 10: Signatures Data Correlation Heatmap



It can be observed above that there are many variables that are highly correlated to other variables.

For this project, a threshold of 0.8 is set up in order to represent high correlation and remove all variables that have a correlation coefficient with any other variables greater than 0.8. This is done below.

```
rmv_cols = []
for c in select_cols:
    if c not in rmv_cols:
        corr_cols = cor.loc[cor[c] >= 0.8,c].index.tolist()
        corr_cols = [x for x in corr_cols if x != c]
        if len(corr_cols) > 0:
            rmv_cols += corr_cols

print('Using correlation, {} columns are removed'.format(len(rmv_cols)))
```

```
Using correlation, 13 columns are removed
```

After the execution of the above chunk, the dataset features have been reduced from 72 to 59.

```
select_cols = [x for x in select_cols if x not in rmv_cols]
data_2_select = data_2.loc[:, select_cols + ['class']]
```

Chi-Square Test

The chi-square test for independence can be applied to choose the best 20 features in the signatures dataset as well as presented in the following chunk.

```
chi2_features = SelectKBest(chi2, k=20)
X_kbest = chi2_features.fit_transform(data_2_select.loc[:,select_cols], data_2_select['class'])
```

The top 20 features chosen through this process are as follows.

```
select_cols = [data_2_select.columns.tolist()[i] for i in chi2_features.get_support(indices=True)]
select_cols
```

```
['transact',
 'Ljava.lang.Class.getCanonicalName',
 'Ljava.lang.Class.getMethods',
 'Ljava.lang.Class.cast',
 'Ljava.net.URLDecoder',
 'getBinder',
 'Landroid.content.Context.registerReceiver',
 'Ljava.lang.Class.getField',
 'Ljava.lang.Class.getDeclaredField',
 'getCallingUid',
 'TelephonyManager.getLine1Number',
 'HttpGet.init',
 'android.telephony.gsm.SmsManager',
 'TelephonyManager.getSubscriberId',
 'createSubprocess',
 'abortBroadcast',
 'TelephonyManager.getDeviceId',
 'Ljava.lang.Class.getPackage',
 'TelephonyManager.getSimSerialNumber',
 'PackageInstaller']
```

The dataset is now subsetted for only these columns, which will be used for analysis.

```
data_2_select = data_2.loc[:, select_cols + ['class']]
```

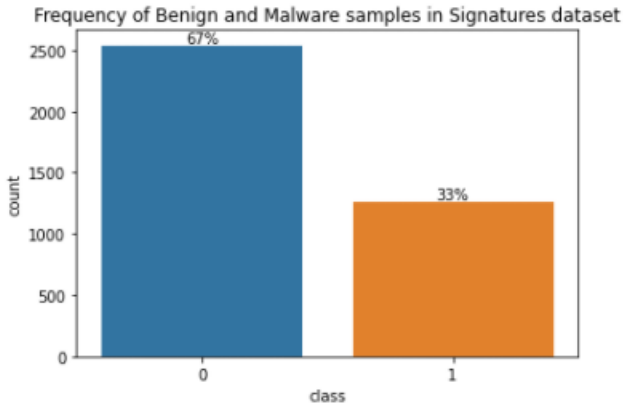
- **Exploratory Analysis**

In this section, exploratory analysis is performed for the permissions dataset.

First, check the split between malware samples and benign samples in the data. It is required a sufficient number of each type of samples to run data mining algorithms.

```
y = data_2_select.loc[:, 'class']
total = float(len(y))
ax = sns.countplot(y)
for p in ax.patches:
    height = p.get_height()
    ax.text(p.get_x()+p.get_width()/2.,
            height + 20,
            '{:1.0f}%'.format(height/total * 100),
            ha="center")
ax.set_title('Frequency of Benign and Malware samples in Signatures dataset')
```

Figure 11: Class Distribution of Signatures Data

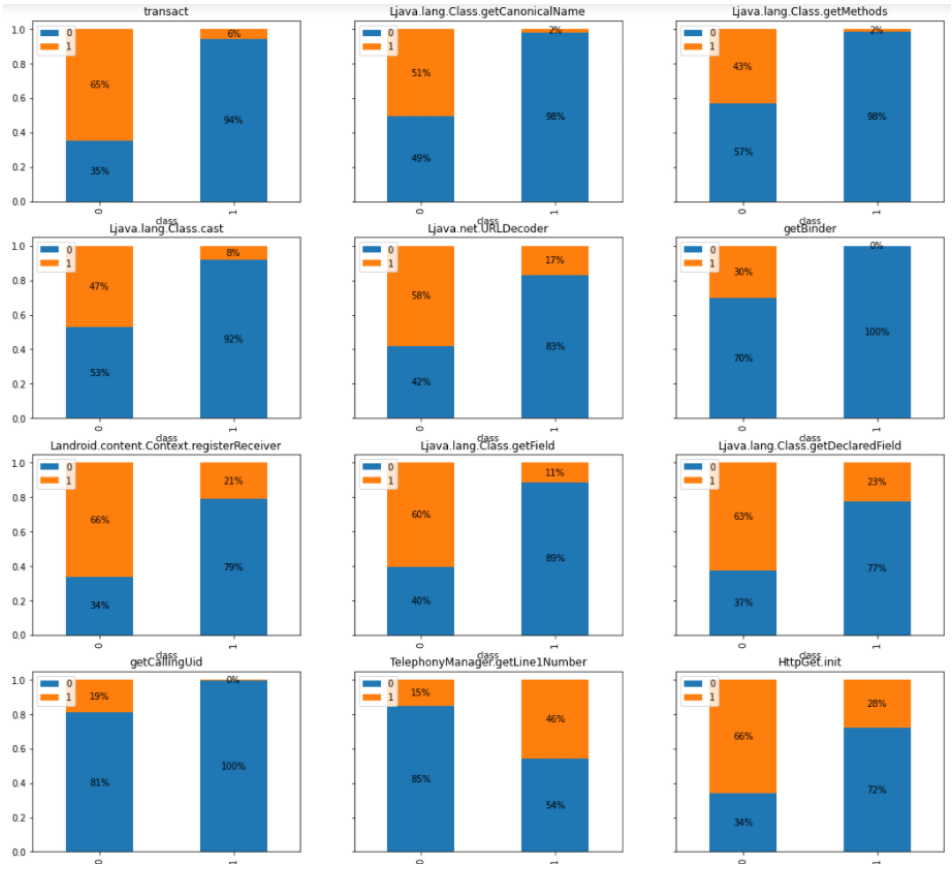


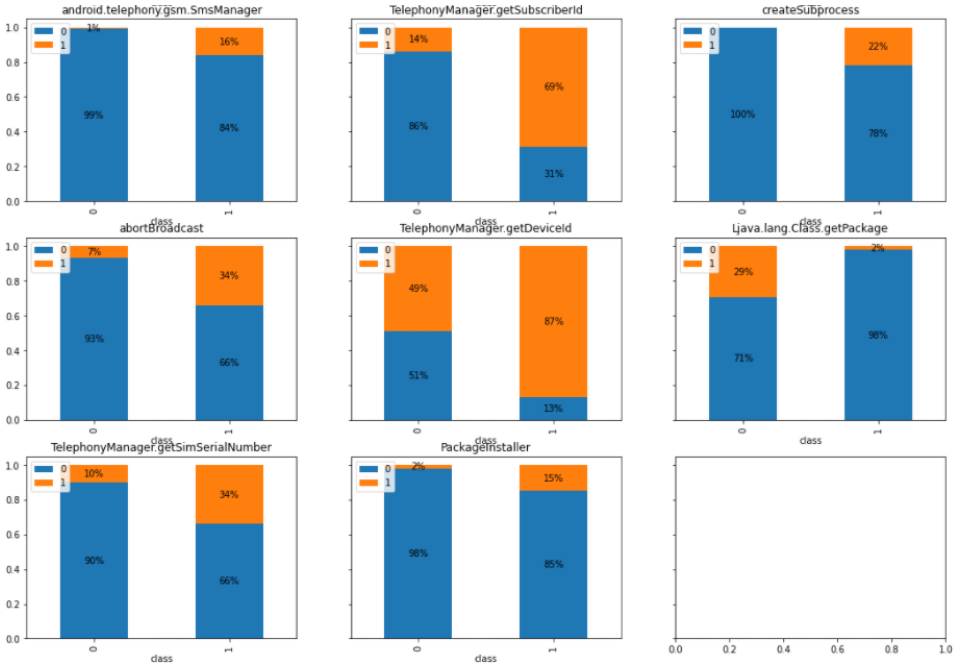
Therefore, 67% of records in the data belong to the 'Benign' class, and 33% belong to the 'Malware' class. There is a sufficient number of records in each class to proceed further.

We can now look at the split of 'Yes' and 'No' for each feature when broken down by Benign and Malware samples. These plots are shown below.

```
classes = ['benign', 'malware']
titles = select_cols
fig, axs = plt.subplots(int(np.ceil(len(select_cols)/3)), 3, sharey=True, figsize=(18,30))

for i,ax in enumerate(axs.flat):

    try:
        plt_data = data_2_select.groupby('class')[select_cols[i]].value_counts(normalize=True).unstack()
    except IndexError:
        continue

    plt_data.plot(kind='bar', stacked=True, ax=ax)

    ax.set_title(titles[i])
    ax.legend(loc='upper left')

    for c in ax.containers:
        labels = ['{}%'.format(int(round(v.get_height()*100,0))) if v.get_height() > 0 else '' for v in c]
        ax.bar_label(c, labels=labels, label_type='center')

    ax.set_title(titles[i])
    ax.legend(loc='upper left')

plt.show()
```

Figure 12: Signatures Data Exploratory Analysis Results





We can visually identify that the features that appear to differentiate Malware samples from Benign samples the most are *'transact*,' *'getCanonicalName*,' *'getSubscriberId*,' etc. These may be important predictors in the next step.

With the above analyses, the exploratory analysis and feature selection phase is concluded. The next step is to build classification models to compare the performance measures within and across the two types of malware detection methods.

3.2.2 Predictive Modelling

3.2.2.1 Data Mining Introduction

The two datasets have been subsetted, and variables that best differentiate malware and benign applications have been identified. The subsetted datasets are used in this Predictive Modelling section.

Data mining techniques can be categorized into three main types -

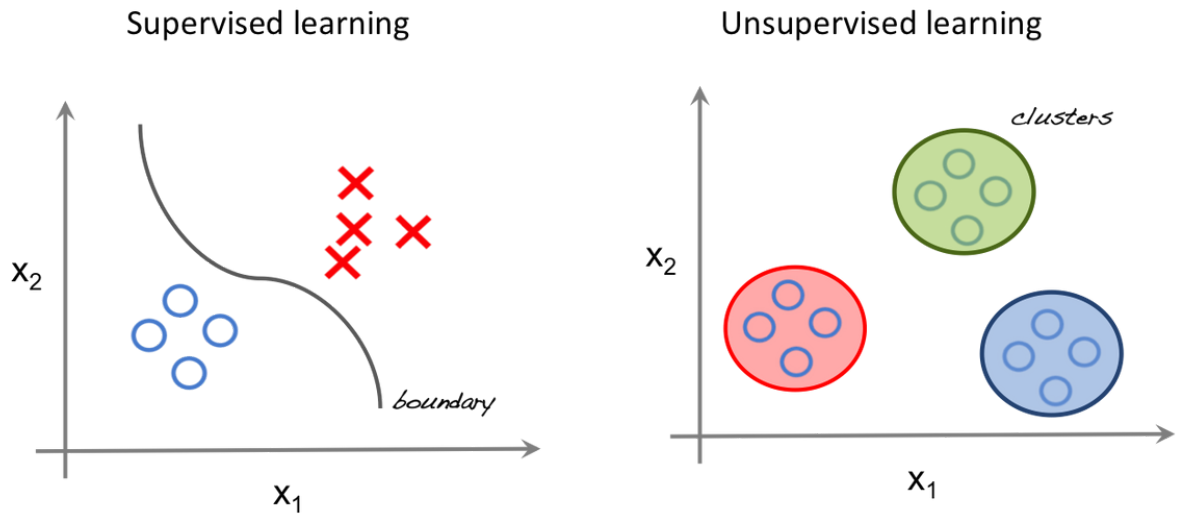
- **Association Rule Learning** - systemically searching for relationships (or associations) between variables
- **Classification** - categorizing data into discrete classes
- **Regression** - mathematically modeling numerical data to predict the next value in the sequence

The objective of this project is Malware Detection. In this problem, we need to categorize the data into 'Malware' and 'Benign' classes. Therefore, the best-suited technique for this purpose is Classification.

Machine Learning methods can be supervised and unsupervised. Supervised classification is used when knowledge of the classes for samples in the training data is known. So, if we already know whether the applications we are training the model with are malware or benign, we can use supervised classification.

Unsupervised classification is used with unlabeled data. Clustering is a typical example of unsupervised classification where we group uncategorized samples into clusters based on their attributes.

Figure 13: Types of Classification



In our case, we have prior knowledge of the classes for samples in the training data. Thus, we will use supervised classification techniques.

There are many different types of supervised classification methods, each with its benefits. Some classification methods are rudimentary, while others are more complex. Usually, the performance results we can observe from different data mining methods will be different. So, we will experiment with three different supervised classification methods to identify the one that gives the best results.

The classification algorithms selected for this study are as follows:

- k-Nearest Neighbour
- Logistic Regression
- Random Forest

The logic behind each of these algorithms is explained below.

- **kNN Algorithm**

The k-nearest neighbor algorithm works by comparing the proximity of a new sample to known samples. Proximity is usually measured using the *Euclidean distance*.

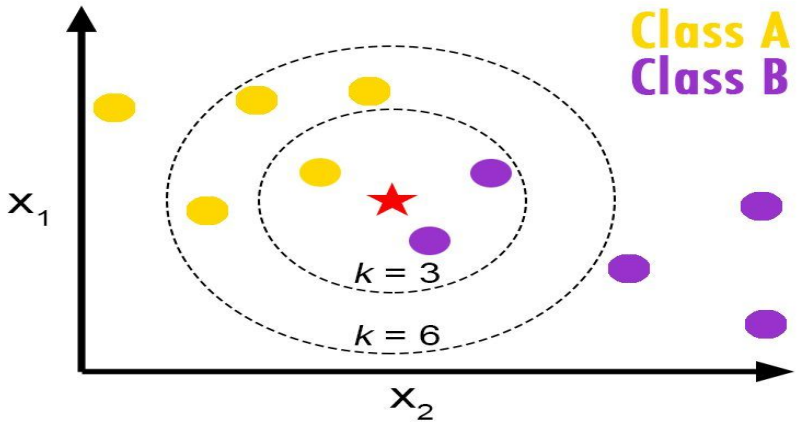
The value of k represents the number of closest known samples considered to determine the class of the new unknown sample. This is an input to the kNN algorithm, and the final result of the classification is dependent on this input value.

First, k samples from the existing known set that is closest to the new unknown sample are identified using a proximity measure. The values of the features of each sample are used to calculate the proximity measure. Once the k nearest neighbors are identified, the split of classes among these k samples is determined. The class that occurs most frequently among these k nearest samples is assigned to the new unknown point.

An example is shown below where the *star* point represents the new unknown sample, and the circles represent existing known samples. The color as shown represents the class of the existing samples.

If we choose a k of 3, two of the three closest neighbors belong to Class B, whereas one of the three closest neighbors belongs to Class A. So, the class assigned to the new point will be *Class B*. If we choose a k of 6, two of the six closest neighbors belong to Class B whereas four of the six closest neighbors belong to Class A. So, class assigned to the new point will be *Class A*.

Figure 14: kNN Algorithm Concept



- **Logistic Regression**

The Logistic Regression technique works by using a special function called the *'Logistic (or Sigmoid) function'* to model the binary dependent variable.

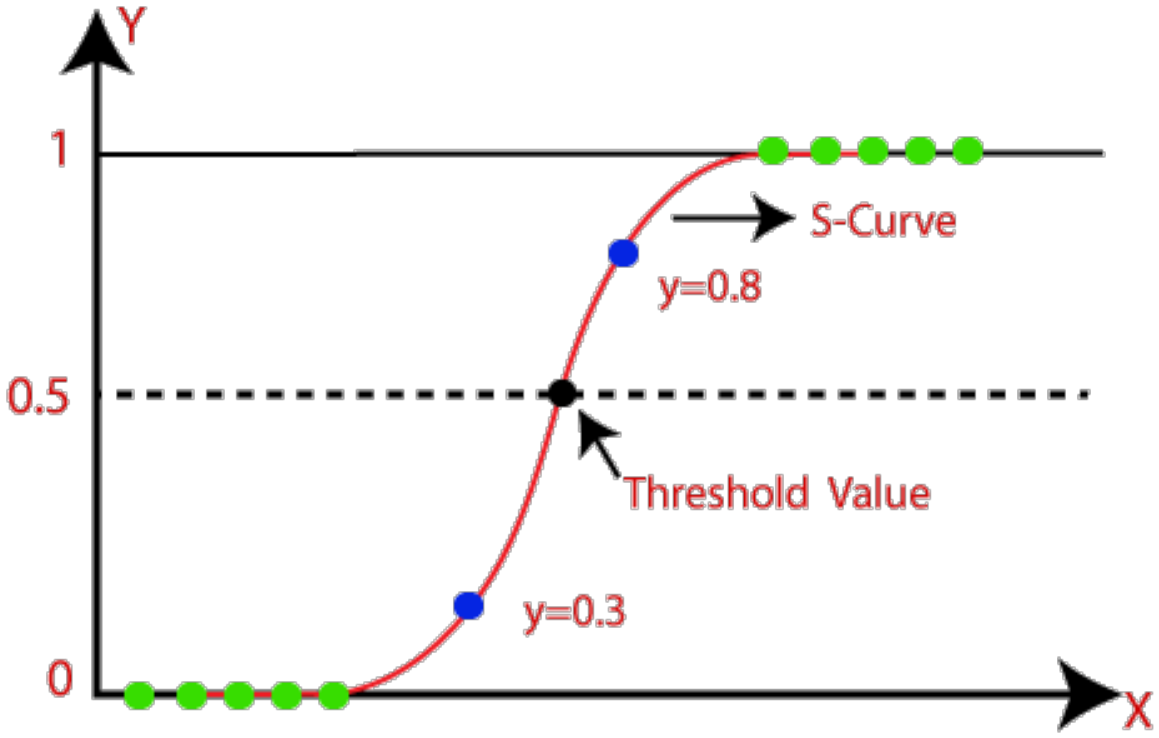
It essentially models the log-odds of the binary dependent variable using a linear regression model. The log odds are then converted to probabilities which vary from 0 to 1 for each class. Depending on the value of this predicted probability, classification can be made.

For example, if the predicted probability is greater than a certain threshold (usually 0.5), the sample is classified as Class A, and if it is less than the threshold, the sample is classified as Class B.

The below diagram demonstrates how this classification is done using the Sigmoid curve.

Probability is shown on the Y-axis, and the threshold value is highlighted. Depending on whether the new sample lies above or below this threshold value, classification is done.

Figure 15: Logistic Regression Concept



The equation of the logistic curve is shown below. This equation is solved similarly to a linear regression model.

Here,

$\mathbf{X_i}$ features are used to make the prediction

$\boldsymbol{\beta_i}$ are the coefficients that are obtained by solving the equation

P is the probability of the sample belonging to the positive class

The left-hand side term is typically called the 'Log-likelihood.'

$$ln\left(\frac{P}{1-P}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_k X_k$$

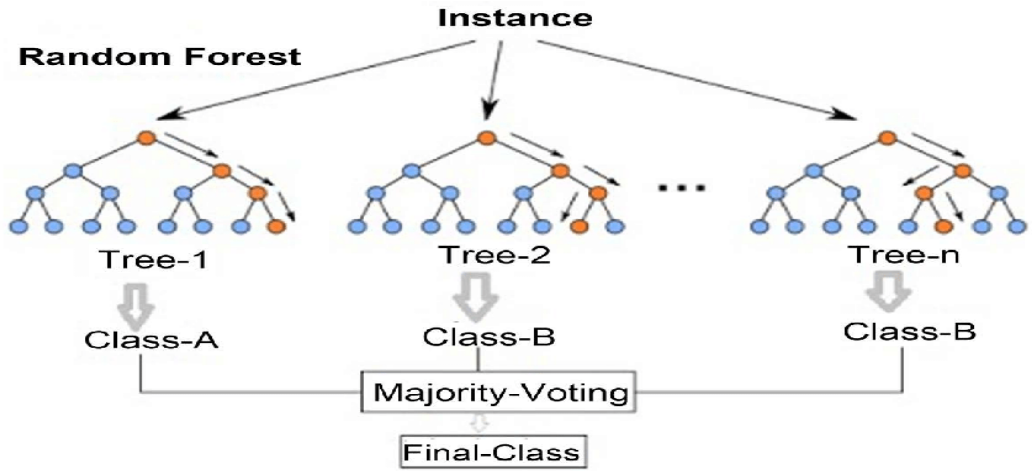
- **Random Forest**

The third model used in the project is the **Random Forest** algorithm.

Random Forest is an extension of the decision tree algorithm. It works by building many decision trees using a random subset of samples and a random subset of features for each tree. This process of selecting only a subset of the samples and features is called *bagging*.

An example is illustrated below. Individual decision trees are constructed, each with a different random subset of samples and features. Then, the output classes from each of the individual decision trees are considered, and the majority class is allocated as the final class to the unknown sample.

Figure 16: Random Forest Concept



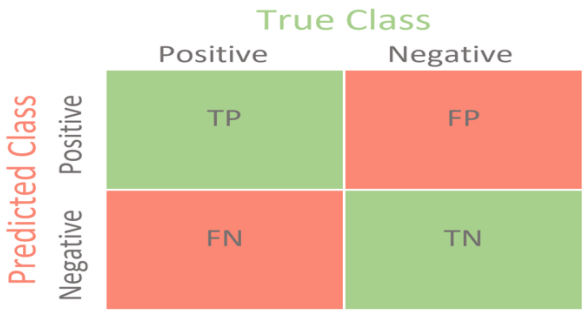
- **Performance Measurement**

It is essential to visualize the confusion matrix and calculate specific performance metrics to evaluate any data mining classification algorithm. These will help in analyzing the performance of each method and comparing the performance of different methods.

Confusion Matrix

Confusion Matrix is the tabular representation of the actual classes vs. predicted classes. It indicates the number of samples in each quadrant. It helps in understanding the True Positives, False Positives, True Negatives, and False Negatives predicted by the model. Thus, it helps in assessing how best the model has performed the classification. A confusion matrix is shown below.

Figure 17: Confusion Matrix Illustration



Whereas:

TP - True Positive: Refers to the positive tuples that were correctly labeled by the classifier.

FP - False Positive: Refers to the positive tuples that were incorrectly labeled by the classifier.

FN - False Negative: Refers to the negative tuples that were incorrectly labeled by the classifier.

TN - True Negative: Refers to the negative tuples that were correctly labeled by the classifier.

Performance Metrics

In addition to the Confusion Matrix, I calculate the following performance metrics and compare these across models to determine which model works best.

- **Precision:**

  Precision measures how frequently a sample app is malware when the model predicts that it is malware. In other words, it is the *True Positive Rate* of the model. It is calculated by dividing the True Positives by the total number of Positive predictions.

$$Precision = \frac{TP}{TP + FP}$$

- **Recall:**

  Recall measures how much of the actual malware in the data is correctly predicted as malware by the model. This measure helps in ensuring that the model accurately detects as much malware as possible to prevent harm. It is calculated by dividing the True Positive by the total number of actual Positive samples.

$$Recall = \frac{TP}{P}$$

- **F1-score:**

  The F1 score is the harmonic mean of the precision and the recall. It is a measure of the overall accuracy of the classification model.

  The highest possible F1 score is 100, and the lowest possible is 0. Higher the F1 score, the better the accuracy of the model.

$$F1\ score = 2\frac{precision * recall}{precision + recall}$$

The below function written in python will be used to visualize the confusion matrix and calculate performance metrics in subsequent sections.

```
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=0)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

    precision = round(100 * cm[1,1] / (cm[0,1] + cm[1,1]), 2)
    recall = round(100 * cm[1,1] / (cm[1,0] + cm[1,1]), 2)

    print()
    print('Precision: ' + str(precision) + '%')
    print('Recall: ' + str(recall) + '%')
    print('F1 Score: ' + str(round(2*((precision*recall)/(precision+recall)),2)))
```

Now that we understand how each of the data mining techniques works and how to use the performance metrics to decide which technique is better, we can proceed with applying these to the malware detection datasets.

In the following sections, the results of each type of classifier are described first on the permissions dataset and then on the signatures dataset.

3.2.2.2 Permissions-based Approach

First, read the cleaned dataset that was generated in the Descriptive Analysis section.

```
data_1_select = pd.read_csv('dataset_1_select.csv')
```

It is necessary to separate the feature variables and class variables. The feature variables are the independent variables that will help with the prediction, and the class variable is the dependent variable that indicates whether the sample is malware or benign.

```
X = data_1_select.loc[:,data_1_select.columns[:-1]]
y = data_1_select.iloc[:,-1].reset_index(drop=True).astype(int)
```

Next, I split the whole dataset into train and test sets. The train set will be used to train the data mining models, and the test set will be used to measure their performance. I randomly select **70%** of the samples as the train set and the remaining **30%** as the test set.

```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.3, random_state = 42)
```

Now, let's initialize the models.

For the purpose of model evaluation and to compare models, I use the *recall* score. This is because it is crucial for a malware detector to detect as much malware as possible to minimize harm accurately.

I also do a 5-fold cross-validation that will split the train set into five equal parts and run the model 5 times, each time considering one of the five parts as the validation set. This way, I calculate the average recall score, which is a more robust value than if the model is run only once.

```
scr = 'recall' # Recall score used for model evaluation
model_lr = LogisticRegression(solver='liblinear')
model_kn = KNeighborsClassifier(n_neighbors=3)
model_rf = RandomForestClassifier()
skf = StratifiedKFold(5) # 5 fold stratified cross validation
```

kNN Classifier

Let's start with the results of the kNN classifier. Select k=3 and run the model.

```
# kNN with cross validation to evaluate the average performance score with train dataset
sc_kn = cross_val_score(model_kn, X_train, y_train, cv=skf, scoring=scr)
print("kNN's average recall across validation sets is: " + str(round(sc_kn.mean() * 100, 2)) + '%')
```

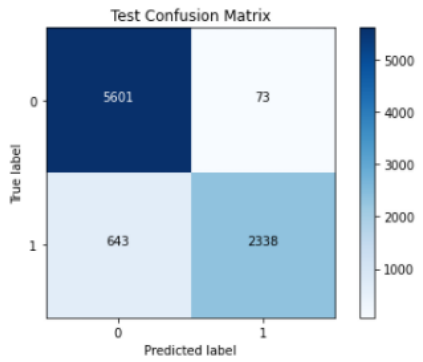
```
kNN's average recall across validation sets is: 78.9%
```

The confusion matrix and performance metrics for the test set can be viewed as follows –

Test Data:

```
# Make predictions on the test dataset using logistic regression and visualize the confusion matrix
y_pred_test_kn = kn.predict(X_test)
cnf_matrix_test_kn = confusion_matrix(y_test, y_pred_test_kn)
class_names = ['0','1']
plot_confusion_matrix(cnf_matrix_test_kn, classes = class_names, title = 'Test Confusion Matrix')
```

```
Precision: 96.97%
Recall: 78.43%
F1 Score: 86.72
```



Results

Recall of ~78% indicates that the model is able to correctly capture 78% of all the malware in the testing set.

Now, we should see if the other models give better performance metrics than the kNN algorithm.

Logistic Regression Classifier

The Logistic Regression performance is calculated for the testing dataset.

```
# Logistic Regression with cross validation to evaluate the average performance score with train dataset
sc_lr = cross_val_score(model_lr, X_train, y_train, cv=skf, scoring=scr)
print("Logistic Regression's average recall across validation sets is: " + str(round(sc_lr.mean() * 100, 2)) + '%')
```

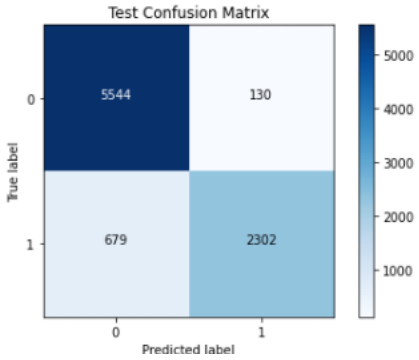
```
Logistic Regression's average recall across validation sets is: 77.71%
```

The confusion matrix and performance metrics for the test set are as follows –

Test Data:

```
# Make predictions on the test dataset using logistic regression and visualize the confusion matrix
y_pred_test_lr = lr.predict(X_test)
cnf_matrix_test_lr = confusion_matrix(y_test, y_pred_test_lr)
class_names = ['0','1']
plot_confusion_matrix(cnf_matrix_test_lr, classes = class_names, title = 'Test Confusion Matrix')
```

```
Precision: 94.65%
Recall: 77.22%
F1 Score: 85.05
```



Logistic Regression has a recall score of ~77%, which is not an improvement over the 78% we saw with kNN. Therefore, we can say that Logistic Regression is not the best model.

Random Forest Classifier

The Random Forest model is selected as the classifier.

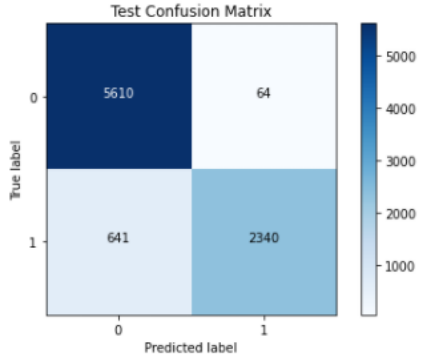
```
# Random Forest with cross validation to evaluate the average performance score with train dataset
sc_rf = cross_val_score(model_rf, X_train, y_train, cv=skf, scoring=scr)
print("Random Forest's average recall across validation sets is: " + str(round(sc_rf.mean() * 100, 2)) + '%')
```

```
Random Forest's average recall across validation sets is: 79.22%
```

Test Data:

```
# Make predictions on the test dataset using Random Forest and visualize the confusion matrix
y_pred_test_rf = rf.predict(X_test)
cnf_matrix_test_rf = confusion_matrix(y_test, y_pred_test_rf)
class_names = ['0','1']
plot_confusion_matrix(cnf_matrix_test_rf, classes = class_names, title = 'Test Confusion Matrix')
```

```
Precision: 97.34%
Recall: 78.5%
F1 Score: 86.91
```



Therefore, the Random Forest model has a recall score of ~79%, which is slightly higher than the recall score of the kNN model.

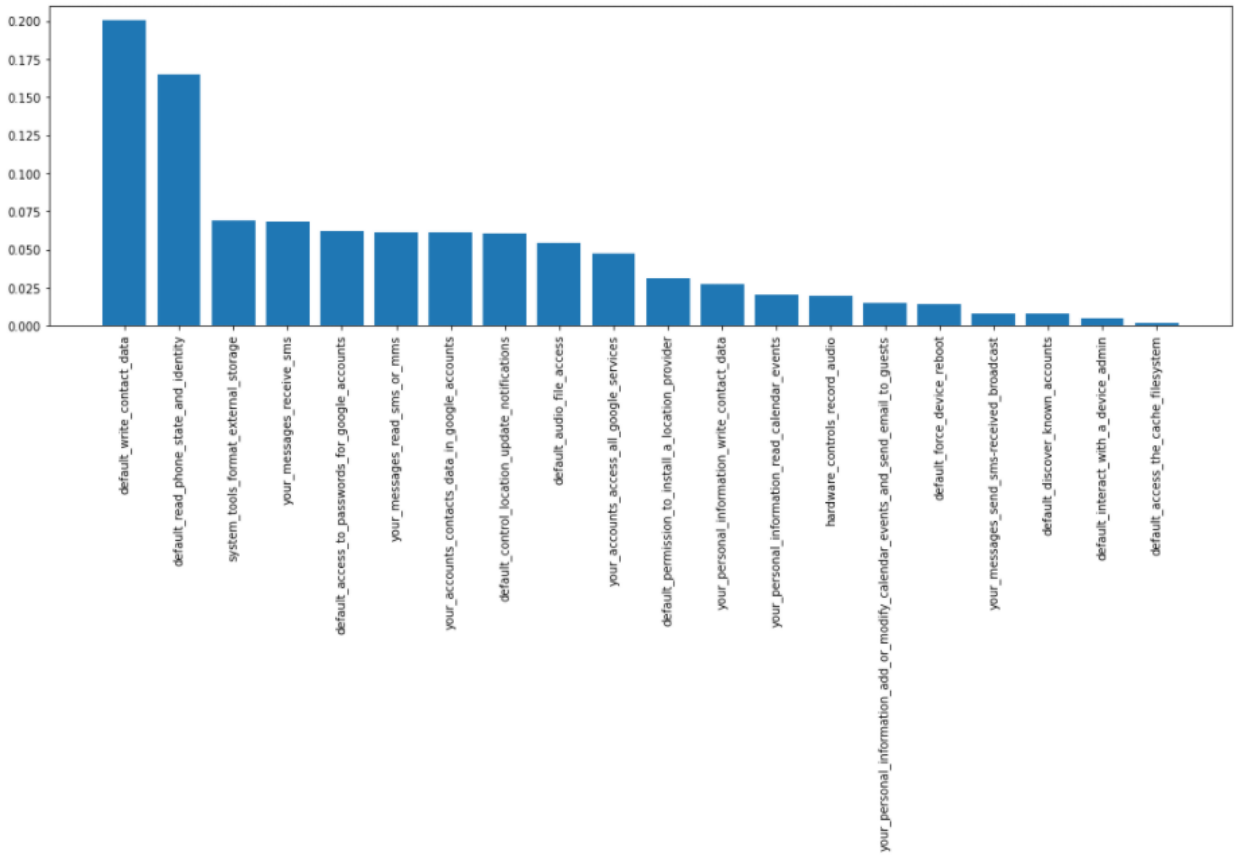
There is no single model that is best in the case of the Permissions dataset. All perform similarly, giving a recall score of ~78-79%. However, the Random Forest seems to be only slightly better than the others, so it is recommended the Random Forest model for the purpose of malware detection with permissions data.

Feature Importance

Let's now see which of the features used in the permissions dataset are the best predictors of malware. I plot the importance of features below in decreasing order; the taller the bars, the more important the feature for the final prediction.

```
importance = rf.feature_importances_

X_vars = X_train.columns.tolist()

imp_dict = {}
# summarize feature importance
for i,v in enumerate(importance):
    print('Feature: {}, Score: %.5f'.format(X_vars[i]) % (v))
    imp_dict[X_vars[i]] = v

imp_dict = {k:v for k,v in sorted(imp_dict.items(), key=lambda item: item[1], reverse=True)}

# plot feature importance
f, ax = plt.subplots(figsize=(18,5))
plt.bar(imp_dict.keys(), imp_dict.values())
plt.xticks(rotation=90)
plt.show()
```

Figure 18: Feature Importance of Permissions Data Model



Therefore, it can be seen that the two most important features are - **Permission to write contact data** and **Permission to read phone state and identity**.

This is an interesting finding and makes intuitive sense. Not many apps will need to write contact data or read the phone's identity. If an app is requesting these permissions, the likelihood that the app is malware is higher.

In summary, the best model for the Permissions-based approach is Random Forest, with a recall score of ~79%. I now move on to the Signatures-based approach and see if a different set of features helps give better results.

3.2.2.3 Signatures-based Approach

Similar to the permissions-based approach, the cleaned dataset is loaded that was generated in the descriptive analysis section.

```
data_2_select = pd.read_csv('dataset_2_select.csv')
```

I now separate the X and y variables and split the dataset into train and test sets. I follow the same 70-30 split as in the permissions approach.

```
X = data_2_select.loc[:,data_2_select.columns[:-1]]
y = data_2_select.iloc[:,-1].reset_index(drop=True).astype(int)
```

```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.3, random_state = 42)
```

Let's initialize the same models as earlier.

```
scr = 'recall' # Recall score used for model evaluation
model_lr = LogisticRegression(solver='liblinear')
model_kn = KNeighborsClassifier(n_neighbors=3)
model_rf = RandomForestClassifier()
skf = StratifiedKFold(5) # 5 fold stratified cross validation
```

kNN Classifier

Let's see how the kNN classifier performs with the signatures dataset.

```
# kNN with cross validation to evaluate the average performance score with train dataset
sc_kn = cross_val_score(model_kn, X_train, y_train, cv=skf, scoring=scr)
print("kNN's average recall across validation sets is: " + str(round(sc_kn.mean() * 100, 2)) + '%')
```

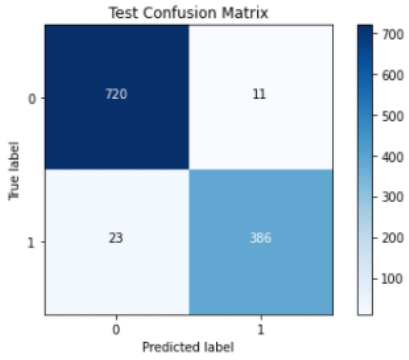
```
kNN's average recall across validation sets is: 94.01%
```

The confusion matrix and performance metrics of the test datasets are as follows:

Test Data:

```
# Make predictions on the test dataset using logistic regression and visualize the confusion matrix
y_pred_test_kn = kn.predict(X_test)
cnf_matrix_test_kn = confusion_matrix(y_test, y_pred_test_kn)
class_names = ['0','1']
plot_confusion_matrix(cnf_matrix_test_kn, classes = class_names, title = 'Test Confusion Matrix')
```

```
Precision: 97.23%
Recall: 94.38%
F1 Score: 95.78
```



Therefore, the kNN classifier with a k=3 gives a recall score of ~94% with signatures relate features. This is a significant improvement from the permissions-related features.

Let's now run similar analyses with the other models.

Logistic Regression Classifier

The average performance of the Logistic Regression Classifier across the cross-validation sets is as follows –

```
# Logistic Regression with cross validation to evaluate the average performance score with train dataset
sc_lr = cross_val_score(model_lr, X_train, y_train, cv=skf, scoring=scr)
print("Logistic Regression's average recall across validation sets is: " + str(round(sc_lr.mean() * 100, 2)) + '%')
```

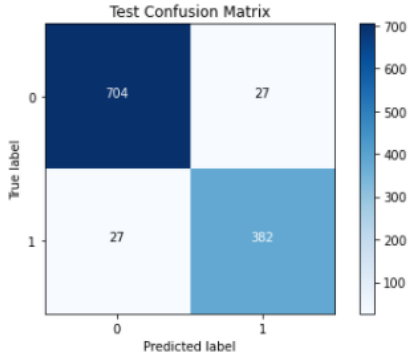
```
Logistic Regression's average recall across validation sets is: 92.6%
```

The confusion matrix and performance metrics of the test datasets are shown below –

Test Data:

```
# Make predictions on the test dataset using logistic regression and visualize the confusion matrix
y_pred_test_lr = lr.predict(X_test)
cnf_matrix_test_lr = confusion_matrix(y_test, y_pred_test_lr)
class_names = ['0','1']
plot_confusion_matrix(cnf_matrix_test_lr, classes = class_names, title = 'Test Confusion Matrix')
```

```
Precision: 93.4%
Recall: 93.4%
F1 Score: 93.4
```



Logistic Regression gives a recall score of ~93%, which is comparable to what we saw with the kNN algorithm.

Random Forest Classifier

In this section, the Random Forest model is selected as the classifier.

```
# Random Forest with cross validation to evaluate the average performance score with train dataset
sc_rf = cross_val_score(model_rf, X_train, y_train, cv=skf, scoring=scr)
print("Random Forest's average recall across validation sets is: " + str(round(sc_rf.mean() * 100, 2)) + '%')
```

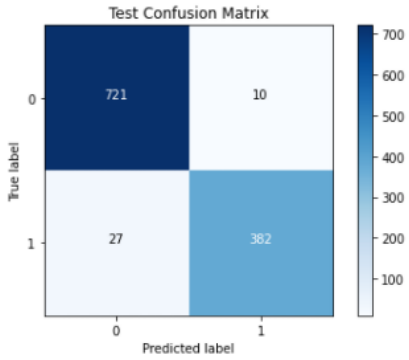
```
Random Forest's average recall across validation sets is: 93.9%
```

The confusion matrix and performance metrics of the testing dataset are as follows –

Test Data:

```
# Make predictions on the test dataset using Random Forest and visualize the confusion matrix
y_pred_test_rf = rf.predict(X_test)
cnf_matrix_test_rf = confusion_matrix(y_test, y_pred_test_rf)
class_names = ['0','1']
plot_confusion_matrix(cnf_matrix_test_rf, classes = class_names, title = 'Test Confusion Matrix')
```

```
Precision: 97.45%
Recall: 93.4%
F1 Score: 95.38
```



Random Forest also gives a recall score of ~93%. We further observe that the F1 scores of the three models are also similar.

Therefore, there is no single algorithm that performs much better than the others for the application of malware detection using signatures-based features. We can use either of the three we experimented with above.

For the purpose of comparison, I proceed with choosing the Random Forest model as with the permissions approach.

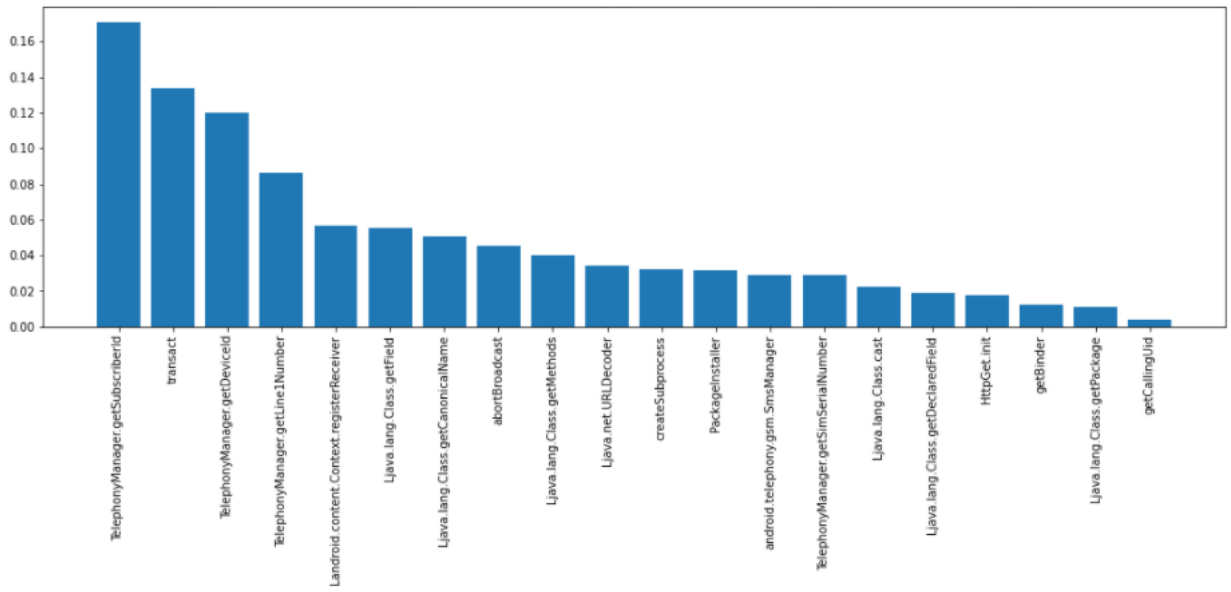
Feature Importance

Similar to the permissions-based approach, let's now view which of the signatures-based features are most important.

A similar plot as earlier is shown below.

```
importance = rf.feature_importances_

X_vars = X_train.columns.tolist()

imp_dict = {}
# summarize feature importance
for i,v in enumerate(importance):
    print('Feature: {}, Score: %.5f'.format(X_vars[i]) % (v))
    imp_dict[X_vars[i]] = v

imp_dict = {k:v for k,v in sorted(imp_dict.items(), key=lambda item: item[1], reverse=True)}

# plot feature importance
f, ax = plt.subplots(figsize=(18,5))
plt.bar(imp_dict.keys(), imp_dict.values())
plt.xticks(rotation=90)
plt.show()
```

Figure 19: Feature Importance of Signatures Data Model



Therefore, the three most important features when it comes to API signatures built into the app are – **Get Subscriber ID**, **Transact** and **Get Device ID**

These are similar to the most important features we noted with the permissions approach. If the app has built-in functionality to retrieve identification information about the device and the functionality to call the Transact API, then it is usually not a benign app and is more likely to be malware.

With the above results, we can conclude that the signatures-based approach (recall=93%) is much better than the permissions-based approach (recall=79%) for detecting malware.

There is no single model that performs much better than the other models. We could use the Random Forest and the kNN algorithms with equal impact.

3.2.3 Comparison of Results

To summarize, the performances of all models tested in both approaches are shown below –

Table 3: Summary of Results

		kNN Classifier	Logistic Regression	Random Forest
Permissions-based	Precision	96.97%	94.65%	97.34%
	Recall	78.43%	77.22%	78.5%
	F1-Score	86.72	85.05	86.91
Signatures-based	Precision	97.23%	93.4%	97.45%
	Recall	94.38%	93.4%	93.4%
	F1-Score	95.78	93.4	95.38

The best models for each approach are highlighted above.

As we can see by comparing the numbers, the best model for the permissions-based approach is Random Forest, and the best model for the signatures-based approach is kNN Classifier. However, for all practical purposes, we can assume that both Random Forest and kNN Classifier perform similarly.

Further, if we compare the two approaches, it can be seen that the signatures-based approach gives much better recall than the permissions-based approach indicating that using signatures-based features is better for malware detection.

Chapter 4

4.1 Conclusion

In this project, my goal was to use data mining techniques to solve the problem of detecting malware applications on Android phones. I solved this problem using two different approaches and, through comparison, identified the best approach. I experimented with three different supervised classification techniques and identified the best technique for each approach.

During the study, the signatures-based approach provides high-quality results. The signatures-based approach uses features related to API signatures available in the source code of the application. Codes of different applications are written differently. Thus there are markers in the code that will help in differentiating malware applications from benign applications.

In addition, Random Forest and kNN Classifier provide the best results in both approaches. Precision and Recall scores of 95%-97% were achieved in the project. This means that we can identify 95%-97% of malware through this technique.

In conclusion, I successfully developed a framework for detecting Android-related malware, which is a major concern for law enforcement agencies around the world. This type of framework will help cyber security departments of law enforcement agencies in Dubai and across the world to swiftly identify malware that could be potentially harmful to mobile users.

4.2 Recommendations

As a direct result of the work done on this project, the following recommendations are:

- Select the signatures-based data approach when attempting to detect malware
- Use kNN or Random Forest algorithms trained using data mentioned in this project to perform the classification.

The following research is required to improve the project further:

- Explore alternative approaches in addition to permissions and signatures described in the project
- Explore the possibility of expanding the types of malware that can be detected by combining both permissions and signatures datasets
- Experiment with more advanced classification techniques such as Neural Networks to check if they improve the results further.

Additionally, to create a practical implementation of the work done in this project, an Android background application can be developed that will automatically scan any newly installed app and detect whether it is malware or benign. This app would run in the background and provide security to the mobile phone, just like how antivirus software offers security to a computer.

References

Almin, S. B., & Chatterjee, M. (2015). A novel approach to detect Android malware. *Procedia Computer Science*, *45*, 407-417.

An Odusami et al. (2018, November). Android malware detection: A survey. In *International conference on applied informatics* (pp. 255-266). Springer, Cham.

Arshad et al. (2016). Android malware detection & protection: a survey. *International Journal of Advanced Computer Science and Applications*, *7*(2), 463-475.

Assisi, A., Abhijith, A., Babu, A., & Nair, A. M. Significant permission identification for machine learning based android malware detection: a review.

Atlas. (n.d.). Malware statistics. Retrieved Oct, 2021 from https://portal.av-atlas.org/malware/statistics

Christiana, A., Gyunka, B., & Noah, A. (2020). Android Malware Detection through Machine Learning Techniques: A Review.

Daoudi et al. (2021, August). Dexray: A simple, yet effective deep learning approach to android malware detection based on image representation of bytecode. In *International Workshop on Deployable Machine Learning for Security Defense*, 81-106.

Fallah, S., & Bidgoly, A. J. (2019). Benchmarking machine learning algorithms for android malware detection. *Jordanian Journal of Computers and Information Technology (JJCIT)*, *5*(03).

Jiang et al. (2020). Android malware detection using fine-grained features. *Scientific Programming*.

Kumar, R., Xiaosong, Z., Khan, R. U., Kumar, J., & Ahad, I. (2018, March). Effective and explainable detection of android malware based on machine learning algorithms. In *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence* (pp. 35-40).

Kyaw, M. T., & Kham, N. S. M. (2019). *Machine Learning Based Android Malware Detection using Significant Permission Identification* (Doctoral dissertation, MERAL Portal).

Li, J., Sun, L., Yan, Q., Li, Z., Srisa-An, W., & Ye, H. (2018). Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics*, *14*(7), 3216-3225.

Liu et al. (2020). A review of android malware detection approaches based on machine learning. *IEEE Access*, *8*, 124579-124607.

Needham, M. (2021). Smartphone market share: Supply chain constraints finally catch up to the global smartphone market, contributing to a 6.7% decline in third quarter shipments, according to IDC. Retrieved Oct, 2021 from https://www.idc.com/promo/smartphone-market-share/os

Rana, M. S., Gudla, C., & Sung, A. H. (2018, December). Evaluating machine learning models for Android malware detection: A comparison study. In *Proceedings of the 2018 VII International Conference on Network, Communication and Computing* (pp. 17-21).

Rathore et al. (2020, December). Detection of malicious android applications: Classical machine learning vs. deep neural network integrated with clustering. In *International Conference on Broadband Communications, Networks and Systems*, 109-128.

Roy et al. (2020). Android Malware Detection based on Vulnerable Feature Aggregation. *Procedia Computer Science*, *173*, 345-353.

Shao, K., Xiong, Q., & Cai, Z. (2021). FB2Droid: A Novel Malware Family-Based Bagging Algorithm for Android Malware Detection. *Security and Communication Networks*.

Sharma, S., Krishna, C. R., & Kumar, R. (2020, November). Android Ransomware Detection using Machine Learning Techniques: A Comparative Analysis on GPU and CPU. In *2020 21st International Arab Conference on Information Technology (ACIT)* (pp. 1-6). IEEE.

Singh, D., Karpa, S., & Chawla, I. (2022). "Emerging trends in computational intelligence to solve real-world problems" Android Malware Detection Using Machine Learning. In the International *Conference on Innovative Computing and Communications* (329-341). Springer, Singapore.

Syrris, V., & Geneiatakis, D. (2021). On machine learning effectiveness for malware detection in Android OS using static analysis data. *Journal of Information Security and Applications*, *59*, 102794.

Wen, L., & Yu, H. (2017, August). An Android malware detection system based on machine learning. In *AIP Conference Proceedings* (Vol. 1864, No. 1, p. 020136). AIP Publishing LLC.

Yerima, S. Y., & Sezer, S. (2018). Droidfusion: A novel multilevel classifier fusion approach for android malware detection. *IEEE Transactions on Cybernetics*, *49*(2), 453-466.