

Machine Learning Techniques for Distinguishing Android Malware Variants

Irwansyah Irwansyah^{1*}, Tri Basuki Kurniawan², Deshinta Arrova Dewi³, Mohd Zaki Zakaria⁴, Nurhafifi binti Azmi⁵

¹*Faculty of Vocational, Universitas Bina Darma, Palembang, Indonesia*

²*Postgraduate Program, Universitas Bina Darma, Palembang, Indonesia*

^{3,5}*Faculty of Data Science and Information Technology, INTI International University, Malaysia*

⁴*Faculty of Computer & Mathematics Sciences, University Technology Mara, Malaysia*

(Received: June 13, 2024; Revised: September 20, 2024; Accepted: October 12, 2024; Available online: December 27, 2024)

Abstract

The advancement of portable devices has been quickly and dramatically reshaping the usage trend and consumer preferences of electronic devices. Android, the most common mobile operating system, has a privilege-separated protection system with a complex access control mechanism. Android apps require permission to get access to confidential personal data and device resources. However, studies have shown that various malicious applications can acquire permission and target systems and applications by misleading users. In this study, we suggest a machine-learning approach to classifying Android malware variants by mining requested permissions, real permissions, suspicious calls, and API calls that were obtained and used in Android malware applications. Selected features were selected using a feature selection called KBest. Feature selection techniques are used to minimize the scale of the features and increase the performance. Two types of Naïve Bayes classifiers, called Multinomial distribution and multivariate Bernoulli distribution, are used and compared in malware family classification for text classification. Both naïve Bayes types are evaluated using a confusion matrix based on 4022 Android malware applications belonging to 10 families. Experimental findings show that the Multinomial distribution offers a reliable performance from three tests experiment with an average accuracy of 95%.

Keywords: Machine Learning, Android, Malware, Process Innovation

1. Introduction

The Android operating system (OS) is one of the most widely used mobile platforms globally, powering electronic devices such as smartphones and tablets, alongside competitors like iOS, Windows Mobile, and BlackBerry OS. As of April 2020, Android dominated the global mobile operating system market with a 70.68% share, followed by iOS (28.79%), and smaller players like Samsung (0.17%), KaiOS (0.12%), and Windows (0.7%) [1]. Android's open-source nature, supported by Google, makes it particularly appealing, especially in developed countries. This popularity has driven a rapid increase in users, with global smartphone adoption reaching 4.77 billion users between 2013 and 2017, of which more than 2 billion are active monthly Android users [2].

However, this widespread adoption has also made Android a prime target for cybercriminals. Malware developers exploit vulnerabilities to gain unauthorized access to user data, creating significant security challenges for Android users and developers. Many users download free applications from the Google Play Store or unknown sources without adequate security measures, while others store sensitive personal data without proper backup or protection [3]. This exposes them to malicious software, commonly referred to as malware, which has seen an alarming increase in prevalence. For instance, research has shown that G-DATA reported 744,065 new Android malware samples in the fourth quarter of 2017, with malware increasingly bypassing traditional security measures like antivirus programs and system authorizations [2].

The variety of features and behaviors inherent to Android, such as permissions, call logs, network traffic, activities, and intents, makes it challenging to detect and classify malware effectively [4]. Identifying malicious behaviors and the critical features required for detection remains a complex task. As such, it is essential not only to detect malware

*Corresponding author: Irwansyah Irwansyah (irwansyah@binadarma.ac.id)

DOI: <https://doi.org/10.47738/jads.v6i1.493>

This is an open access article under the CC-BY license (<https://creativecommons.org/licenses/by/4.0/>).

© Authors retain all copyrights

but also to classify it into distinct categories to better understand its capabilities and impact [5]. Research suggests that focusing on a limited number of key malware features can achieve similar outcomes to more complex analyses, making feature selection crucial for optimizing malware detection and classification techniques [6].

Various approaches have been explored for Android malware classification, with Naïve Bayes being a commonly used algorithm. While it may lack the precision of more advanced classifiers, its simplicity allows for easier implementation and interpretation, making it a valuable tool in this domain [5].

2. Literature Review

Numerous scholars, academics, students, and researchers have applied clustering models to identify, organize, and categorize Android malware [3]. Various approaches to mobile malware detection and classification exist, including static analysis, dynamic analysis, and hybrid malware detection analysis.

2.1 Static Analysis

Static analysis involves inspecting software programs by examining the code without executing it. According to Qamar, this approach uses the program's properties to analyze the system and determine whether it is malicious or benign [7]. Some researchers have employed machine learning techniques, such as Bayesian classification, to expose practical malware behaviors through static analysis [2]. For example, researchers analyzed 1,000 benign apps and 1,000 malware samples from the Android Malware Genome Project. Of the total 1,600 samples, half were used as training sets, while 200 of the remaining 400 samples were used for testing.

2.2 Dynamic Analysis

Dynamic analysis involves monitoring application behavior in a controlled execution environment [8]. This method executes the program in real time to detect errors and identify malware during runtime. Applications are evaluated on real devices or virtual environments like the Android Virtual Device. The primary benefit of dynamic analysis is its ability to examine application behavior during runtime, generating critical behavioral data. For example, sandbox virtual machines or other tools are often used to simulate an execution environment [9]. Tools such as VizMal have been proposed to support dynamic analysis by visualizing active malware behaviors in Android applications. Another widely discussed method is the TaintDroid system, which uses dynamic taint analysis to track sensitive data throughout the application. TaintDroid identifies potentially harmful applications by tagging or labeling sensitive variables and tracking their data flow during execution. However, this approach has a limitation: it cannot monitor data that leaves the channel and returns as a network reply [9].

2.3 Hybrid Analysis

While both static and dynamic analyses have advantages, they also have notable limitations. Static analysis is time-intensive, whereas dynamic detection consumes significant computing resources. Hybrid analysis, which combines static and dynamic approaches, addresses these limitations by leveraging the strengths of both techniques. Qamar [10] describes how hybrid analysis uses permissions (a static feature) and network traffic (a dynamic feature) for malware detection. Tools like Wireshark capture network traffic, including HTTP and TCP communications. Key features in traffic analysis include packet size, average bits sent/received, and the ratio of incoming to outgoing network bytes. These features are analyzed to differentiate between benign and malicious applications. Hybrid analysis achieves up to 95.56% accuracy using a dataset of 115 malicious and benign applications collected from the MalGenome and Google Play Store datasets [10].

Another noteworthy hybrid approach is the SAMADroid model, which combines static and dynamic analysis to detect malicious behaviors using permissions, API system calls, and network addresses. The model employs a three-level hybrid structure integrating remote and localhost analysis, machine learning techniques, and Random Forest classifiers, achieving an accuracy of 99.07%. The SAMADroid model demonstrates superior performance compared to traditional detection methods like MADAM [2].

3. Methodology

This project has a few phases that must be going through to be able to accomplish the goals and objectives of the project. The research method is shown in figure 1.

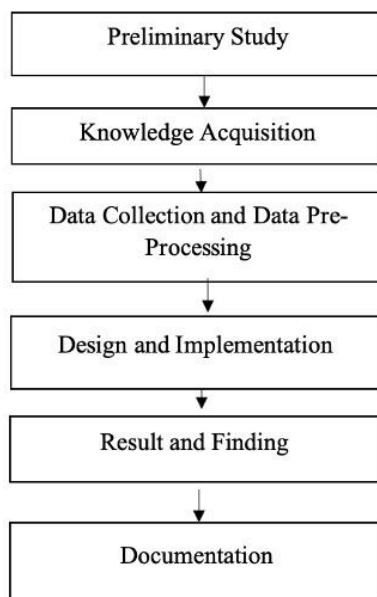


Figure 1. Research Framework

The project will start with a preliminary study of the main phases of domain learning and this project planning. The next step requires knowledge acquisition that needs reading to understand all the methods and techniques used based on existing work and research. After that, this project continues with the data collection and pre-processing phase, design and implementation phase, result and finding, and will finish with the documentation phase. [Figure 2](#) illustrates the system architecture used in this study.

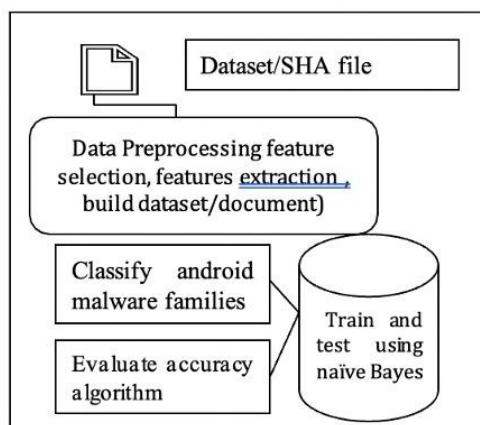


Figure 2. System Architecture

This research methodology starts with a preliminary study. It was an early study of the subjects correlated to the anticipated value of assessment or valuation and explanatory context. It was also a beginning step that required more focus and understanding of the proposed ideas. In this study, we identify the features of the project in an analytic review.

Information is needed mostly on domain Android malware and the technique used for Android malware analysis [11]. We gathered the pieces of knowledge of the related domain and techniques from various sites. Books, blogs, websites, journals, articles, researched papers, and reports were used for reading to comprehend the background study of the project, identify the problem statements, and come up with project objectives and their significance. The study shows the information and statistics on Android smartphones in the market, numbers of Android users, cyber threats including viruses and malware, cybercrimes such as attackers and hackers, and cybersecurity features [12]. A few questions became a good lead-in understanding of the domain. For instance, what has triggered the Android malware attack? Another thing learned in this study is the method and the technique used as an initiative taken.

In this phase, we went through a lot of reading and essential studies. We learned more details about our related domain and the method used from various sources such as review papers, academic articles, and journals. We also read and went through the research papers from a variety of online databases. Besides, through the literature review, more ideas became clear and could be added to the list of alternatives. Through this process, we were able to identify types of malware such as trojan, ransomware, scareware, key-loggers, and spyware.

Data collection phases are not simple as the data can be gathered from various sources. Legal sources such as academic research labs required authentication to access the dataset and bound to the policy and standard. Unable to fulfill requirements may result in a delay of time. Plus, some researcher have stopped sharing their Android malware dataset.

The dataset given consists of a directory named `features_vector` and a CSV file known as `sha256_family.csv`. The dataset contains about 129013 applications, and 5560 samples among them are malware. The `sha256_family.csv` file contains a list of malware applications[14]. Every line contains the SHA256 hash of the app and, therefore, the family represented by a label to which the malware belongs. For every app, there is a go into the `feature_vectors` directory, whose name is the SHA256 hash of the connected app, and it contains the list of all the extracted features, as shown in [figure 3](#) and [figure 4](#).

[illegible][illegible]

Figure 4. Drebin- feature vector file

Feature selection is the process of removing as much irrelevant and redundant information as possible. The presence of irrelevant information in machine learning algorithms may lead to several problems, such as difficulties in the learning phase, over-fitting of data, increasing complexity and runtime of classifier, affecting the accuracy of the model [17]. In this undertaking, we utilized a feature selection element technique called SelectKbest by using sklearn tools python called sklearn.features_selection. We implement the selection of features to reduce over-fitting [18].

In designing malware family classification, there are two main tasks, which are building a dataset, where we create with build dataset function and model the Naïve Bayes text. The build dataset function receives a few parameters, which are input the path of the features_vector directory and CSV file of the DREBIN dataset. The number of samples to extract (max_samples) and the file parsing mode (mode) are also parameters used to parse the file in the features_vector directory and build the dataset, as shown in [table 1 \[19\], \[20\]](#).

Table 1. Drebin Feature Vector

Features of Manifest.xml	Label
S1: Requested hardware components	feature
S2: Requested permissions	permission
S3: App components	activity,service_receiver, provider, service
S4: Filtered intents	intent
S5: Restricted API calls	api_call
S6: Used permissions	real_permission
S7: Suspicious API calls	call
S8: Network addresses	URL
Features of Manifest.xml	Label

The naïve Bayes text consists of a few function definitions, including split dataset function, fitting function, classify function, and evaluate function definition [\[21\]](#). This method receives the dataset built with the build dataset function as a parameter, and we randomly divide it into training and test sets. It method returns the training_set, training_class, test_set, test_class, and class. This training set and test set are the list of dictionaries where each dictionary contains a list of class features. We set the training class and test class as lists of class marks(family labels) linked to the training set and test set.

The fit function uses the training set and training class input parameters to train the Naive Bayes algorithm and to compute probability class(Pclass) and Pword_class. Pclass is a dictionary, and each member represents the likelihood of a particular class labeling/family (c) given in the training set. Pword_class is a dictionary, and each element represents the likelihood of the term provided by the class label and the docs in the training set. The training set comprises the samples (docs) used to train in the Naive Bayes algorithm to learn a specific class and features. At the same time, the training class provides the class labels for the samples in the training set. To classify the samples in test_set, it returns the list of predicted (the most probable) class labels for the samples in test_set. In this method, we estimate the best class value to assign to the document.

This method uses its input parameters test_set and predicted_labels to evaluate the classification performances by computing the accuracy and the confusion matrix. We calculate by counting the number of miss classification if the predicted class is not equal to the true class. Thus, the confusion matrix is calculated, and the accuracy is count by 1 - miss/length of predicted_labels. This section includes the steps and functions used during the implementation. The algorithm was coded using Python programming language and executed with Jupiter notebook and VS Code to produce a report performance to evaluate the model accuracy.

We first initialize the global variable and parameter of the built dataset function. Next, we create and build a dataset by selecting malware from the dataset that belongs to the 10(num_classes) most numerous classes. Then, we split the dataset into training and testing datasets. Next, we initialize the average accuracy variable to collect the average accuracy for both the Naive Bayes models. We train the model classifier using the fitting function, and next, we classify the sample application in the test set. We compute the accuracy and the confusion matrix, and we print the classification report using the classification_report method of the Scikit-learn library. Lastly, we compare the accuracy of the different naïve Bayes models, as shown in [figure 5](#).

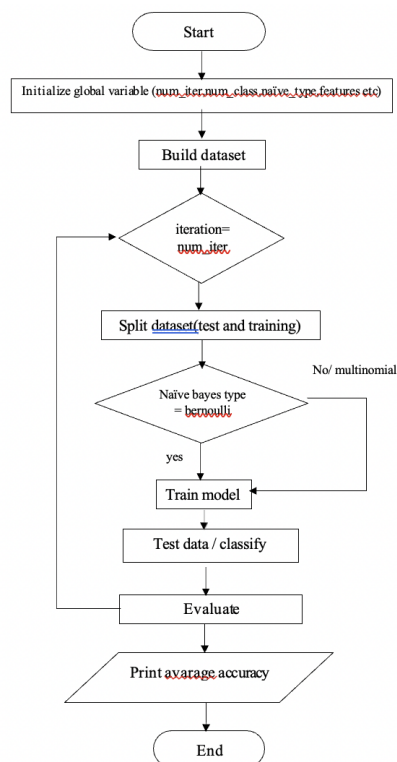


Figure 5. Flowchart Implementation

4. Results and Discussion

This chapter presents results and observations of the classification technique with different naïve Bayes types and the implementation of KBest features selection. The outcome obtained will decide which features of datasets are more important and which families are triggered by them. During feature selection, we ranked each feature based on its occurrence. We count the occurrence of each feature that appears in both benign and malware applications of the Drebin dataset. As a result, the four selected features from KBest are S2, S7, S6, and S5, with scores of 19501.372, 2537.355, 2370.475, and 2203.355, respectively. Based on the summary of four selected features, in five applications in the Drebin dataset, there were 48 total occurrences of requested permission, 17 total API calls, 15 total real permissions, and 21 total suspicious calls. Thus, we decided on the features permission(S2), call(S7), real_permission(S6), and API_call (S5) for us to use on our model as training and testing for Android malware family classification, as shown in [table 2](#) and [figure 6](#).

Table 2. KBest Feature Selection Score

Features	Col 2	Score
1: Requested hardware components	feature	855.957
2: Requested permissions	permission	19501.372
3: App components	activity, service_receiver, provider, service	226.068
4: Filtered intents	intent	1721.599
5: Restricted API calls	api_call	2203.355
6: Used permissions	real_permission	2370.475
7: Suspicious API calls	call	2537.355
8: Network Address	URL	608:951

```
Reading data from CSV file...
Found (5560) malwares in csv file.

Reading dataset files from feature vector directory...
Found (129013) files to classify.
Found (5560) malware files.
Found (123453) safe files.

Features Selection based on KBest:

scores for each attribute and the 4 attributes chosen:
[ 855.957 19501.321 226.068 1721.599 2203.355 2370.475 2537.355
 608.951]
[[11 7 6 11]
 [11 6 5 6]
 [ 4 2 2 2]
 [ 1 1 1 2]
 [21 1 1 1]]
```

Figure 6. Result- Feature Selection

It includes the outcome of the implementation of android family classification and also includes a comparative review of the effectiveness of two naive Bayes Classifiers for the identification and classification of 1,340 target malware samples belonging to 10 distinct families. To build the dataset, we first extract all the features and behavior of malware applications in the Drebin dataset. The dataset in the features_vector directory is extracted. All features, permission, API calls, real permission, service receiver, and others are extracted from each application. It is original and unclean data features an application, as shown in figure 7.

```
Selected Malware App ./drebin/feature_vectors/20b63c41abc231d278f499ade26ab4d1b2218489868182ee215d4468564bda7f

['feature::android.hardware.touchscreen', 'api_call::org.apache/http/impl/client/DefaultHttpClient', 'call::printStackTrace',
 'permission::android.permission.READ_PHONE_STATE', 'permission::android.permission.RECEIVE_SMS', 'permission::android.permission.INTERNET', 'service_receiver::services.SMSenderService', 'api_call::android/app/Activity->startActivity', 'activity::FirstActivity', 'service_receiver::sms.BinarySMSReceiver', 'api_call::android/telephony/TelephonyManager->getLineNumber', 'call::getSimCountryIso', 'permission::android.permission.SEND_SMS', 'call::getSystemService', 'intent::android.intent.action.DATA_SMS_RECEIVED', 'real_permission::android.permission.SEND_SMS', 'api_call::android/telephony/SmsManager->sendTextMessage', 'feature::android.hardware.telephony', 'real_permission::android.permission.READ_PHONE_STATE', 'call::sendSMS', 'intent::android.intent.action.MAIN', 'real_permission::android.permission.INTERNET', 'intent::android.intent.category.LAUNCHER']
```

Figure 7. Sample Original Feature

A dataset is a built-in form of the dictionary that contains all behavior from selected features and labels. Each application(document) is returned with its features as the document's word (family label and feature behavior). The dataset(document) is built from only ten family classes and only contains selected features to be extracted. The set of features that are stored in the vocabulary will be used as the training set. This is because we want to use the specific word to train the model. Figure 8 shows the new dataset built with the family label and clean features of requested permission, suspicious call, real permission, and API call with its occurrences in the current malware application.

```
In [11]: 1 dataset
Out[11]: {'20b63c41abc231d278f499ade26ab4d1b2218489868182ee215d4468564bda7f': {'features': {'org.apache/http/impl/client/defaulthttpclient': 1,
 'printstacktrace': 1,
 'read_phone_state': 2,
 'receive_sms': 1,
 'internet': 2,
 'android/app/activity/startactivity': 1,
 'android/telephony/telephonymanager/getlineNumber': 1,
 'getsimcountryiso': 1,
 'send_sms': 2,
 'getsystemservice': 1,
 'android/telephony/smsmanager/sendtextmessage': 1,
 'sendSMS': 1,
 'target': 'FakeInstaller'},
 'cd30d736d6128ae9680e149cd0c93cb17376d82d36bcf44c602cc269003a6566': {'features': {'getdeviceid': 1,
 'access_fine_location': 1,
 'getpackageinfo': 1,
 'httpost': 1,
 'android/location/locationmanager/isproviderenabled': 1,
```

Figure 8. New Dataset

After building the datasets, we selected applications from the top classes based on the specified number of classes, resulting in what we refer to as the parsed dataset. This parsed dataset is utilized for malware family classification and serves as our training and testing dataset. A total of 4,022 malware samples across 10 different classes were included in this parsed dataset. This selection process involves setting the desired number of classes for training and classification to 10, then aggregating all malware applications from these classes within the Drebin dataset. The breakdown of this process is illustrated in figure 9, figure 10, and figure 11.

```
=====Malware Family Classification =====  
Number of docs in the parsed dataset: 4022  
  
TEST 1  
Number of docs in the training set: 2682  
Number of docs in the test set: 1340
```

Figure 9. Split parsed dataset

```
training_dataset - Notepad (Not Responding)  
File Edit Format View Help  
'access_wifi_state': 2, 'android/media/mediaplayer/start': 1, 'access_coarse_location': 1, 'vibrate': 2,  
'process_outgoing_calls': 1, 'android/content/contentresolver/query': 1, 'obfuscation(base64)': 1,  
'android/media/audiorecord': 1, 'android/location/locationmanager/getbestprovider': 1}  
  
DroidKungFu -> {'android/net/connectivitymanager/getactivenetworkinfo': 1, 'access_fine_location': 2,  
'getpackageinfo': 1, 'getdeviceid': 1, 'httpost': 1, 'read_external_storage': 1, 'change_wifi_state': 2,  
'access_location_extra_commands': 1, 'android/net/wifi/wifimanagers/getconnectioninfo': 1, 'getsystemservice': 1,  
'access_network_state': 2, 'read_sms': 1, 'android/webkit/webview': 1, 'read_logs': 1, 'install_packages': 1,  
'cipher(desede)': 1, 'read_contacts': 1, 'android/content/context/startactivity': 1, 'write_sms': 1, 'cipher  
(aes/cbc/pkcs5padding)': 1, 'read_phone_state': 2, 'access_wifi_state': 2,  
'android/telephony/telephonymanager/getdeviceid': 1, 'read/write external storage': 1, 'getwifi_state': 1,  
'printstacktrace': 1, 'java/lang/runtime/exec': 1, 'setvifienabled': 1, 'cipher(aes)': 1,  
'android/net/wifi/wifimanagers/setvifienabled': 1, 'internet': 2, 'execution of external commands': 1,  
'set_wallpaper': 1, 'android/telephony/telephonymanager/getcelllocation': 1, 'write_external_storage': 1,  
'android/app/notificationmanager/notify': 1, 'get_tasks': 2, 'android/app/activitymanager/getrunningtasks': 1,  
'access_coarse_location': 1, 'vibrate': 1, 'android/content/contentresolver/query': 1, 'obfuscation(base64)': 1,  
'system/bin/su': 1, 'getsubscriberid': 1}
```

Figure 10. Training dataset

```
testing_dataset - Notepad (Not Responding)  
File Edit Format View Help  
DroidKungFu -> {'getdeviceid': 1, 'printstacktrace': 1, 'android/telephony/telephonymanager/getdeviceid': 1,  
'read/write external storage': 1, 'getwifi_state': 1, 'access_wifi_state': 1, 'java/net/socket': 1, 'wake_lock': 1,  
'change_wifi_state': 1, 'android/media/mediaplayer/start': 1, 'setvifienabled': 1, 'getsystemservice': 1,  
'java/lang/runtime/exec': 1, 'cipher(aes)': 1, 'execution of external commands': 1,  
'android/content/context/startservice': 1, 'read_phone_state': 1, 'android/net/wifi/wifimanagers/getwifi_state': 1,  
'read_logs': 1, 'android/net/wifi/wifimanagers/setvifienabled': 1, 'internet': 2, 'system/bin/su': 1}  
  
DroidKungFu -> {'java/net/httpURLConnection': 1, 'printstacktrace': 1,  
'android/telephony/telephonymanager/getdeviceid': 1, 'read_phone_state': 2,  
'android/net/connectivitymanager/getactivenetworkinfo': 1, 'access_fine_location': 1, 'access_wifi_state': 2,  
'getpackageinfo': 1, 'read_contacts': 1, 'get_tasks': 2, 'obfuscation(base64)': 1,  
'android/webkit/webchromeclient/ongelocationpermissionsshowprompt': 1, 'android/content/context/startactivity': 1,  
'java/lang/runtime/exec': 1, 'vibrate': 1, 'setvifienabled': 1, 'getsubscriberid': 1, 'getsystemservice': 1,  
'read/write external storage': 1, 'android/content/contentresolver/query': 1, 'change_wifi_state': 2,  
'access_network_state': 2, 'cipher(aes)': 1, 'getdeviceid': 1, 'execution of external commands': 1,  
'android/net/wifi/wifimanagers/getwifi_state': 1, 'system/bin/su': 1, 'read_logs': 1, 'write_external_storage': 1,  
'android/net/wifi/wifimanagers/setvifienabled': 1, 'getwifi_state': 1, 'internet': 2, 'install_shortcut': 1,  
'android/app/activitymanager/getrunningtasks': 1, 'android/app/notificationmanager/notify': 1,  
'receive_boot_completed': 1, 'httpost': 1}
```

Figure 11. Testing dataset

The number of documents in the training and test set, along with the number of documents of each class, show that FakeInstaller has the highest number with 589 documents, followed by DroidKungfu:448, Plankton:438 and the lowest is Gemini with only 64 documents. This is probably because we randomly split the dataset from the ten most probable classes into training and test sets. Therefore, we can conclude that in Drebin, the top 10 classes or the ten most probable are Opfake, GinMaseter, FakeInstaller, BaseBridge, FakeDoc, Gemini, Kmin, Plankton, DroidKungFu, and Iconosys.

After we train the model to learn, we classify each sample document in the test set by computing the probability of the documents and predicting it using argmax to select the best class for the document. The argmax result shows that the most probable class is FakeInstaller. Based on the Multinomial class, most malware documents are predicted as FakeInstaller, while in Bernoulli, Iconosys is labeled as FakeDoc, and Opfake is labeled as FakeInstaller. This is because, at the time of the computation, the probability of FakeInstaller is higher than other classes. Thus, the current document is stored to the predicted classes target FakeInstaller.

The result obtained is the average accuracy over three tests. The accuracy and the matrix of both models, naïve Bayes, are represented in the classification report and confusion matrix. Although we create random malware in the training and test set, the performance of the Bernoulli and Multinomial model is unexpected. During this evaluation, we can see that Multinomial has classified most of the documents correctly, while Bernoulli is not as accurate as the predicted class is misclassified. However, Multinomial often miss classified documents as FakeDoc. The accuracy of the Multinomial is 0.95, while Bernoulli is 0.83. Among ten classes of malware families detected and classified by Bernoulli, the precision of class DroidKungfu achieved the lowest precision, which is 0.28 and 0.53 precision. At the same time, the Kmin achieved 0.99 precision, BaseBridge, and 1, respectively, in Bernoulli. During those classes in Multinomial, only Kmin has 0.89, Gemini has 0.93, and FakeDoc has 0.95 precision.

However, most of the malware classes in Multinomial achieved 80 percent and above. This is because Multinomial learn also the duplicates since multinomial learn how frequently the word(behavior) occurs for every class. Figure 10 and figure 11 show the confusion matrix of each of naïve Bayes. The performance results of the classification models are summarized in table 3, table 4, and table 5. Table 3 presents the evaluation metrics for the 3-Bernoulli model, including the predicted class, the number of true predictions, the number of misclassifications, and the precision for each class. The model achieves high precision for several classes, such as Fake Doc and Base Bridge, with perfect scores of 1.00, indicating no misclassifications. However, its performance varies significantly across classes, with Fake Installer showing notably poor precision at 0.28, reflecting substantial misclassification issues. Overall, the 3-Bernoulli model performs reasonably well for most classes, achieving precision values near or above 0.90 for key classes like Gin Master and Gemini.

Table 3. Result Test 3-Bernoulli

Predicted Class	Total True	Total Miss	Precision
Op fake	94	7	0.93
Gin Master	233	31	0.88

Fake Installer	68	70	0.28
Gemini	216	14	0.94
Plankton	37	18	0.67
Kmin	84	1	0.99
Fake Doc	29	0	1
Droid Kungfu	51	45	0.53
Base Bridge	139	0	1
Iconosys	206	2	0.97

In contrast, [table 4](#) details the results for the Multinomial model, which demonstrates superior performance across nearly all classes compared to the 3-Bernoulli model. Several classes, such as Fake Installer and Iconosys, achieve perfect precision (1.00), indicating all predictions were accurate. Even for classes with lower performance, such as Kmin (0.86) and Gemini (0.93), the Multinomial model consistently outperforms the 3-Bernoulli model. This demonstrates the robustness and reliability of the Multinomial approach for malware classification.

Table 4. Result Test 3-Multinomial

Predicted Class	Total True	Total Miss	Precision
Op fake	97	2	0.98
Gin Master	219	5	0.98
Fake Installer	39	0	1
Gemini	321	24	0.93
Plankton	37	1	0.97
Kmin	98	16	0.86
Fake Doc	36	2	0.95
Droid Kungfu	51	2	0.96
Base Bridge	181	5	0.97
Iconosys	204	0	1

[Table 5](#) highlights the overall accuracy of both models across three test scenarios. The Multinomial model consistently achieves higher accuracy, ranging from 0.948 to 0.957, outperforming the Multi-variate Bernoulli model, which achieves accuracy values between 0.833 and 0.851. These results indicate that the Multinomial model is not only more precise but also more reliable for this classification task. The consistent improvement in accuracy and precision across classes suggests that the Multinomial model is better suited for malware family classification tasks, particularly when precise class differentiation is required.

Table 5. Result Accuracy

	Test 1	Test 2	Test 3
Multinomial	0.948	0.958	0.957
Multi-variate Bernoulli	0.833	0.851	0.834

This project only trains and tests the naïve Bayes model for malware family classification on random malware applications from the Drebin dataset based on features that we select using KBest features selection. In the future, it is recommended to test the model using features that are from the implementation of other feature selections. Also, it would be good if we could try this naïve Bayes of malware family classification on another dataset, such as AMD, Malgenome, etc, with optimization to improve the random selection of the data.

5. Conclusions

In this study, we investigated the effectiveness of machine learning approaches, particularly Naïve Bayes classifiers, in classifying Android malware families. Using a dataset of 4,022 Android malware applications spanning ten distinct families, we applied feature selection techniques such as KBest to optimize data analysis, thereby enhancing the efficiency and accuracy of the classifiers. Our findings highlight the superior performance of the Multinomial Naïve Bayes classifier compared to the Bernoulli variant. The Multinomial model achieved an impressive average accuracy of 95% across multiple tests, demonstrating its ability to effectively handle frequency-based features and accurately distinguish between diverse malware types.

The study also emphasized the pivotal role of feature selection in improving machine learning models. By focusing on key features—such as requested permissions, suspicious API calls, real permissions, and restricted API calls—we streamlined the model training process and achieved higher classification accuracy. Future research could expand this work by testing the models on additional datasets and exploring alternative feature selection techniques to further validate and enhance the robustness of classification models. Moreover, developing adaptive feature selection methods to accommodate the evolving nature of Android malware could lead to more dynamic and resilient malware detection systems. This research contributes to both academic and practical advancements in Android malware classification, providing a solid foundation for further studies aimed at addressing the growing sophistication of malware threats in the mobile ecosystem.

References

- [1] Z. D. Patel, "Malware Detection in Android Operating System," in *Proceedings of the IEEE 2018 International Conference on Advances in Computing*, vol. 2018, no. 1, pp. 366–370, 2018.
- [2] A. Qamar, A. Karim, and V. Chang, "Mobile malware attacks: Review, taxonomy & future directions," *Future Generation Computer Systems*, vol. 97, no. 3, pp. 887–909, 2019. doi: 10.1016/j.future.2019.03.007.
- [3] I. R. A. Hamid, N. S. Khalid, N. A. Abdullah, N. H. A. Rahman, and C. C. Wen, "Android Malware Classification Using K-Means Clustering Algorithm," *IOP Conference Series: Materials Science and Engineering*, vol. 226, no. 1, pp. 1–8, 2017. doi: 10.1088/1757-899X/226/1/012105.
- [4] A. Zeller, "Mining apps for anomalies," *Mining Apps for Anomalies*, vol. 2016, no. 1, pp. 1–1, 2016. doi: 10.1145/2975961.2990476.
- [5] S. Banin and G. O. Dyrkolbotn, "Multinomial malware classification via low-level features," in *Proceedings of the Digital Forensic Research Conference, DFRWS 2018 USA*, vol. 26, no. 4, pp. S107–S117, 2018. doi: 10.1016/j.diin.2018.04.019.
- [6] M. N. A. Zabidi, M. A. Maarof, and A. Zainal, "Challenges in high accuracy of malware detection," in *Proceedings of the 2012 IEEE Control and System Graduate Research Colloquium, ICSGRC 2012*, vol. 2012, no. 7, pp. 123–125, 2012. doi: 10.1109/ICSGRC.2012.6287147.
- [7] A. T. Kabakus and I. A. Dogru, "An in-depth analysis of Android malware using hybrid techniques," *Digital Investigation*, vol. 24, no. 1, pp. 25–33, 2018. doi: 10.1016/j.diin.2018.01.001.
- [8] A. I. Ali-Gombe, B. Saltaformaggio, J. R. Ramanujam, D. Xu, and G. G. Richard, "Toward a more dependable hybrid analysis of Android malware using aspect-oriented programming," *Computers and Security*, vol. 73, no. 1, pp. 235–248, 2018. doi: 10.1016/j.cose.2017.11.006.
- [9] M. Choudhary and B. Kishore, "HAAMD: Hybrid Analysis for Android Malware Detection," in *Proceedings of the 2018 International Conference on Computer Communication and Informatics, ICCCI 2018*, vol. 2018, no. 1, pp. 1–4, 2018. doi: 10.1109/ICCCI.2018.8441295.
- [10] A. Qamar, A. Karim, and V. Chang, "Mobile malware attacks: Review, taxonomy & future directions," *Future Generation Computer Systems*, vol. 97, no. 3, pp. 887–909, 2019. doi: 10.1016/j.future.2019.03.007.
- [11] D. Thakur, J. Singh, G. Dhiman, M. Shabaz, and T. Gera, "Identifying Major Research Areas and Minor Research Themes of Android Malware Analysis and Detection Field Using LSA," *Complexity*, vol. 2021, no. 1, pp. 1–15, 2021. doi: 10.1155/2021/4551067.
- [12] M. T. Ahvanooey, Q. Li, M. Rabbani, and A. R. Rajput, "A Survey on Smartphones Security: Software Vulnerabilities, Malware, and Attacks," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 10, pp. 1–10, 2020. doi: 10.14569/IJACSA.2017.081005.
- [13] ZhiXiong, "Dataset for Android Malware Detection," *IEEE DataPort*, vol. 2021, no. 1, pp. 1–2, 2021.

-
- [14] M. Huang and F. K. Chong, "Empowering Travelers with Airfare Comparison, Flight Tracking, and Real-Time Weather Forecasts on Android," in *Proceedings of the 2023 IEEE 11th Conference on Systems, Process & Control (ICSPC)*, vol. 2023, no. 12, pp. 7–12, Dec. 2023. doi: 10.1109/ICSPC.2023.00012.
 - [15] S. Sarwar, Z. Ul-Qayyum, and A. Kaleem, "Machine learning based intelligent framework for data preprocessing," *International Arab Journal of Information Technology*, vol. 15, no. 6, pp. 1010–1015, 2018.
 - [16] I. Gemp, G. Theocharous, and M. Ghavamzadeh, "Automated data cleansing through meta-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 2017, no. 1, pp. 1–6, 2017.
 - [17] L. Berti-Équille, "Learn2Clean: Optimizing the sequence of tasks for web data preparation," in *Proceedings of The World Wide Web Conference*, vol. 2019, no. 1, pp. 1–10, 2019.
 - [18] M. Bilal, G. Ali, M. W. Iqbal, M. Anwar, M. S. A. Malik, and R. A. Kadir, "Auto-Prep: Efficient and automated data preprocessing pipeline," *IEEE Access*, vol. 10, no. 1, pp. 107764–107784, 2022.
 - [19] C. Chai and G. Li, "Human-in-the-loop techniques in machine learning," *IEEE Data Engineering Bulletin*, vol. 43, no. 3, pp. 37–52, 2020.
 - [20] R. Gawhade, L. R. Bohara, J. Mathew, and P. Bari, "Computerized data-preprocessing to improve data quality," in *Proceedings of the 2022 International Conference on Power, Control, and Computing Technologies (ICPC2T)*, vol. 2022, no. 1, pp. 1–6, 2022.
 - [21] M. Huang and F. K. Chong, "Empowering Travelers with Airfare Comparison, Flight Tracking, and Real-Time Weather Forecasts on Android," in *Proc. 2023 IEEE 11th Conf. on Systems, Process and Control (ICSPC)*, vol. 2023, no. Dec., pp. 7–12, 2023.