Android Malware Prediction by Permission Analysis and Data Mining

by

Youchao Dong

**A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Computer and Information Science)
in The University of Michigan-Dearborn
2017**

Master's Thesis Committee:

**Associate Professor Di Ma, Chair
Associate Professor Jinhua Guo
Associate Professor Shengquan Wang**

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor Professor Di Ma, for giving me the opportunity to work in the fields of mobile security using data mining and machine learning, and for providing me continuous guidance and encouragement during my research work. I learned a lot from her in the numerous heated discussions about those tough research and practical problems.

Second, I would like to thank the rest of my thesis committee for reviewing my thesis and making insightful comments.

Third, I would like to thank my colleagues Linxi Zhang, Jiafa Liu, Ying Zou, Haoyu Li, Zheng Zhang in the Security Research Lab for discussing various interesting research topics with me, which inspired me in my research.

Next, I wish to thank the Computer and Information Department of the University of Michigan, Dearborn campus for sponsoring my study and research work at the university.

Last but not least, my deepest gratitude goes to my family for their never ending assistance and love. Special thanks to my wife, Yarong Gu, for not only supporting as a family member but also as a colleague who gave so many awesome suggestions and helps.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In recent years, smartphones have brought people's lives to a new high level. Smartphone applications, or Apps, are accelerating the process with many more functions getting developed, such as browsing the Internet, making payments, taking photos and share. However, the "Apps" are bringing potential vulnerability when they access private information from the phones, and mobile security has never been so much focused on like today. In this paper, we presented a novel Android Permission based malware detection technique. We first gather a huge set of both malware and benign Apps through web clawer and develop a tool to decompile Apps to source code and manifest files automatically. Then permissions with other information are extracted for each App, making up to a raw data set. Afterward, we apply data cleaning, dimension reduction and statical analysis to the raw data set. We find that the distribution of permissions for Apps shares a difference between malware dataset and benign dataset. Finally, we take advantage of machine learning algorithms, including Logistic Regression Model, Tree Model with Ensemble techniques, Neural Network and finally an ensemble model to find patterns and more valuable information. Other models are also discussed. Extended experiments using these various machine learning models are conducted in the end. From the results, we can see that our method generates a good accuracy, F-score and overall performance of malicious App prediction.

CHAPTER 1

Introduction

1.1 Motivation

Personal Portable Devices (PPD), or more commonly called mobile devices, have significantly brought people's personal lives to a new high level. Take personal phones for example, in the previous century when it was just invented and put into public use, all it can do is to text and make phone calls. Then smartphones came into the world, along with it there were many more functions such as browsing the Internet, making payments, take a photo and share it, people can even download more smartphone applications for more personalized or complicated things. During the first decade of this century, smartphones begin to be equipped with sensors such as light sensor, gyroscope sensor, Barometer sensor and more, which allows them to accomplish even more sophisticated work. To some degree, smartphones have surpassed personal computers regarding user frequency and dependence.

The broader capabilities of smartphones, along with increasing number of activities that users perform on them, has arisen quite grave concern concerning personal privacy protection and device security. Personal private information includes and is not limited to name, gender, age, phone number, photos, voice or video records, credit card info, even users' location history and web-browsing history.

Among all existing smartphone operating systems like Apple iOS, Blackberry, Windows Phone, Android, Palm OS and Tizen [1], Linux-based Android and Unix-based Apple iOS are the most two popular platforms. These operating systems are designed to be highly secured, plus they keep evolving and updating to fix bugs or loopholes. So it is quite hard to break through the systems and perform malicious actions. However, smartphone applications or Apps, provide potential vulnerability for accessing privacy through them.

As a matter of fact, there are more than two million Apps on both Google Play Store and Apple App Store[2], which are the largest providers of Apps for Android and Apple iOS, respectively. For such a huge amount of Apps, it must be hard for systems to determine which App is acting abnormally, or distinguish some sensitive operations are permitted by users or not.

To describe more about what is the whole process for a malicious application getting into action, here is a brief example. Assume there is an App called Better Photo, which leads users and App Store administrators to consider it as an App that can take photos and beautify them afterward. So smoothly the App developer releases it on the App Store, get approved, then users see it and download to use daily. Once this malicious App gets successfully installed, it performs just like other benign Apps and takes its routines correctly. However, deep down this App is programmed to access user's photo library, then steal the private photos and upload to some individual servers. Such mechanism and behavior are absolutely behind the scene, and no user can get aware of this. As long as the user does not uninstall this App, it will continuously steal photos all the time, which is definitely a big issue but the user doesn't know it at all. Malicious Apps must be detected and prevented!

1.2 Problem Statement

Due to Android's top popularity and open-source nature, it has also been the priority target of malicious Apps over other mobile operating systems, so our main focus will be on Android malicious App prediction. Truly, there already exists quite a few successful anti-virus software companies and products, but their approach to detecting malware is not totally suitable for mobile devices, which will be discussed and proved with more details in the next chapter. Again, take the Better Photo App as an example. From the perspective of an ordinary mobile device user, he wants to know if this App might perform malicious behavior before installing it, not get informed after he already did it, which will bother him again to uninstall. He may also wish that such protection is built in with his mobile phone operating system, not a third-party App which may take extra effort to look for, or may cost money, or just consumes more battery energy. What is more, such protection mechanism ought to act quite fast, invisible when not needed, not always rely on the Internet, and never suggest users do a virus scan or pop-up any advertisement. Meanwhile, even an App itself is not malware, as long as it is over-privileged, the App is still highly possible to be taken advantage of and passively assist malware according to [3] To sum up, what we aim to study and design is a light-weight, fast and invisible mechanism that could tell users not to install certain malicious Apps before they do it. These malicious Apps are not only true virus and malware, but also over-privileged or vulnerable Apps.

1.3 Our Approach

Our approach focuses on using as less information and low computational resources as possible to generate acceptable accuracy rate for detecting malicious Apps without actually installing them.

To achieve this goal, we studied Android system's permission model and found the breakthrough. Every App can be granted access to some particular information or sensors if and only if it claims to request corresponding system permissions at the time of installation. In other words, an active malicious App must already get approved of some over-privileged permissions before it can perform malicious activities. So, if users can be warned of claims for these over-privileged permissions, malicious Apps can be prevented from being installed. Then we also found Google's official APK-Tool, which allows APK files of Apps to be decompiled into source files. Permission claims for each App are stored separately in its manifest file, which saves a lot of computational resources of digging source code.

In this thesis, we first developed a web clawer to automatically download most popular and benign Apps in each category from Google Play App Store, then requested a malware App data set from an open source website. After that, we programmed Google's APK-Tool to batch to decompile benign and malware Apps to source code and manifest files. Then permissions with other several information were extracted for each App and stored as our raw data. Afterward, we applied data cleaning, dimension reduction and statical analysis to the raw data set. We found that the distribution of permissions for Apps shared a quite difference between malware dataset and benign data set, but it was really hard to come up with a rule to distinguish them. So we took advantage of machine learning algorithms to find patterns and more valuable information. Finally, we generated a good accuracy of malicious App prediction by tree models and neural networks with Ensemble technique.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 discusses the related works about malware detection and analysis applications, typical malicious behavior detection technologies, related study about Android permission model and permission-based detection. Chapter 3 describes the procedure to retrieve and pre-process the raw data, extract information and perform data cleaning. Chapter 4 analyzes the cleaned data from logic and detailed statistics perspective, then draws some useful conclusions. Chapter 5 first discusses some commonly used machine learning techniques and algorithms, including dimension reduction, re-balancing, Ensemble learning and three machine learning models, then applies these techniques, compares and analyzes results and performance. 6 concludes the thesis and highlights some future research directions.

CHAPTER 2

Related Work

In this chapter, we discuss the work related to our study in this thesis. Section 2.1 lists some related works about malware detection and analysis applications. Section 2.2 reviews some commonly used malware detection techniques. Section 2.3 introduces Android permission model. Section 2.4 explains some of the most typically risky permissions. Section 2.5 first states that permissions are being abused, then demonstrates the characteristics of malware permission use through some research results. We also present some recent researches in permission-based malware detection in this section.

2.1 Malware Detection Applications

There are quite a lot of powerful and modern malware detection companies or applications on the market, such as Bitdefender, Kaspersky, Norton, 360 Safe Guard, and Tencent Security Manager [4]. Some of them only focus on personal laptop systems like MacOS and Windows; some also support mobile systems like IOS and Android.

Since our primary focus is on Android, we found some specially Android-targeted security applications that had already gathered millions of downloads. To name a few most popular ones, they are 360 Boost, AndroHelm, Avira, TrustGo and AVAST [5]. Most of them store a huge database of malware's hash value on their server, then try to find a match in users' phones by scanning and computing hash value for all Apps. More techniques are discussed in detail in below sections.

2.2 Common Techniques

The most well-known technique for malware detection is signature-based detection, and it is also the primary method for most of the anti-virus Apps. Inter-Component Communication(ICC) analysis and Dynamic analysis are often utilized by companies or App servers to discover and collect more information from App source code and runtime behavior. Android Permission Model and Permission-based method are to study system permissions requested by Apps, which more specified for mobile systems, including IOS and Android.

2.2.1 Signature-Based Detection

The signature-based detection approach is frequently carried out by most of the daily used anti-virus software such as Norton and McAfee. To identify malware, this method first calculates the target file's signature which usually being its MD5 hash value, then try to find a match in its database of signatures that generated from known malware.

Such a strategy is quite convenient and efficient, but it is problematic because it usually fails to detect variants of known malware, which is easily accomplished by modifying any source code, or by using program obfuscation[6] such as packing and junk insertion. Therefore, signature-based detectors can only provide very limited zero-day protection, which means it is almost unable to detect malware unless this particular file has already been captured with the signature generated and stored in its database. Furthermore, since signature-based detectors have to use one separate signature for each malware variant, the database of signatures needs quite an ample storage space because it grows at an exponential rate. Attackers are quick to exploit this weakness by using program obfuscation techniques such as altering source code, polymorphism, and metamorphism to generate different manifestations of the same malicious behavior. Unless the Google Play Store itself provides such first-layer prevention before putting Apps on the market, it is not practical for users to seek for such techniques.

One more issue is that, since the malware signature database is too large to store on every user's device, this mechanism must use an Internet connection to transmit value or result, which is against our original intention to make it still functional when offline.

2.2.2 Inter-Component Communication Analysis

To overcome the disadvantages of signature-based approach, Inter-Component Communication Analysis, or ICC analysis is now one of the primary methods in Android malware detection[7]. This method is to track the information transferred between components and try to find if any privacy data is accessed or leaked.

Android applications are developed in Java and compiled to Java bytecode, then translated into DEX bytecode (Dalvik VM)[8]. To analyze data flow, it is required to decompile Android APK files to Java bytecode or Java source code. The ICC analysis includes permission call analysis, API call analysis, control flow analysis, data flow analysis, structural analysis, semantic analysis and so on.

For example, Jeong et al.[9] analyzes the instructions related to the system call sequence in a binary executable and demonstrates the result in the form of a topological graph called the code graph. Other researchers[10] combine the static and dynamic binary translation techniques. Critical API Graph based on Control Flow Graph is generated to do subgraph matching with the defined Malware Behavior Template. Some other researchers[11] infer the system call graphs from execution traces, and then derive a specification by computing the minimal differences between the system call graphs of malicious and benign programs. Shabtai et al.[12] applied machine learning classifier techniques like decision trees, Naive Bayes (NB), Bayesian Networks (BN), etc. to classify Android applications like games and utilities citing the non-availability of malware applications. They collected around 22,000 features initially and later reduced to 50 features for the purpose of classification. William[13] proposed an approach where a certificate is generated during an application's installation. This certificate gives complete information about the application by rating them using Kirin security rules, which are based on the combinations of permissions extracted from Manifest file. DroidAnalyt[14] is a signature-based system for detecting repackaged applications.

The first obvious drawback of such a technique is high implementation difficulty. Furthermore, to implement this on a sufficient scale, considerable computational resources and long process time are necessary, which is not quite a good choice according to our original intention of making it small, efficient and effective.

2.2.3 Behavior-Based Dynamic analysis

Dynamic analysis is also proposed for purposes of tracking data flow in mobile applications, especially in Android malware detection. It focuses on App's runtime behavior, such as accessing private information without notifying the user or try to send text messages in the background. TaintDroid[15] and Vet-Droid[16] are dynamic taint analyses that track information flow by instrumenting the Dalvik VM. However, in fact, many benign Apps also transmit sensitive data for performing their required functionality, especially multi-functional or communication and social Apps like Wechat, Facebook, and Instagram. For example, in Facebook App, one user takes a photo, then access its GPS information, after that he accesses his contact list and send it to one of his friends. In this scenario, the App accesses quite a lot sensitive information and transmit them through SMS message or Internet, which should alert the dynamic analysis mechanism and regarded as malicious behavior. Thus, taint analyses are not enough accurate and sufficient to automatically differentiate malicious Apps from benign Apps, and the combination of taint analysis with high-level malware signatures is needed to identify malicious code efficiently.

More importantly, dynamic analysis technique demands Apps to be installed first; then it can start monitoring and analyzing it in an active system or sandbox. Thus, it's not ideal based on our intentions, which is to notify users beforehand, not after installation.

2.3 Android Permission Model

Permission models have become one of the primary security mechanisms for smartphone operating systems to provide access control to sensitive information or hardware. According to Android's official developer documentation [17], no Android App has permission to perform any operations that affect user's information, or the operating system, or other Apps by default. Every Android App operates in its process sandbox with distinct system identity, so each App is separated from others and the running system. Thus, in this architecture, each App must explicitly declare the hardware, information or resources they need to share or use, which is contained in system permissions.

Among modern mobile operating systems namely Android, Windows Phone, Apple iOS and Blackberry OS, Android's permission model has extremely high controllability over OS. Meanwhile, the naming of permissions gives intuitive and exact information on what it requests from OS [18].

Android system also demands every App must contain a precisely named AndroidManifest.xml file. The purpose of this file is to list all relevant information about this App to notify the Android system, including system permissions which are under the <uses-permission> tag.

There are four types of permissions in Android permission system. The first two categories, Signature and System permissions, are not designed for third party applications, so they are not in our consideration. The later two are Normal and Dangerous permissions, available for requesting by Apps. For Normal permissions, the system will automatically grant them without involving users, while Dangerous permissions will alert the user before installing. However, most daily users will ignore this warning and proceed to continue.

Currently, there are almost two hundred different permissions according to our experiment.

With such a large number of permissions, it is quite hard for ordinary users to distinguish whether an App is over-privileged or not, or judge if it needs some certain permission to carry out its function. Furthermore, users only interact with the permission system for installing, if they deny the permission requests, installation process is terminated right away.

Among over 200 available permissions provided by Android and 75-80 prevalently request permissions, despite different Android Versions, the study in [18] proves there is quite low redundancy level, which means almost every permission has its own significance, and the correlation coefficient between any two permissions or permission groups is quite low.

2.4 Typical Permissions

Some of them are more dangerous than others in the sense of user security among over 200 permissions. We list some typical permissions below in the following content [19].

2.4.1 Privacy Related Permissions

Some malware attempts to gather users' private information that is not intended to share with them. Most of these permissions are read-related. For example, a basic calculator does not need to access user's voice mail, and it is not normal to request it.

- ACCESS_COARSE_LOCATION & ACCESS_FINE_LOCATION: Allows an application to access approximate or precise location.
- BODY_SENSORS: Allows an application to access data from sensors that users use to measure his/her body status, for example, heart rate.
- CALL_PHONE: Allows an application to start a phone call directly, skipping the system's Dialer user interface.
- CAMERA: Allows an application to access all camera features.

- GET_ACCOUNTS: Allows access to the list of accounts in Account Service.
- READ_CALENDAR: Allows to read the calendar data.
- READ_CALL_LOG: Allows an application to read the user's call log.
- READ_CONTACTS: Allows an application to read the user's contacts.
- READ_PHONE_STATE: Allows an application to read the phone state, for example, phone number and status of ongoing calls.
- READ_SMS & RECEIVE_MMS & RECEIVE_SMS: Allows an application to read or monitor user's SMS and MMS messages.
- RECORD_AUDIO: Allows an application to record audio from the user.

2.4.2 Write or Modify Related Permissions

Some malware attempts to modify users' private information, some of them may be misleading to the users. For example, it is abnormal if basic calculator requires sending text messages, and this malware may try to send user's private information. Some even attempt to modify a contact number to a misleading fraud number.

- ADD_VOICEMAIL: Allows an application to add voicemails into the system.
- PROCESS_OUTGOING_CALLS: Allows access to the outgoing call number, and redirect or abort the call.
- SEND_SMS: Allows an application to send SMS messages.
- WRITE_CALENDAR: Allows an application to write on user's calendar.
- WRITE_CALL_LOG: Allows an application to write the user's call log data.
- WRITE_CONTACTS: Allows an application to write the user's contacts data.

2.5 Permission-Related Research

2.5.1 Permission Abuse

Privacy protection is so fragile when on mobile devices, and more attacks could be brought to cause privacy leakage, with new permission or combinations usage. For example [20] has proposed a method to recover human speech by directly accessing to the gyroscope sensor on a mobile device, which is after just requesting a permission and got granted.

Stowaway [21], used for detecting over-privileged permissions for Android Apps. Their research shows there is almost one in every three Android Apps that has over-privilege issues.

Moreover, although some malware attacks mobile devices directly, some may just steal sensitive information from the back. In [22], it is stressed that after the approval of two permissions named "READ_EXTERNAL_STORAGE" and "WRITE_MEDIA_STORAGE", the App will have access to mobile device's real SD card, where usually important photos, videos, and notes are stored. It also points out that, Android system did not make a quite clear statement that such permissions should only be granted top-priority Apps or the system itself, not for every App to request freely.

Thus, it has gradually become a consensus that permissions need to be controlled. Although by principle, every Android App should declare all the permissions that it may require, which also decides what this App is able and not able to do, there do exits one approach called collusion attacks [3] that can lighten one malware's suspicion by not declaring some sensitive permissions. In such cases, malware developers have to program at least two Apps that work in conjunction. The first App, which is usually big in size and only intended to act generally, requests several extra permissions and establishes an inner connection at runtime to the second App, which is developed to do the dirty work but did not declare those permissions to elude anti-virus software. In such situations, we regard Apps like the first App as malicious or vulnerable because it might be potential accessory malware or it can be easily re-programmed to. It's definitely more dangerous and harder to detect such cases, so it comes quite necessary to distinguish Apps that declares more or fewer permissions than it should, compared with benign Apps.

2.5.2 Malware Permission Characterization

Lots of studies on malware characterizations has been conducted including [23, 24, 25].

According to David Barrera's work [23], more than one thousand popular benign Android Apps are analyzed regarding different categories tend to request a different set of permissions, there are still several permissions commonly requested among all, such as Internet and READ_PHONE_STATE.

The study in [24] shows that Android malware tends to utilize some permissions that not as frequently requested from benign Apps, such like "sendTextMessage" to send a text message without notifying the users. It also researched statistics on both 1260 malicious Apps and benign Apps and recognized a significant difference between two data sets regarding most frequently requested 20 permissions.

DroidSwan [25] points out eight permissions as suspicious, including "WRITE_APN_SETTINGS", "INTERNET", "WRITE_SMS" and "WRITE_EXTERNAL_STORAGE", which corresponds to other works. It also weighs suspicious permission combinations as 5 points and suspicious permissions as 3 points.

2.5.3 Permission-Based Malware Detection

[26] is a permission-based Android malware detection technique. Similar as in our method, it first extracts features from .apk files, and creates a dataset from the features. It then performs machine learning approaches (K-means, J48 Decision Tree, Random forests and Classification and Regression Tree) to classify the samples. However, the results are not ideal since the authors have not complied detailed feature engineering (some of their arguments are too arbitrary) and data preprocessing, and their machine learning algorithms are very limited. Their dataset is also small.

[27] is a lightweight method to detect Android malware using static analysis. They extract features from the manifest file including hardware components, requested permissions, App components and filtered intents, and also feature sets from disassembled code. It then applies learning-based detection. However, this paper focuses on various aspects, although the authors have some study on permissions, they do not put more emphasis on the details. Furthermore, their learning-based detection only uses Support Vector Machines, neglecting deeper research on other machine learning algorithms. Similarly, [28] implements permissions and control flow graphs as the features, and also only uses Support Vector Machines. [29] utilizes extra information like permissions, Intent messages passing K-means to classify clusters, and then uses Singular Value Decomposition on the low-rank approximations. Other than the typical drawbacks above, this kind of unsupervised learning is seldom applied in practice nowadays.

CHAPTER 3

Data Acquisition and Preprocessing

3.1 Data Source

Android application package (APK) is a format created by Google, which is used to distribute and install an application onto the Android operating system. VirusShare.com is the most famous free virus-sharing website with currently 27 million virus samples, as well as a repository of malware samples to provide security researchers, incident responders, forensic analysts, and the morbidly curious access to samples of malicious code. We were provided by it with a malware dataset consisting of 24317 Android malicious APK files. These APK files have already been renamed to their hash value, which made it impossible to retrieve the original name or category. This data set was generated in April 2015.

To validate the authenticity of this malware set, we utilized several popular anti-virus software to run through the set. However, due to high consumption of computational resources or workload of the software, all of them crushed before completing scanning. As far as the completed scans of 7% in the data set, there are around 1700 malware APK files, which will be scaled to exactly 24,000 if they are all scanned successfully. Thus, we can conclude that almost every APK file is indeed a malicious App.

However, due to the high popularity of VirusShare.com, it is possible that anti-virus companies also got this malware set and added it to the database before we carried out experiments. In this situation, our experiment could be pointless, and there would be no effective and efficient ways to validate the authenticity of this malware set. On the other hand, we have not seen any related paper that discusses this malware set or uses it as training sample in any format.

As for benign Apps, we chose official Google Play Store as the data source. We utilized an open-source batch web clawer tool obtained from Github that introduced in[30] to automatically download Apps. There were more than one million Apps that have frequently been downloaded, but there was no way to validate them all or to get absolute purely benign Apps, so our best approach was to trust Google Play Store's review system and only download highly popular Apps or Apps from trusted and well-known sources. There were also different categories of Apps, which might affect the distribution of permissions. For example, an App from Social Category would be regarded as normal when requesting for access to contacts, but it should be malicious for an App such as basic calculators from Tool category. We decided to respect the quantitative frequency distribution of all Apps between categories and download a certain amount of top Apps in each category accordingly. As a result, we got 4100 benign Apps, which ought to be a good sampling of benign Apps. To examine if malware exists in this set, again we utilized the same several anti-virus software to run through the set and found zero malicious App.

As a conclusion, we got 24,317 malware and 4,100 benign Apps as raw data set. The next step will be decompiling and extracting permissions.

3.2 APK Decompiling

An Android APK file is a zipped directory, Manifest file, classes.dex file and raw resources such as images and layout files are compressed in it. There are quite several open-source APK decompiling tools such as Dex2jar[31], ApkTool[32] and JD-GUI[33]. We utilized ApkTool since it ought to be more powerful and easy to scale. However, these tools were designed only to work for one APK file at a time, so we developed a Batch script in Windows platform for continuously invoking ApkTool to decompile APK file, rename Manifest file accordingly and delete others that we were not interested. Time needed for decompiling one APK file may vary from several seconds to around one minute depending on file size. The whole process was carried on SafeLab's desktop computer in Windows platform, and it took around two hundred hours.

We found there were around 100 malware APK files that could not be decompiled, which meant they were not in APK file format at all. We tried to install them onto a real Android device, none of them succeeded. These files were then ignored from our research, as they only counted less than 1% and the reason could simply be filename extension mistakes by developers. This situation did not happen on benign Apps.

During this process, we noticed that malware appeared to contain fewer images and much less source code, which made them smaller in size than benign Apps. Since it is entirely logical, reasonable and may be useful when distinguishing malware from benign Apps, we decided to include file size as one feature along with permissions. The statistics regarding file size will be discussed in more details in the following chapter. As a result, we decompiled 28847 valid APK files and got their Manifest files; next step will be extracting permissions from them.

3.3 Permission Extraction and Feature Aggregation

Each Manifest file is encoded by Extensible Markup Language(XML) as manifest.xml, which defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. System permissions in this file requested by App shows a specific prefix string pattern "<uses-permission android:name="***"/>", in which "***" is the target permission. According to Android developer's guide, such format is mandatory and consistent among all Android versions; it is also unique from other formats, which would avoid extracting any non-permission strings. Figure 3.1 shows an actual and more intuitive example.

There are four permissions highlighted in this case. We also notice that in some cases, there exist duplicates even in the same manifest file. According to Android developer's guide, the duplicate

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="2" android:versionName="1.1" package="com.localizemeagora"
  xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    <uses-permission android:name="android.permission.SEND_SMS" />
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
    <uses-permission android:name="android.permission.INTERNET" />
    <application android:label="@string/app_name" android:icon="@drawable/icon">
        <receiver android:name="com.localizemeagora.SMS.SmsListener" android:enabled="true">
            <intent-filter>
                <action android:name="android.provider.Telephony.SMS_RECEIVED" />
                <category android:name="android.intent.category.HOME" />
            </intent-filter>
        </receiver>
        <receiver android:name="com.localizemeagora.Boot.ClassBootSystem">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
                <category android:name="android.intent.category.HOME" />
            </intent-filter>
        </receiver>
        <activity android:label="@string/app_name" android:name="com.localizemeagora.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name="com.localizemeagora.ServiceLocalizeme" android:enabled="true" />
    </application>
</manifest>
```

Figure 3.1: Example: permissions in a manifest file

request will not bring benefit or harm, so we planned to only count each permission once in the same file. The rest part of this manifest file could not provide more valuable information, and those contents could vary a lot, so ignoring them should be the right approach.

With the help of an open-source library "BeautifulSoup4", we developed a Python-based batch-processing tool to extract permissions and build the whole data set in two phases.

To build a more intuitive data set, we firstly wanted to put the most frequent permissions in the front and less common in the later. So in the first phase, we just iterate through all of the manifest files and record the times appeared for each permission. We successfully processed 24,377 malware Apps and 4,070 benign Apps, aggregate numbers both respectively and together, then sort the permissions by frequency decreasing order. Figure 3.1 shows the top 10 permissions by descending order of frequency. All the data later will be presented in such order. Table 3.2 and Table 3.3 shows top 10 permissions extracted from benign and malware Apps respectively, which indicate some difference between the distribution of permissions. Such difference and the information behind will be stressed and analyzed in the next chapter.

Table 3.1: Top 10 frequent permissions extracted from all Apps

android.permission.INTERNET	16675
android.permission.READ_PHONE_STATE	15896
android.permission.WRITE_EXTERNAL_STORAGE	15275
android.permission.ACCESS_NETWORK_STATE	15171
android.permission.ACCESS_WIFI_STATE	11982
android.permission.WAKE_LOCK	8577
android.permission.GET_TASKS	8530
android.permission.ACCESS_COARSE_LOCATION	7907
android.permission.VIBRATE	7642
android.permission.RECEIVE_BOOT_COMPLETED	7249

Table 3.2: Top 10 frequent permissions extracted from benign Apps

android.permission.INTERNET	2688
android.permission.WRITE_EXTERNAL_STORAGE	2662
android.permission.ACCESS_NETWORK_STATE	2657
android.permission.READ_PHONE_STATE	2520
android.permission.ACCESS_WIFI_STATE	2381
android.permission.GET_TASKS	1777
android.permission.ACCESS_COARSE_LOCATION	1715
android.permission.VIBRATE	1661
android.permission.WAKE_LOCK	1639
android.permission.ACCESS_FINE_LOCATION	1418

Table 3.3: Top 10 frequent permissions extracted from malware Apps

android.permission.INTERNET	13987
android.permission.READ_PHONE_STATE	13376
android.permission.WRITE_EXTERNAL_STORAGE	12613
android.permission.ACCESS_NETWORK_STATE	12514
android.permission.ACCESS_WIFI_STATE	9601
android.permission.WAKE_LOCK	6938
android.permission.GET_TASKS	6753
android.permission.SEND_SMS	6193
android.permission.ACCESS_COARSE_LOCATION	6192
android.permission.RECEIVE_BOOT_COMPLETED	5992

In [34], the authors examined 34,000 Apps from Android market (the author did not specify, but we should consider more than 99% of them are benign Apps here) and find that Apps of different categories have ranged percentage concerning containing duplicated permission requests, from roughly 0% to almost 30%. They also extracted top ten frequently repeated permissions. During the first phase, we also found 5138 malware and 1013 benign Apps declared duplicated permissions, making the proportion 21 and 33 percent respectively. From a statistical perspective, such a difference might be useful to determine whether an unknown App is malicious or not, so we added "whether duplicated permissions existed" as one of the features. As for the first step, duplicated permissions are only counted once in the same App. Another interesting fact is that there exist 2117 different permissions extracted from all Apps, but Android system itself only provides around 170 system permissions according to developer guide [19], numbers may vary between the various versions. This issue will be stressed and handled in the next section.

For the second phase, we also wanted the whole data set to be more structured, so we utilized the decreasing order of total frequency as a fixed universal order for all Apps. For each App, the tool will assign every requesting permission into its position and change the value to "1", and all the remaining permissions are set to default "0".

After executing these two-phase steps, a raw data set has been generated. The next move is to conduct data cleaning and add extra useful features.

3.4 Permission Data Cleaning

As mentioned above, invalid permission issue is the first problem to overcome. By mentioning "invalid" permission, it is not necessarily wrong or non-existent permissions. Although Android system provides around 170 permissions, only 100 of them get frequently requested, which means some permissions are rarely useful for daily Apps, or they have just come out with newer Android version. For the most situations, invalid permissions are typically caused by several reasons: input typo or wrong format; requesting self-defined or inner-App permissions; requesting hyper level permissions; requesting customized Android system's permissions(Some Android phone OEMs customize the system to be more appealing to local users, such as Huawei and Xiaomi). Because of that, among total 2117 permissions, 1382 of them only occurred once, 245 occurred twice, and 1854 occurred below ten times. So we eliminated more than 1910 invalid permissions and reserved the less frequent but actual valid permissions for future evaluation, making it totally 174 permission-based features. We also added the following features as discussed above: whether including duplicated permission; file size; whether malware. As a result, there are totally 177 features for one App. Table 3.4 shows a representative example with 14 permissions from beginning, and also the last three permissions.

Table 3.4: Permission extraction result for a typical App

1	1	1	1	1	0	0	0	1	1	0	0	1	0	.	.	.	0	0.5	1

Another issue is "zero" data. There are totally 7791 Apps that request zero permissions and their file size are relatively quite small, including 7205 Apps with rounded-up 0.0 MB in size. Reasons behind it could be mistaken data source(might not be a valid Android App at all), or errors when decompiling due to Google's APK tool. Although this kind of data had a significant amount, they were still discarded because they could not provide much useful permission information.

CHAPTER 4

Logical and Statistical Analysis

As mentioned in previous chapter, Table 3.2 and Figure 3.3 shows a significant difference regarding permission distribution between benign and malware Apps data set. This chapter will focus on the analysis of such difference from logical and statistical perspective.

4.1 Permission-Malware Correlation Analysis

Chi-squared test, also written as χ^2 test, is any statistical hypothesis test wherein the sampling distribution of the test statistic is a chi-square distribution when the null hypothesis is true. It is often constructed from a sum of squared errors, or through the sample variance. Test statistics that follow a chi-squared distribution arise from an assumption of independent normally distributed data, which is valid in many cases due to the central limit theorem. A chi-squared test can be used to attempt rejection of the null hypothesis that the data are independent[35].

In this section, Chi-squared test is utilized to determine whether or not permissions are requested equally regardless of malware or not, also known as discrete uniform distribution. Taken it as the null hypothesis, which also means each App's request for certain permission is independent of the App being malware or not. Then χ^2 is the sum of squared deviations divided by degrees of freedom, which can be calculated by the formula:

$$\tilde{\chi}^2 = \frac{1}{d} \sum_{k=1}^{n} \frac{(O_k - E_k)^2}{E_k} \tag{4.1}$$

Here O_k is Kth observed data, E_k is Kth expected data, d is degrees of freedom, using $d = (row-1)(col-1)$ to calculate the number of parameters of the system that may vary independently.

Table 4.1: Observed and expected count of Apps for $android.permission.SEND\_SMS$

Observed Number of Apps (Expected Number)	Malware	Benign	Total
Request Permission SEND_SMS	6380 (5799.3)	485 (1065.7)	6865
None Request	8916 (9496.7)	2326 (1745.3)	11242
Total	15296	2811	18107

For instance, Contingency Table 4.1 illustrates that $SEND\_SMS$ is requested 6380 times among 15296 malware but only 485 times among 2811 benign Apps, but according to the null hypothesis, numbers are expected to be 5799.3 and 1065.7, resulting in χ^2 603.4 and p-value $1.8\mathrm{e}^{-130}$ based on degrees of freedom as 1. Taking α as 0.05, according to Chi-Square distribution table, the threshold for χ^2 is 7.879, from which null hypothesis should be denied with a conclusion that there does exist a significantly statistical correlation between $SEND\_SMS$ and malware.

Instead of manually picking specific permissions to study like other papers, we developed a batch tool to conduct χ^2 test on all training features, including 174 permissions and one extra feature (whether requests duplicate permissions). There is one feature remaining and will be discussed in the next section, file size, which is non-applicable for χ^2 test because its value is continuous.

Table 4.2 lists part of all permissions ranked by descending χ^2 value. Still taking α as 0.05, degrees of freedom as 1 and threshold for χ^2 as 7.879, it can be concluded that top 120 permissions have statistical significance when predicting malware, especially permissions on the top have extremely low p-value, which means they are highly correlated to malware. There are 55 permissions currently show no strong statistical correlation to malware, but thy are only concluded non-significant when each single of them takes the χ^2 test. Later sections will also discuss the correlation between permission combinations and malware results. Another interesting fact is that most but a few of the permissions are negative-related to malware, which will also be discussed later.

Table 4.2: Permissions ranked by χ^2

Rank	Permission	χ^2	Correlation
1	android.permission.MANAGE_ACCOUNTS	1395.47	-1
2	android.permission.BROADCAST_STICKY	1274.89	-1
3	android.permission.DISABLE_KEYGUARD	967.1	-1
4	android.permission.WRITE_SETTINGS	874.96	-1
5	android.permission.CAMERA	792.38	-1
6	android.permission.RECORD_AUDIO	773.61	-1
7	android.permission.USE_CREDENTIALS	674.74	-1
8	android.permission.SEND_SMS	603.38	1
9	android.permission.AUTHENTICATE_ACCOUNTS	518.32	-1
10	android.permission.ACCESS_WIFI_STATE	513.61	-1
...			
120	android.permission.CLEAR_APP_USER_DATA	7.88	-1
121	android.permission.ACCESS_LOCATION	7.76	-1
...			
171	android.permission.BIND_WALLPAPER	0.03	-1
172	android.permission.SET_WALLPAPER_HINTS	0.03	1
173	com.anddoes.launcher.permission.READ_SETTINGS	0.02	-1
174	com.fede.launcher.permission.WRITE_SETTINGS	0.01	1
175	com.htc.launcher.permission.READ_SETTINGS	0	-1

According to Table 4.3, it is also notable that frequency ranking of permissions is not necessarily or strictly correlated to χ^2 ranking. Figure 4.1 more intuitively shows the distribution of Frequency-χ^2 ranking of permissions, in which only around 60 percent permissions have corresponding relations. As a short conclusion, frequent permissions do not always value more.

Table 4.3: Frequency and χ^2 ranking

Permission	Rank of Frequency	Rank of χ^2
android.permission.INTERNET	1	63
android.permission.READ_PHONE_STATE	2	110
android.permission.WRITE_EXTERNAL_STORAGE	3	27
android.permission.ACCESS_NETWORK_STATE	4	24
android.permission.ACCESS_WIFI_STATE	5	10
android.permission.WAKE_LOCK	6	42
android.permission.GET_TASKS	7	20
android.permission.SEND_SMS	8	8
android.permission.ACCESS_COARSE_LOCATION	9	16
android.permission.VIBRATE	10	18

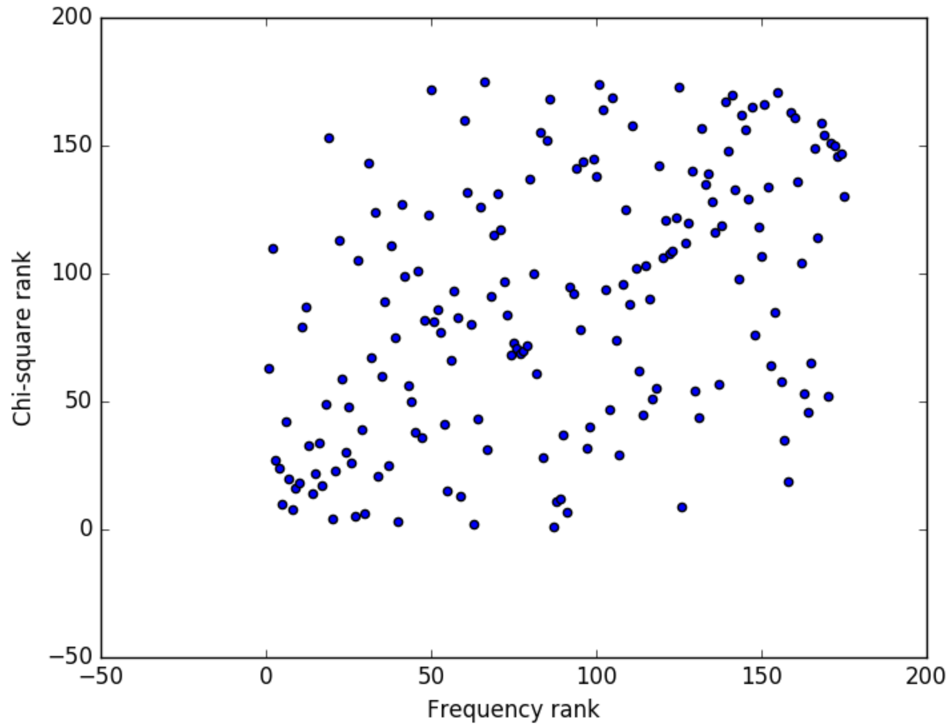

Figure 4.1: Frequency-ChiSquare ranking distribution

4.2 File Size Distribution and Malware Correlation Analysis

To better visualize the distribution of file size, we utilized histograms, set bin-width to be 1.2 and take $log_2(size)$ as x-axis value. Figure 4.3 and 4.2 both show nearly normal distributions for the file size of Benign App set and Malware set, which makes it appropriate to conduct a t-test on the two samples for determining whether the two distributions are statistically different. Take t-test type of "paired t-test with independent two data sample," statistic value turns out to be 40.41. According to t-table, the threshold value is 31.599 under two-tails' highest Confidence Level 99.9% and degree of freedom as 2, from which conclusion can be drawn that Malware and benign Apps' file size has a significantly statistical difference.

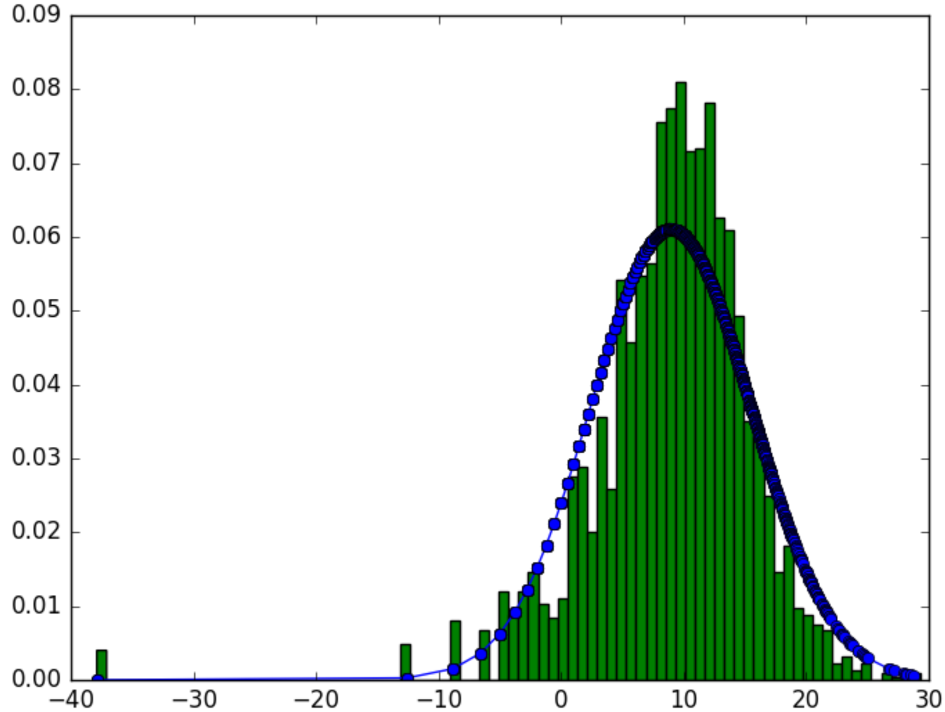

Figure 4.2: Benign App file size distribution

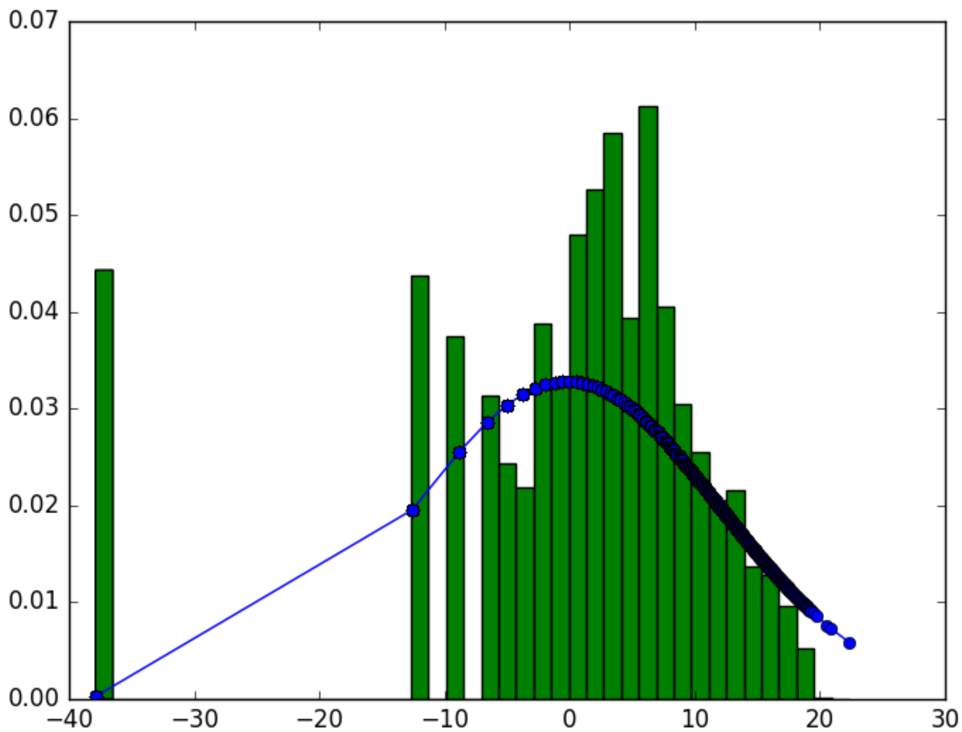

Figure 4.3: Malware file size distribution

4.3 Logical Analysis on Training Features

In previous sections, we only analyze the data and features from a statistics perspective, this section we will focus on the reasoning logic behind that, try to understand the meaning of data and test results. Additionally, because all permission requesting was conducted and coded by real human developers instead of automatically generated, we also try to speculate the developing process, motivations, thinking process or mental activities of developers, which is also quite important.

The difference in file size between malware and benign Apps is quite easy to understand: Benign Apps are usually programmed to realize more practical and complicated behavior, the logic of coding is also more sophisticated, all leading to bigger code base; while malware is typically intended to conduct some specific malicious behavior, making it straightforward and small.

According to the Chi-square test results, among 120 single relative permissions, 37 of them are positively correlated and 83 negatively correlated, besides there are only three positive permissions in top 50. Such a fact together with all previous conclusions could bring several reasonable inferences. First, malware tends to request some particular range of permissions in order to do something specific, such as $SEND\_SMS$ to send malicious text messages for illegal profit, or $INSTALL\_PACKAGES$ to secretly install other Apps. Second, Benign Apps tend to have more functions or abilities, resulting in requesting a wider range of permissions, such as online accounts related $MANAGE\_ACCOUNTS$ and $AUTHENTICATE\_ACCOUNTS$, or hardware like $VIBRATE$ and $ACCESS\_WIFI\_STATE$. Third, there also exist a range of permissions that get requested much more frequently in benign Apps than malware, such as $BROADCAST\_STICKY$, a permission involved in process/thread synchronization.

These inferences are not just reasonable but also true according to our knowledge and understanding of benign Apps as well as malware behavior and intentions: steal information, make profit or harm systems.

We are also interested in whether requesting duplicate permissions would help to decide malware, but its Chi-square test turns out to be 6.35 and slightly negatively correlated. To understand this, first our best guess before getting this result was that: for benign Apps, because it's often a big project with lots of permissions and may involve two or more developers, App developers may not be aware that certain permission has already been requested, which results in duplicate requests; while for malware, since it's usually small with fewer permissions and maybe just one developer, it's rather unlikely to request delicately. Following this logic, the correlation should be strong. On the other side, benign Apps tend to be developed by more organized developers if not by one; also good code management tools such as Git or SVN can prevent immature mistakes; while malware developers tend to be more self-educated, subterranean and often less organized, increasing the possibility to write duplicate codes. This weakens the assumption that benign Apps tend to have duplicate permission requests. In a word, both malware and benign Apps may or may not have duplicate requests, and that explains our statistic that this feature is slightly negatively correlated to the verdict.

4.4 Permissions Independence, Inner Correlation, and Combination Analysis

In the previous sections, we haven't studied the inner correlation and independence between permissions, which is a fundamental concept in statistics and rather essential for both understanding permissions and the next chapter. In this section, we conduct Chi-square test for each two of the 174 valid permissions, not including the extra features we add manually, making it totally 15225 possible combinations of them.

Table 4.4 shows some of the interesting combinations and its Chi-square statistics. Taking α as 0.05 and degree of freedom as 1, according to Chi-Square distribution table, the threshold for χ^2 is 7.879. There are almost half, totally 7335 of them rejected the null hypothesis and showed a significantly statistical correlation between them.

Table 4.4: Permission to permission Chi-Square test

Rank	Permission 1	Permission 2	Chi Stats
138	android.permission.READ_SMS	android.permission.WRITE_SMS	5920.664
139	android.permission.SEND_SMS	android.permission.READ_SMS	5907.707
140	ACCESS_NETWORK_STATE	ACCESS_WIFI_STATE	5857.667
141	MANAGE_ACCOUNTS	AUTHENTICATE_ACCOUNTS	5850.927
193	android.permission.READ_CONTACTS	android.permission.WRITE_CONTACTS	4325.753
452	android.permission.READ_CONTACTS	android.permission.WRITE_SMS	2109.178
695	android.permission.REBOOT	android.permission.DIAGNOSTIC	1134.952
938	com.android.vending.BILLING	com.android.vending.CHECK_LICENSE	779.545
7335	android.permission.WRITE_CALL_LOG	android.permission.SET_DEBUG_APP	7.887
7336	android.permission.RECEIVE_WAP_PUSH	android.permission.READ_PROFILE	7.876
15213	android.permission.READ_SETTINGS	android.permission.BIND_WALLPAPER	0

The 7335 correlated combinations can generally be divided into several categories with quite a few that raised our interest:

- The combinations of permissions have to be requested together, such as $com.motorola.launcher.permission.INSTALL\_SHORTCUT$ and $com.motorola.dlauncher.permission.INSTALL\_SHORTCUT$.
- Two permissions show a "one or the other" pattern due to different Android versions or OEM customized platforms, such as MOTO and LG.
- Two or more permissions, usually logically function together, such as $android.permission.READ\_SMS$ with $android.permission.WRITE\_SMS$, $ACCESS\_NETWORK\_STATE$ and $CCESS\_WIFI\_STATE$. This category has more practical meaning to us.
- Other combinations. This category takes about half of 15225 combinations; however, it is hard for us to define or explain their correlations, and thus this category does not make much sense to our research, resulting in a very low statistics.

We are also interested in how the dependency relation is between malware features and combinations of permissions. We conducted Chi-square test on all 15225 possible combinations against malware feature. Part of the results is shown in Table 4.5. Taking α as 0.05 and degree of freedom as 3. According to Chi-Square Distribution Table, the threshold for χ^2 is 12.838, and 13552 of the null hypothesis is rejected. Meanwhile, based our previous results, 120 permissions show a correlation with malware feature, making it 7140 possible combinations among 13552 current combinations. In the rest 6412 combinations, there are 290 combinations formed by the 50 previously non-significant permissions, as shown in Table 4.6. The result indicates that more complex combinations or the new type of combinations may also have been involved with our analysis. In fact, keeping more permissions adds more variables to our functions, and that we can calculate the coefficients more precisely.

Table 4.5: Chi-square test on Permission combination and Malware.

Rank	Permission 1	Permission 2	Chi Stats
1	android.permission. SEND_SMS	android.permission. WRITE_SETTINGS	2524.883
2	android.permission. BROADCAST_STICKY	android.permission. MANAGE_ACCOUNTS	2187.319
3	android.permission. DISABLE_KEYGUARD	android.permission. MANAGE_ACCOUNTS	2116.836
4	android.permission. SEND_SMS	android.permission. MANAGE_ACCOUNTS	1992.082
5	android.permission. WRITE_SETTINGS	android.permission. MANAGE_ACCOUNTS	1957.65
6	android.permission. RECEIVE_SMS	android.permission. WRITE_SETTINGS	1922.792
...			
13552	android.permission. BROADCAST_SMS	ru.apps.android.permission. C2D_MESSAGE	12.858
13553	android.permission. MODIFY_PHONE_STATE	android.permission. ACCOUNT_MANAGER	12.827
15225	com.htc.launcher.permission. READ_SETTINGS	android.permission. BIND_WALLPAPER	0.032

4.5 Conclusion

In this chapter, we first analyzed single permission - malware correlation by paired Chi-square test, found 120 dependent permissions. We then looked into the distribution of the file size of malware and benign Apps, found it a similar normal distribution, then conducted a t-test to conclude a significant difference in size distribution. After that, we logically analyzed the possible reasons behind the patterns of how permissions are requested along with other features. Lastly, we con-

Table 4.6: Previously non-significant permissions combining to be useful

Rank	Permission 1	Permission 2	Chi stats
1	android.permission. WRITE_SECURE_SETTINGS	com.android.alarm.permission. SET_ALARM	132.062
2	android.permission. EXPAND_STATUS_BAR	android.permission. SET_WALLPAPER_HINTS	108.817
3	com.htc.launcher.permission. WRITE_SETTINGS	com.lge.launcher.permission. WRITE_SETTINGS	90.913
4	com.htc.launcher.permission. READ_SETTINGS	com.fede.launcher.permission. READ_SETTINGS	81.96
5	android.permission. MODIFY_PHONE_STATE	org.adwfreak.launcher.permission. WRITE_SETTINGS	68.637
...			
289	com.htc.launcher.permission. READ_SETTINGS	com.lge.launcher.permission. WRITE_SETTINGS	12.976
290	android.permission. RECORD_VIDEO	org.adwfreak.launcher.permission. WRITE_SETTINGS	12.971

ducted Chi-square test between permission and permission, found 7335 among 15225 permission combinations show inner dependency and 13552 combinations show dependency with malware. For now, we have analyzed all possible situations that involving two or less permission. For example Permission A and B, we now know the relation between A, or $\tilde{A}$ against malware, B or $\tilde{B}$ against malware, A, or $\tilde{A}$ against B or $\tilde{B}$, A$\tilde{B}$ or $\tilde{A}$B or AB or $\tilde{A}\tilde{B}$ against malware, totally four situations. However, combination analysis only looks at two permissions, so when a third permission C comes in, currently analysis can not support new logical combinations such like $\tilde{A}$B$\tilde{C}$. The number of possible combinations is actually exponentially growing when there is no limit on permission numbers. Computing all of them is not hard, but analyzing them is. The best practice at this phase is to involve statistical model learning, or nowadays called machine learning techniques, which will be introduced in the next chapter.

CHAPTER 5

Machine Learning and Parameter Tuning

Machine learning is currently the most popular subfield of computer science, statistics [36] and mathematics. It enables programs to have the ability to "learn" the potential rule behind some distribution or phenomenon without being explicitly programmed [37].

Machine learning focuses on the construction, design, and optimization of algorithms that can extract, compute then learn from the provided data, to make correct predictions [38]. It is now widely employed in areas that require sophisticated logic and infeasible to explicitly program, or rules not intuitively captured or understood. This includes but is not limited to computer vision, stock market prediction, malware detection, anomaly prediction and search engines [39].

To implement machine learning techniques, we first generate a group of structured data. A set of training data is called a "dataset", where each record is a description of an event or object, called an "instance" or an "attribute" or a "feature"; a value of an attribute, referred to as an "attribute value"; the mathematical space generated by attributes is called "attribute space", "sample space" or "input space". We also refer to an example as a "feature vector." The number of attributes used to describe the object is also defined as the sample dimensionality.

The process of learning a model from data is known as "learning" or "training," this process is conducted by performing a learning algorithm. The data used in the training process is called "training data," Where each sample is referred to as a "training sample, " and a set of training samples is called a "training set." The learned model corresponds to a certain underlying law about the data, which we call "hypothesis," and this potential law itself is called "truth." Our goal of the learning process is to find or approach the truth. This process is also sometimes referred to as "learner," which can be seen as a learning algorithm in the given data and parameter space on the instantiation.

In this chapter, our goal is to develop a binary classification model for predicting malware through the training set. We utilize three frequently used machine learning algorithms including Liner Model, Tree Model, and Random Forester, and Neural Network and Deep Learning to classify the training Apps. In each section, experiments are conducted to classify test data to validate the accuracy of each of the model. In the end, we ensemble all the three. To measure the performance of the models, receiver operating characteristic (ROC) curve [40] is plotted, which illustrates the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings.

To allow any individual or organization reproduce our experiment results easily, we will always begin with the default model parameters provided by Scikit Learn, the most common machine learning library. We will try to make the least changes with specifying which parameter is changed exactly.

5.1 Dimension Reduction

Since each of our training samples contains 176 features and one class label, and there are totally 18107 training samples, train the whole set would take a lot of computational resources. Before we apply any machine learning algorithms, we want first to seek a method to reduce the dimension of features by utilizing some common techniques, such as PCA that discussed below.

5.1.1 Principal Component Analysis

Principal component analysis, or more commonly called PCA in data science, is a statistical process that uses an orthogonal transformation to convert a set of variables that observations are possibly correlated, into a set of values of linearly uncorrelated variables called principal components [41]. The transformed vectors, or called principal components, have a few number of original ones, but with a high percentage of original variables' variability. The principal components are defined to be sorted in descending order regarding largest possible variance. In such a way, original variables are re-constructed, and users can choose how many principal components to keep, achieving the balance between accuracy and computational resources.

For our whole training set, X, the transformation can be defined by an array of p-sized vectors $\mathbf{w}_{(k)} = (w_1, \ldots, w_p)_{(k)}$ that map each row vector $\mathbf{x}_{(i)}$ of X to a new vector of principal component scores $\mathbf{t}_{(i)} = (t_1, \ldots, t_k)_{(i)}$, given $t_{k(i)} = \mathbf{x}_{(i)} \cdot \mathbf{w}_{(k)}$. Individual variable of t inherits the most variance from x and is sorted in decreasing order.

As a result, we get each of the loading vector $\mathbf{w}_{(i)}$ that has to satisfy:

$$\mathbf{w}_{(k)} = \underset{\|\mathbf{w}\|=1}{\arg\max} \left\{ \|\hat{\mathbf{X}}_k \mathbf{w}\|^2 \right\} = \arg\ \max \left\{ \frac{\mathbf{w}^T \hat{\mathbf{X}}_k^T \hat{\mathbf{X}}_k \mathbf{w}}{\mathbf{w}^T \mathbf{w}} \right\} \tag{5.1}$$

Leaving all the remaining mathematically calculations behind, a full principal components decomposition of X can be represented as

$$\mathbf{T}_{szw} = \mathbf{X}\mathbf{W} \tag{5.2}$$

where W is a p-by-p matrix whose columns are the eigenvectors of $\mathbf{X}^T\mathbf{X}$

In our training set, the value of 174 permissions is categorical and binary specifically. Although PCA is designed for numerous features, binary values can also work. Figure 5.1 shows the percentage of variance for each principal component. The first, also the largest principal component, takes up less than 8%, which indicates the whole training set in a good variance distribution. Figure 5.2 intuitively shows the accumulated explained variance according to the number of principal components. The line is quite smooth, and to retain 99% of the original information, it requires almost

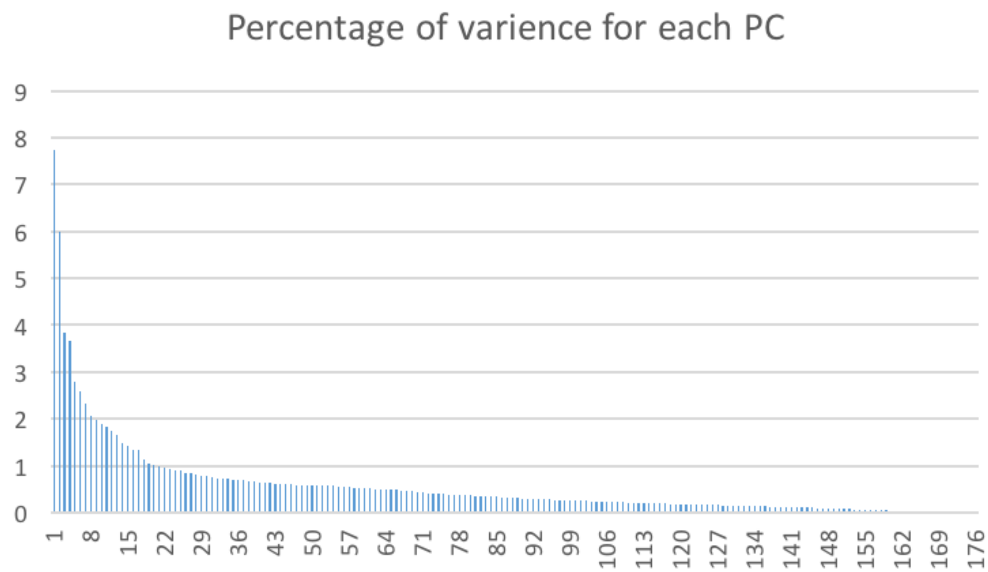

Figure 5.1: Percentage of variance for each principal component

150 features or 100 features for 90% of the information. Obviously, this compression could not bring a significance benefit. Therefore, we prefer to achieve the best accuracy and full information, instead of saving computational resources from compression.

Except for PCA, we also utilized Linear Discriminant Analysis (LDA) and Multiple Discriminant Analysis (MDA), but neither of them achieved an ideal outcome, so we decided to continue the machine learning process without feature compression.

5.2 Class-Imbalance Issue

Class-imbalance issue refers to a situation that numbers from different classes in the training set varies widely, which is common when dealing with those machine learning tasks originated from reality.

In our training set, there are 15296 positive samples and 2811 negative samples, making the positive proportion 84.5%. In other words, without any knowledge or decision criteria, by simply predicting it as positive will return a True Positive rate 84.5%. This result seems good to the training set, but obviously, it has extremely high bias and low variance.

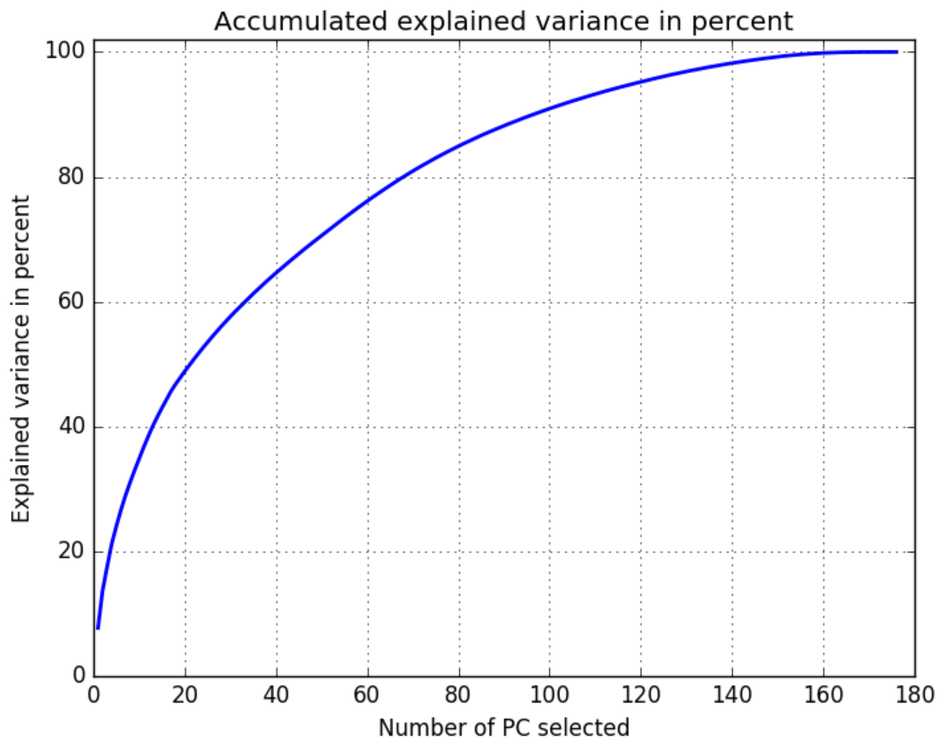

Figure 5.2: Accumulated explained variance

A core strategy of unbalanced class learning is "rescaling." The idea of rescaling is simple, but the practice is not trivial, mainly because the assumption "the training set is the unbiased sampling of the real sample" is always not true with real tasks. In other words, we may not be able to effectively base on training set observation probability to infer the true probability. Generally there are three types of common practices: 1) "under-sampling" the large class in the training set, for example removing some counterexamples so that the number of positive and negative examples are close and then continue learning; 2) "oversampling" the small class instance in the training set, that is, adding some examples to make the number of positive and negative examples close; 3) to learn directly based on the original training set, making adjustments on the threshold itself, which is embedded in its decision-making process when it is predicted by a trained classifier, called "threshold-moving". The time cost of a under-sampling method is usually much smaller than that of oversampling method, because the former discards many negative examples, which makes the training set of the classifier much smaller than the initial training set, while the oversampling method adds many positive examples, whose training set is larger than the original training set.

As for the specific situation of our training set, firstly the threshold moving technique is not feasible because it will be quite hard to move the threshold for every model and the Ensemble model. The under-sampling method may discard some important information if the counterexample is discarded randomly, while an advanced representative algorithm of the under-sampling method is EasyEnsemble [42], in which the authors used the integrated learning mechanism where the inverse cases are divided into several sets for use by different learners, so that each learner is under-sampled, but the important information is not lost in the whole view. The oversampling method must not simply repeat the sampling of the original positive samples. Otherwise, it will lead to a serious over-fitting; over-sampling method representative algorithm is presented in [43], and the authors over-sample through a mathematical interpolation method.

We finally conducted the oversampling technique for several reasons. Firstly, the previous statistical analysis showed there were more permissions and permission combinations indicating a negative correlation to malware, making every malware more valuable even though they were more in numbers, so under-sampling on malware was not preferred. Secondly, because the benign Apps we batched to downloading were the most popular and widely downloaded ones among users in its category, each of them had quite a few competitive products that can conduct very similar functions, resulting in them having almost the same distribution of permissions and file size. Even we did not download their competitive products, but since the training samples only contained permission and size information, we were able to conservatively conduct over-sampling on each benign App by creating five siblings to mimic its competitors. For each sibling, we randomly dropped one of original App's permissions, then re-adjusted its size to a random proportion between 80% and 120%. As a result, we achieved 15296 positive samples(malware) and 16427 negative samples(benign Apps), making it 31732 training samples. This method is right in conjunction with the mathematical interpolation to conduct over-sampling as stated in [43], and thus is a more convincing way compared to threshold-moving and under-sampling.

5.3 Ensemble Learning

Ensemble learning is to build a predictive model by integrating multiple models, and thus it improves the performance by taking advantage of every model. Moreover, some models may fall into local optimal solution, and combining every model may decrease this kind of risk.

In the following sections, we will introduce several machine learning models. We utilize each of the models and combine these models into a final result. In fact, in Section 5.5, ...; and in Section 5.6... We introduce the ensemble learning ahead in this section because this idea is throughout our entire machine learning analysis.

A typical ensemble method contains the following building blocks [44]:

- Training set. The training set is a data set used for ensemble training and has been mentioned many times in previous chapters and sections.
- Base Inducer. The performance of the classifiers depends highly on the induction algorithm used to generate them.
- Diversity Generator. We generate the diverse classifiers using diversity generator.
- Combiner. We combine the various models using a combiner. The combining strategy usually includes average, weighted average, majority voting, weighted voting, and learner.

5.3.1 Common types of ensembles

Bayes optimal classifier

This technique is to sum up all available hypotheses probability via a given vote proportional multiplied by prior probability of that hypothesis[45]:

$$y = \underset{c_j \in C}{\operatorname{argmax}} \sum_{h_i \in H} P(c_j|h_i)P(T|h_i)P(h_i) \tag{5.3}$$

Where y is the final output label, C is the set of all possible classes, and T is the whole data set. This method is not feasible to implement, either not intuitive to analyze.

Bootstrap aggregating (bagging)

Bagging, on the other hand, is easy to understand, control, scale and evolve. It first trains each different or same model with a different subset of the whole training set, then simply adding the code with equal or controlled weight[46]. Random forest is a typical example, and it has been achieving amazingly good accuracy without overfitting [47]. Later in our thesis, we will compare the accuracy of separate models with an ensemble model using bagging, and also try to learn a best weight distribution vector for all involved models.

Boosting

Boosting is another frequently utilized strategy. This method starts with weak learners, then emphasize the misclassified training data by previous models by assigning more weight. So in such a process, it involves incrementally until some termination criteria met. Boosting algorithms have been developing rapidly, among them Adaboost [48] is the most common implementation, while GBDT (Gradient Boosting Decision Tree [49]) is currently taking the dominator position in industry. Boosting has overall better performance than bagging, but it can also easily overfitting on training data since every mistake is stressed.

Stacking

Stacking usually refers to using a combiner machine learning model to take in all other models' output as input data, then train this combiner model (usually a simple logistic regression model).

In this thesis, since the number of training samples is rather limited while feature dimension is large, to avoid overfitting, we will only try a few ensemble methods to balance each models' output. We will start with linear models to try to learn the logic behind predicting malware.

5.4 Linear/Logit Model

5.4.1 Principal Procedure

Linear and logit models are the most commonly used machine learning algorithms. They are intuitive, easy to implement and quite powerful in many practical problems. We utilized Linear Regression as our first attempt to predict malware.

In Statistics, Linear Regression is a regression analysis of the relationship between one or more independent and dependent variables using the least squares function known as a linear regression equation [50]. This function is a linear combination of one or more model parameters called regression coefficients. Linear Regression was originally not designed for classification problem until it was reformed to Logistic Regression, developed by statistician David Cox in 1958 [51] To estimate the probability, David Cox used a cumulative logarithmic distribution function to assign the continuous-number-output to a class variable.

Given a data set $\{y_i, x_{i1}, \ldots, x_{ip}\}_{i=1}^{n}$ of n statistical units, the logistic regression model takes the form:

$$z_i = \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = \mathbf{x}_i^{\mathrm{T}} \boldsymbol{\beta} + \varepsilon_i, \qquad i = 1, \ldots, n \tag{5.4}$$

where β is a parameter vector of dimensional p, error term ε_i and response variable z_i. Then z_i is mapped to $f(z)_i$ using the following logistic function:

$$f(z)_i = \frac{1}{1 + e^{-z_i}}, \qquad i = 1, \ldots, n \tag{5.5}$$

Figure 5.3 shows the shape of the logistic function. The value of previous outcome z_i can vary from negative to positive infinity, but with the help of logistic function, $f(z)_i$ is bounded between 0 and 1, which makes it easy to decide the final class label by formula:

$$y_i = \begin{cases} 1, & f(z)_i >= 0.5, \quad i = 1, ..., n \\ 0, & else \end{cases}$$

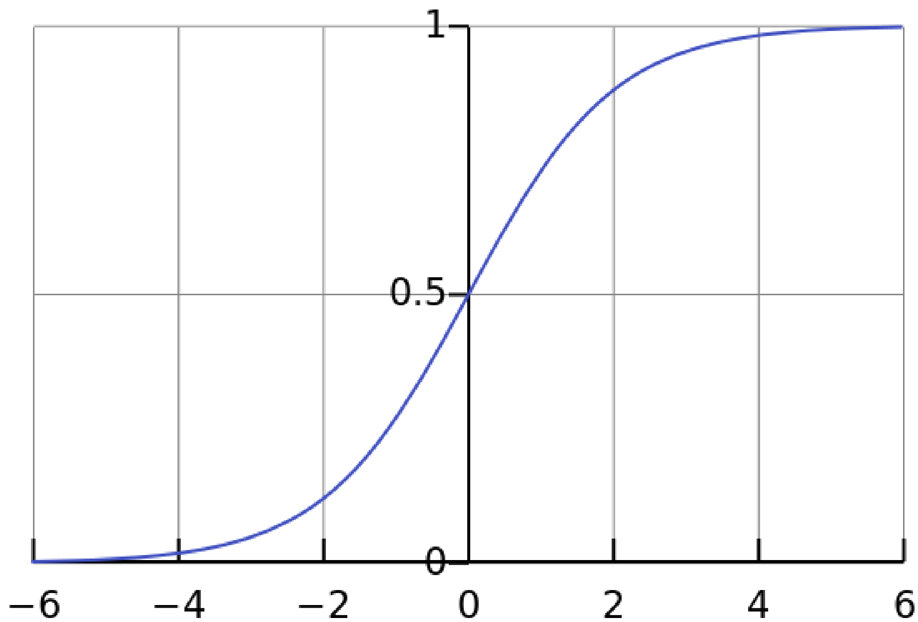

Figure 5.3: Logistic function

The next step is parameter estimation and model fitting through Least-squares estimation technique, which is expressed in formula:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} = \left(\sum \mathbf{x}_i \mathbf{x}_i^{\mathrm{T}}\right)^{-1} \left(\sum \mathbf{x}_i y_i\right) \tag{5.6}$$

After β is settled, the learning process of logistic regression is completed.

5.4.2 Result and Analysis

First, we want to compare model performance with L1 and L2 penalization. Figure 5.4 shows they have almost same performance using other default parameters. Even manually increasing the output model's penalization coefficient, or simply removing penalization term will not cause much difference. We think this results from the existing sparsity of feature distribution and relatively not complicated logic for the model to learn. In the analysis below, we will leave out L1 penalization for simplicity.

Using L2 regularization, the 10-fold cross-validation evaluation returns an overall accuracy of 77.03%, compared to the default 48.2%. On the randomly selected, reserved and un-learned test set (10%of the population), it achieves 78% accuracy as well. Result from the test set shows in Contingency Table 5.1 and Confusion Table 5.2. Changing regularization, the L1 returns 0.4% drop of accuracy, which is not significant.

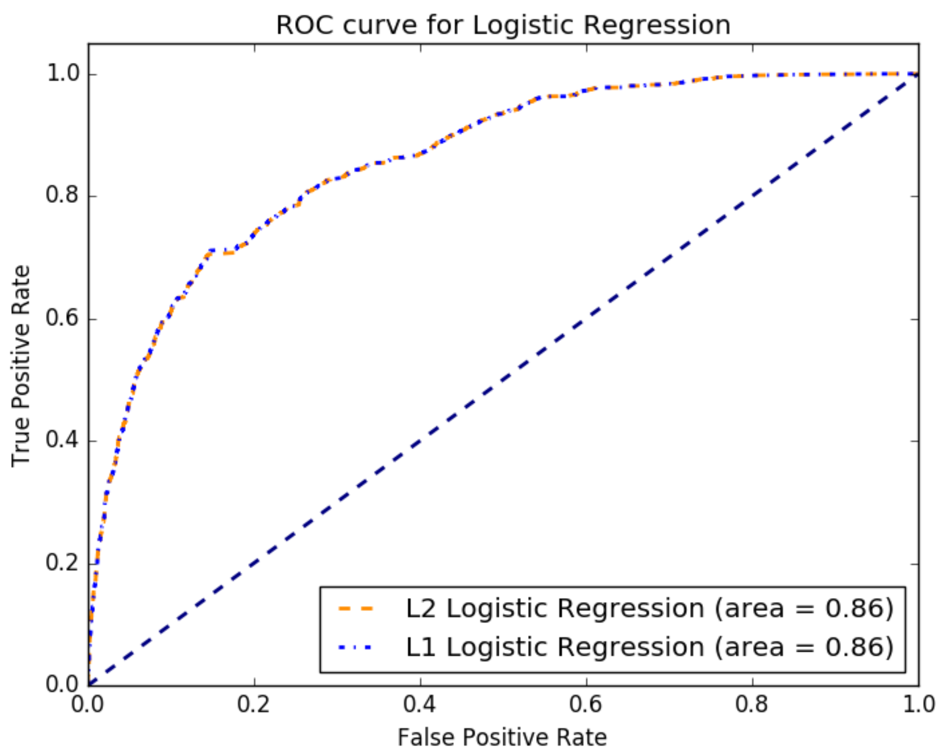

Figure 5.4: ROC curve for L1 and L2 Logistic Regression

Table 5.1: Contingency Table for test set

Contingency Table	Actually Benign	Actually Malware
Classified as Benign	1252	350
Classified as Malware	347	1224

One thing interests us is that, just taking a small percentage of data from the whole set as training samples will achieve a result accurate enough as 5.5 shows the Learning Curve. We also collect data by taking 0.1% to 15% as the training set with every 0.1% increase interval, at each percentage we repeat the training process by randomly selecting training sample for 100 times, record the statistics and generate the figures. Figure 5.6 and 5.7 shows the minimum, maximum and average accuracy along with accuracy deviation according to the percentage of the whole set. 5% of the training set(around 1600 training samples) leads to a steady 77% accuracy. Starting from 5% all the way up to 95%, the result stays the same, and the 10-fold cross-validation evaluation shows almost no sign of overfitting. As a conclusion, with logistic regression, we achieved a steady 77% precision, recall, and f-1 score. This performance is way under our expectation; one possible explanation is that linear models are not good at interpreting complicated logic functions including even a simple 'exclusive OR,' while the process of predicting malicious App by only looking at the binary features require quite a lot of logical conditions. To overcome this issue, we want to introduce decision tree and its ensemble models to achieve better results.

Table 5.2: Confusion table for test set

Confusion Table	Precision	Recall	F1-Score	Support
Benign	0.78	0.78	0.78	1602
Malware	0.78	0.78	0.78	1571
Average/Total	0.78	0.78	0.78	3173

5.5 Tree Models with Ensemble

Decision tree and all tree-based machine learning techniques are quite popular in both academia and industry [52]. Intuitively, this kind of method is implemented with a tree structure, in which root represents the whole input data, and each child node corresponds to a subset of input data, while each edge to child node indicates some condition or splitting rule. Each leaf node contains

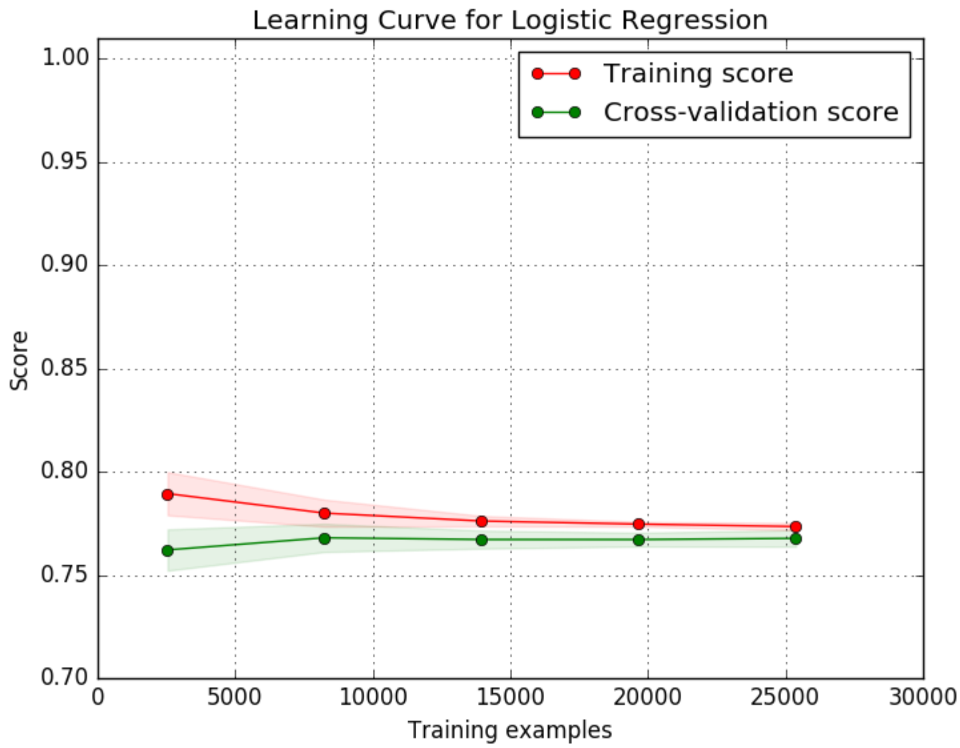

Figure 5.5: Learning curve for default Logistic Regression

the output result of this model, and it is reached by some subset of input data that qualifies all conditions or splitting rules through the path from the root to leaf. For a detailed description of tree algorithm, see [53, 54]

Tree-based algorithms have many advantages [55], such as:

1.Given an already constructed decision tree, its logic is much more easily understandable for humans than other popular machine learning algorithms, because it makes an explicit requirement on every alternative. Such a nature makes it quite feasible to be interpreted and explained for analytics purpose.

2.The computation needed to get the result for the program is also very cheap for the newly input data.

3.They are quite scalable regarding input data size or dimension and can be easily assembled with other algorithms, which will be discussed below in random forest.

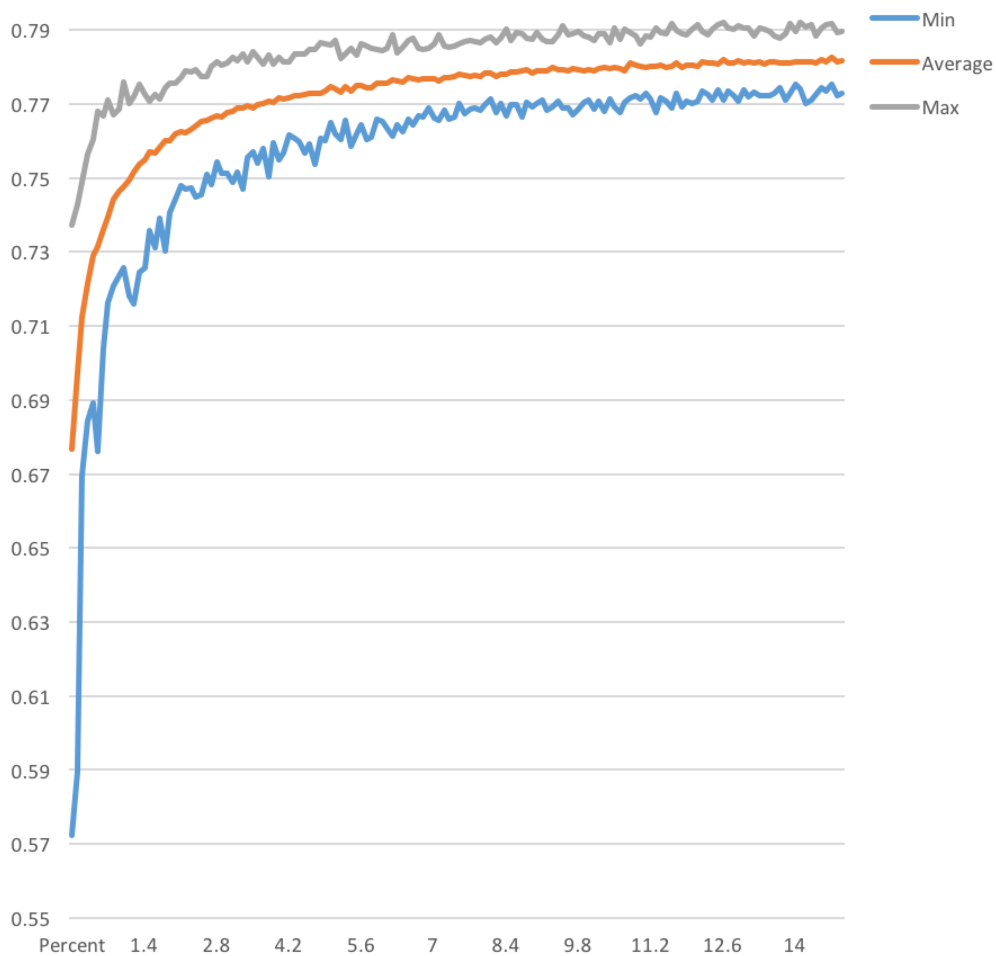

Figure 5.6: Min, max and average accuracy achieved by training set percentage

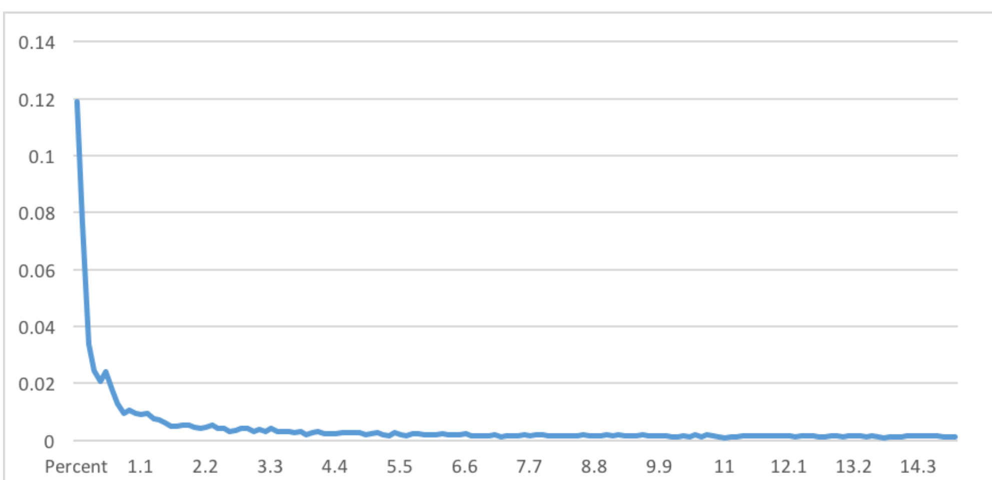

Figure 5.7: Deviation of accuracy by training set percentage

4.They do not need much feature engineering work because all data types for features can be input without transforming, besides nonlinear correlations or non-independence issues between features will not affect performance.

5.5.1 Basic Procedure

The algorithms for constructing a decision tree have been evolving for the past several decades. Most of them work top-down as a greedy approach, which is by choosing the current best variable that splits the set of data for each step(node). This recursive partitioning process is repeated for each node with corresponding data subset. Also, each algorithm has its rule of terminating the recursion process. Normally when all the subset data at current node have already reached the same value of target variable, termination criteria is met.

The definition and evaluation metrics for "best" is actually where different algorithms distinguish from each other. Generally there are three directions that metrics can go: Gini impurity, information gain and variance reduction [53, 54]. Splitting by variance reduction is more used when most target variable is continuous, which is not our case. Gini impurity measures the probability of incorrect label for a random data piece, under conditions that it's randomly assigned label according to label distribution for the whole training set. To compare different splitting rules, the value of Gini impurity can be computed as sum of probability f_i which indicates the chosen time with label i, multiplied by the probability of wrongly labeling it $1 - f_i$:

$$I_G(f) = \sum_{i=1}^{J} f_i(1 - f_i) = \sum_{i=1}^{J}(f_i - {f_i}^2) = \sum_{i=1}^{J} f_i - \sum_{i=1}^{J} {f_i}^2 = 1 - \sum_{i=1}^{J} {f_i}^2 = \sum_{i \neq k} f_i f_k \tag{5.7}$$

Here J is the total number of possible labels. By recursively computing for every node and choosing the current highest impurity value's corresponding splitting rule, finally a leaf node will reach zero as Gini impurity value, indicating that all data in this node belong to to a same label, then this path is completed. This method is slightly correlated but should be distinguished with the following technique. Information gain is the most commonly used metrics which is based on the entropy concept from information theory [56]. Entropy is defined as:

$$H(T) = I_E(p_1, p_2, ..., p_n) = -\sum_{i=1}^{J} p_i \log_2 p_i \tag{5.8}$$

$p_1, p_2, ...$ represent the distribution fraction of each class in the current node, sum of which is always 1. Information gain is defined as entropy value for current node minus the weighted sum of all child nodes' entropy value:

$$IG(T, a) = H(T) - H(T|a) \tag{5.9}$$

, by at each node choosing the largest information gain, algorithms tries to do the max split in order to keep the depth of tree as small as possible. This process is also repeated until all data in current node already belong to to a same label.

5.5.2 Random Forester and GBDT

In most cases different measuring metrics introduced above have similar performances, so we will not discuss more detailed metrics-related differences. Decision tree tends to overfit training set especially when it grows too deep without regulation or early pruning, but this drawback inspired scientists and engineers to apply the concept of ensemble learning into single decision trees. Random decision forests, or more commonly called as random forests [57, 58], is one of the ensemble learning methods that operates by combining multiple single decision trees when training. More specifically, the training process adopts bagging or bootstrap aggregating as introduced in Section 5.3, into a branch of single decision trees. While in each node for every tree, feature bagging is also utilized to ensure each tree does not reflect too strong correlation due to the correlation of specific features: only select a random subset of features, not all of them. Random forests also get developed to different variants including Extremely randomized trees (ExtraTrees [59]) or combining with unsupervised learning, but in this thesis, we will only compare the most common random forester algorithm with other machine learning algorithms.

GBDT(Gradient Boosted Decision Tree) is boosting ensemble technique combining with decision trees and learning by gradient as introduced in Section 5.3. It performs very well at regression and ranking problems[60] and it can also handle classification problems, it is also scalable and not easily to overfit but not that intuitive to understand as Random Forest. In this thesis, we will leave out the algorithm details and directly compare it with other ensemble techniques for reference.

5.5.3 Result and Analysis

In this section, we are presenting and analyzing results for different models. Here for Random Forest and Gradient Boosted Decision Tree, we are using its default hyperparameters from Scikit-Learn: 10 and 100 as a number of trees, Gini impurity as splitting measuring metrics, two as max depth and so on. Default single decision tree already returns a good 94% accuracy with 87% F-score, a 10-tree default Random Forest model returns a steady 95% accuracy and 90% F-score. Default Gradient Boosted Decision Tree only returns an 81% accuracy and 80% F-score, but it evolves to become better when size and depth get larger as expected, but still not as good as Random Forest. Figure 5.9 shows the ROC curves for different tree models including single decision tree, Random Forest, and Gradient Boosted Decision Tree. Random Forest outperforms others by large, but it does not get significantly better performance when increasing parameters to make it more sophisticated according to figure 5.10.

Figure 5.8 demonstrates the Learning Curve for default Random Forest model. It is noticeable that the gap between Cross-validation score and Training score is much larger than Logistic Regression, which indicates the Random Forest model might overfit the training set. Another possibility is that the training set is contradictory to itself to some degree. Even these hypotheses might be true; Random Forest is still much better than Logistic Regression for this particular data set. We are also interested to see model's performance on training set without oversampling by again comparing ROC curves. Taking Random Forest as example model, figure 5.11 shows that resampling does help a lot on improving model performance.

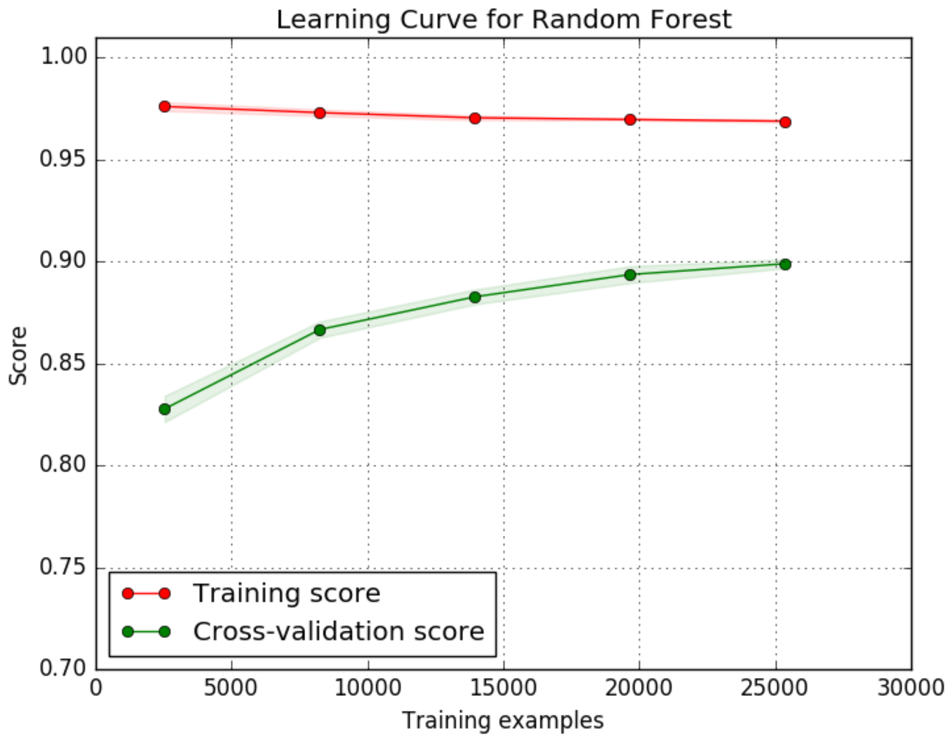

Figure 5.8: Learning curve for default Random Forest

5.6 Multi-layer Perceptron

5.6.1 Principle Procedure

Multi-layer Perceptron, or more commonly called Neural network, is a computational approach based on neural units (perceptrons). Each connected perceptron has a weight associated with it. During the learning phase, the whole network learns by adjusting the weight and thus can predict the correct output tuples, which is the accurate classification in our study. Neural networks involve long training times. However, they usually have a high tolerance of noisy data, and they have the

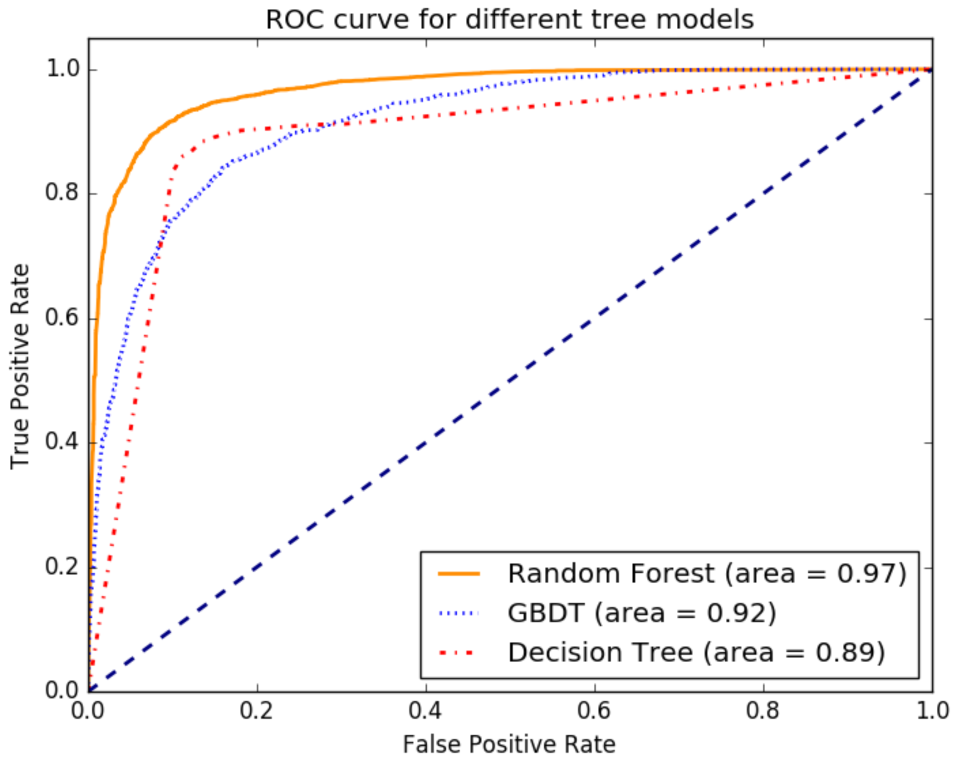

Figure 5.9: ROC curves for different tree models

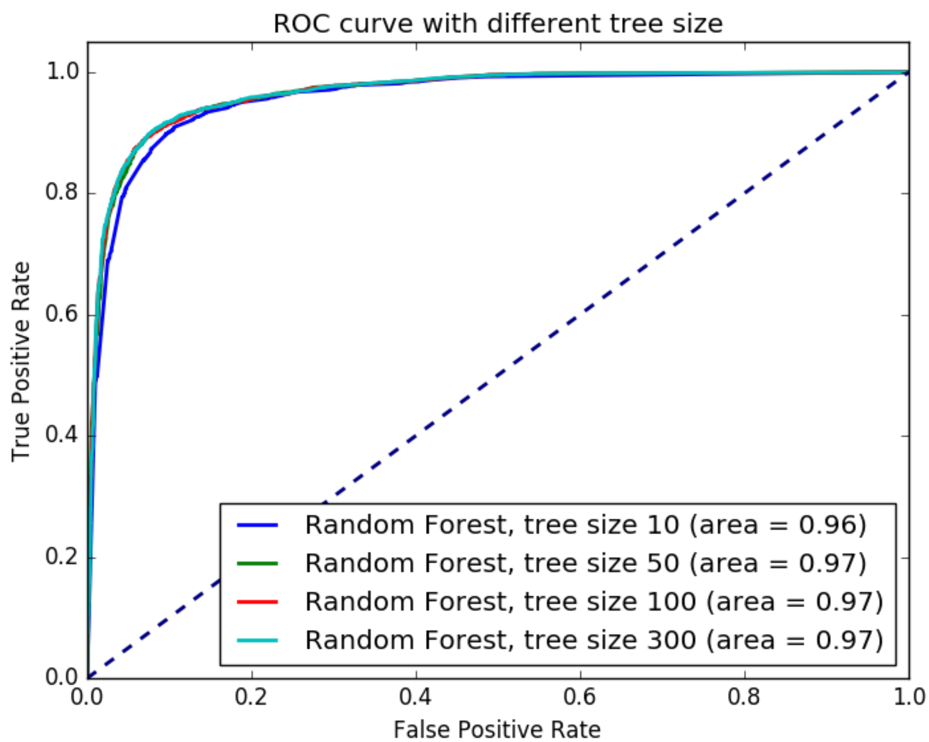

Figure 5.10: ROC curves for different tree size

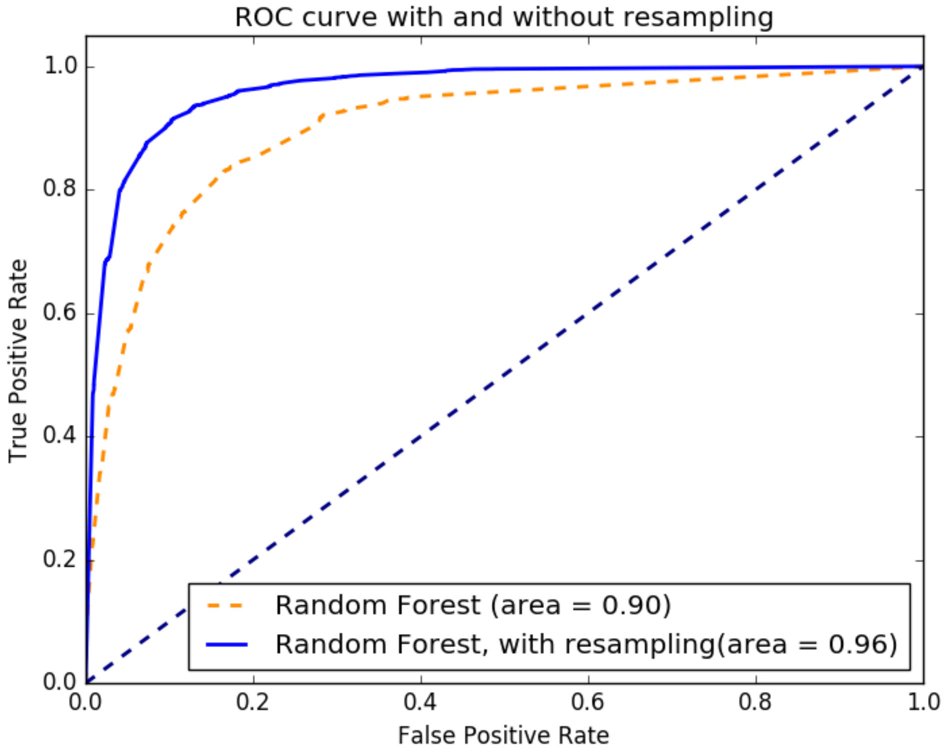

Figure 5.11: ROC curves for with and without resampling

advantage to classify patterns they have not been trained on[61]. This is ideal for us because we have a relatively vague understanding of the relationships between the features.

The perceptron, mentioned above as the basic unit in a neural network, is a supervised learning algorithm of binary classifiers to determine whether an input belongs to a class or another [62]. Each perceptron is a linear classifier, and may have a summation function like the algorithm introduced in Section 5.4. For each training sample j, actual output is:

$$\begin{aligned} y_j &= f(w \cdot x_j) && (5.10) \\ &= f(w_0 x_{j,0} + w_1 x_{j,1} + ... + w_n x_{j,n}) && (5.11) \end{aligned}$$

We use the Heaviside step function as the activation function, as shown below to make a binary classification:

$$f(x) = \begin{cases} 1, & if\ w \cdot x + b > 0 \\ 0, & otherwise \end{cases}$$

For more complicated or non-linearly divisible problems, the perceptrons are connected into a multi-layer feedforward neural network [63, 64]. The perceptrons in each layer are not connected, and each layer is fully connected to the next layer. The first layer has input neurons, sending information to the next layer. The second layer processes information, and finally sends to the last layer of output neurons. The second layer is called hidden layer. The more complex system may have more layers. A neuron's network function $f(x)$ is defined as a composition of other functions $g_i(x)$. Interestingly, $g_i(x)$can further be defined as a composition of $h_i(x)$. We use nonlinear weighted sum to represent the function:

$$f(x) = K\left(\sum_i w_i g_i(x)\right) \tag{5.12}$$

where K is the activation function. Furthermore, $g_i(x)$ follows similar rule, depending upon the output of every $h(x)$. To train such a multi-layer model efficiently, a technique called Backpropagation is invented and commonly used in today's implementations. It repeats a two-phase cycle to training the model until some criteria met. The first step is to calculate the result based on current parameters layer by layer until output. The second step is to compare the current result against target output using a loss function, then propagate backward from output to input to update the parameter according to gradient descent value. A detailed introduction is in [65], here we directly run the model and compare results. It is also indicated in [65] that, with enough layers and enough perceptrons, a neural network can interpret almost any complicated logic, which might outperform any known models.

5.6.2 Result and Analysis

The default model with one hidden layer and total 100 perceptrons returns good accuracy 93% and F-score 90% as shown in figure 5.12, which is quite good already. We are also interested to see how different complication levels affect the result. Not surprisingly, comparing ROCs from models with different hidden layer size and perceptrons number does not show a significant difference as in figure 5.13, even though the performance in accuracy varies from 85% to 94% and F-score ranges from 80% to 90%. This result again proves our conclusion that the logic behind determining an App as malicious is rather straightforward and robust, especially even the default 100-perceptron model outperforms a 40-40-40-40 model by a little bit. We also find the Learning Curves for different models vary a lot, which indicates that tuning parameters for Neural Networks is of particular importance to avoid both underfitting and overfitting.

On the other hand, since it is quite hard to tune a neural network, we want to apply Ensemble techniques to overcome it. What is more, we decided to do ensemble on all three models: Logistic Regression to interpret linear correlation, Random Forest to cover complex logic, and several weak Neural Network models to capture main prediction logic. Figure 5.14 shows the ROC curve for an Ensemble model that contains all three models with default parameters. Unfortunately, the logistic regression performs too bad which lowered the whole performance. Figure 5.15 shows the Ensemble model that only contains Random Forest and Neural Network. This time the Ensemble model takes advantage of both models and performs better than any single one. Overall speaking, we achieved 95.1% accuracy with 90.5% F-score, which is better than other related researches by a lot.

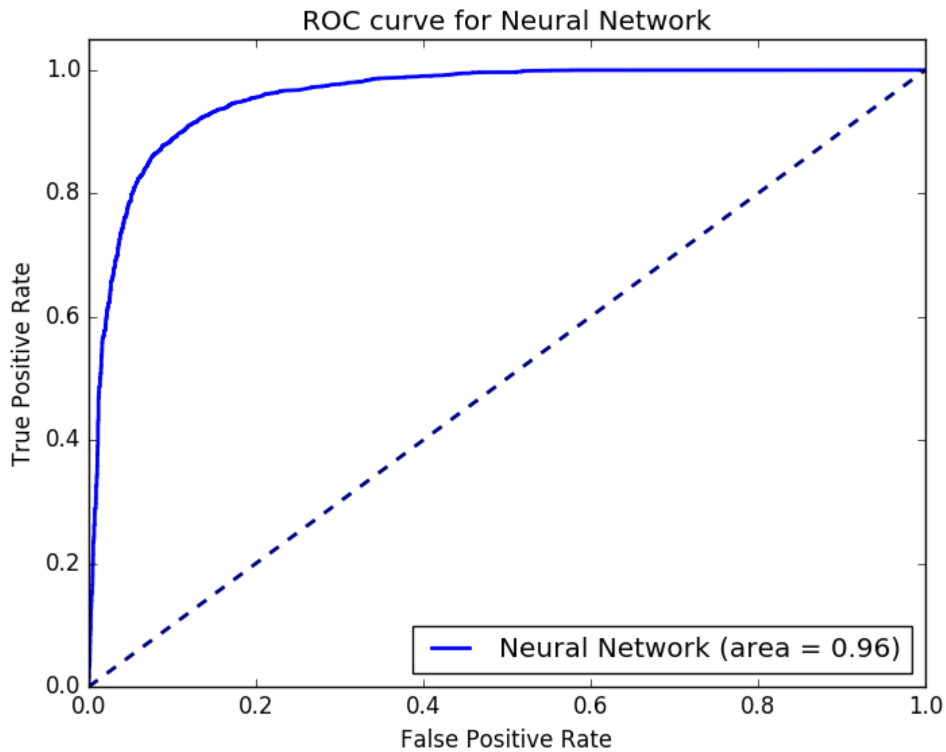

Figure 5.12: ROC curve for default neural network model

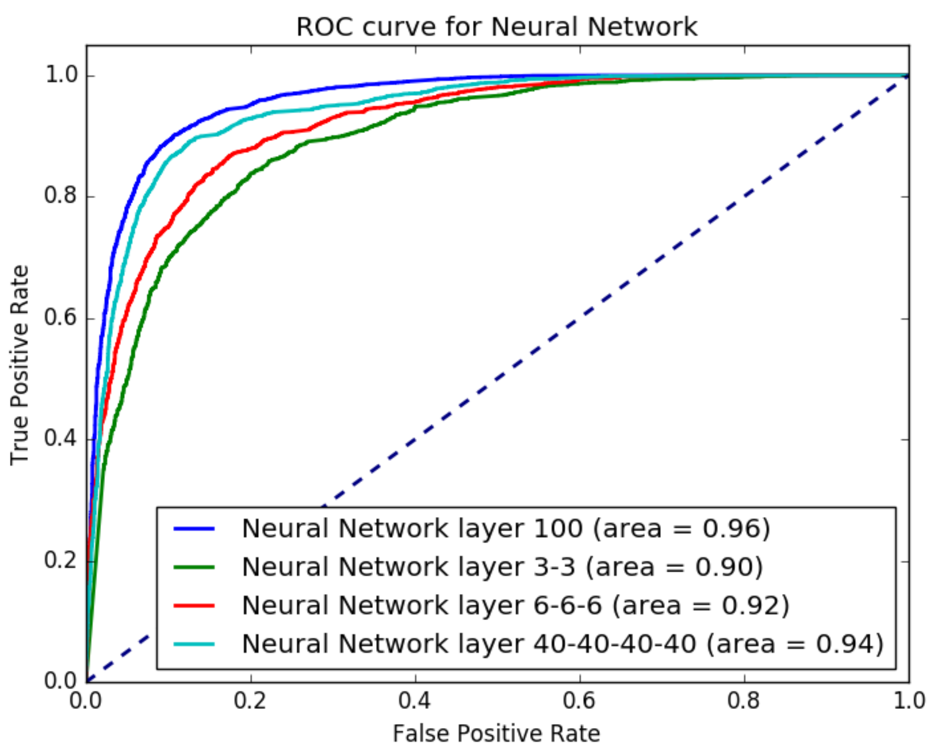

Figure 5.13: ROC curve for different neural network models

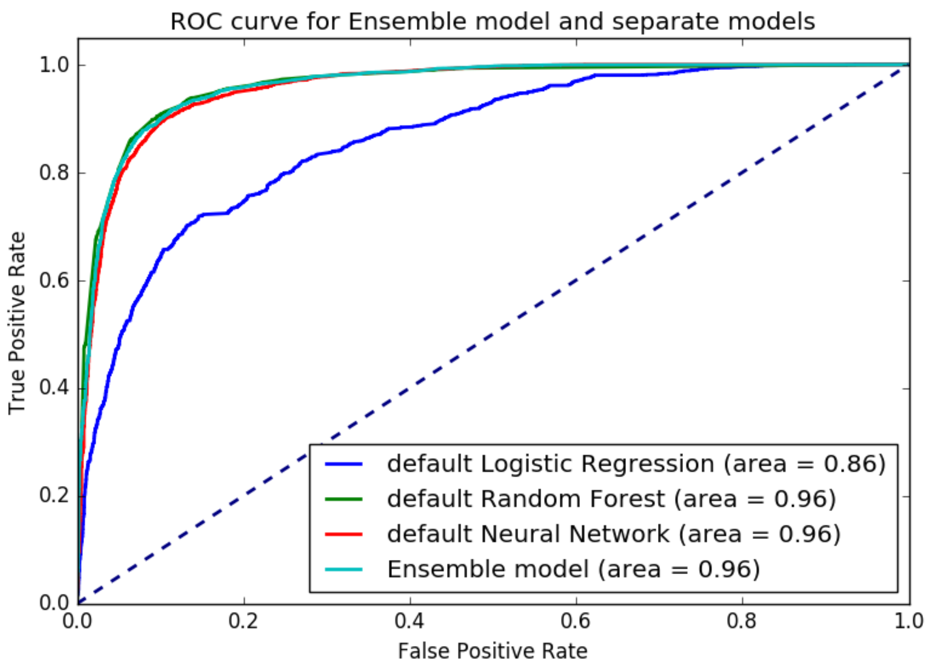

Figure 5.14: ROC curve for Ensemble model

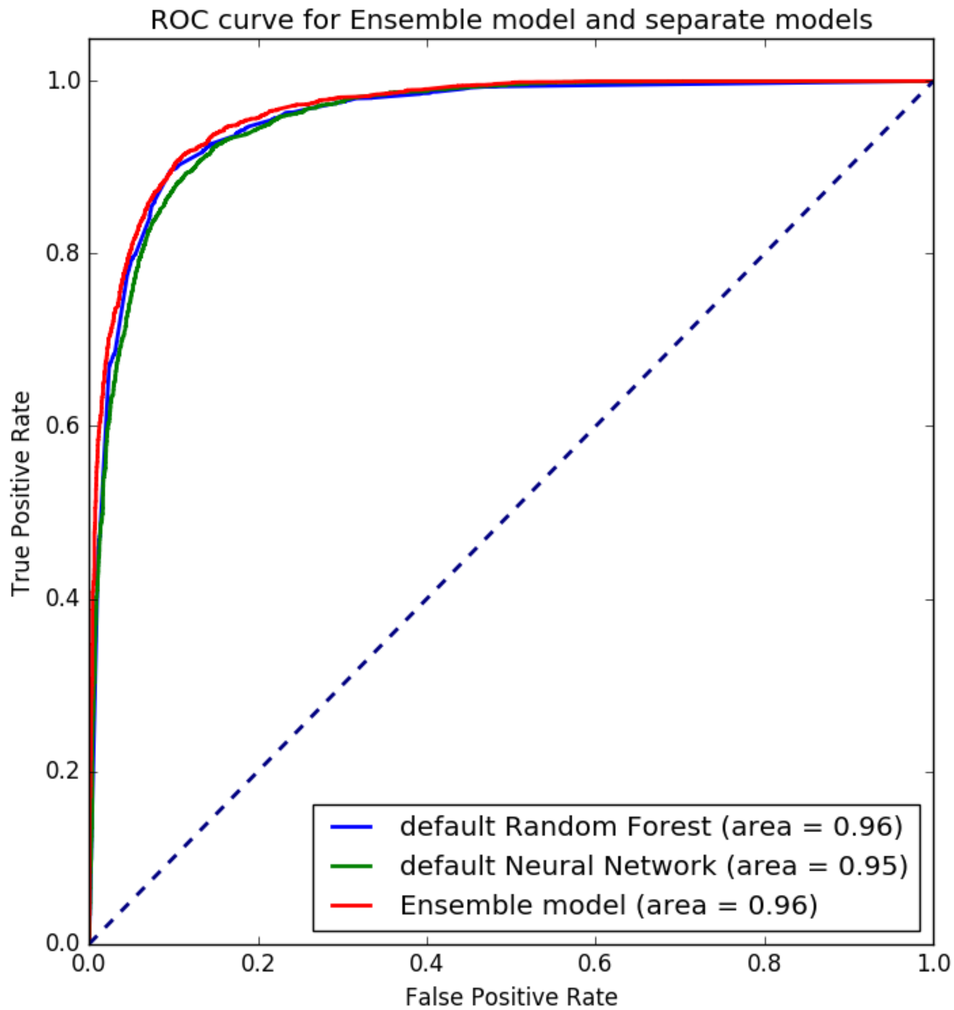

Figure 5.15: ROC curve for Ensemble model - excluding Logistic Regression

CHAPTER 6

Conclusion

Personal Portable Devices (PPD), or more commonly called mobile devices, have significantly brought people's personal lives to a new high level. Smartphone applications or simplified as Apps, provide potential vulnerability for accessing privacy, which is harmful to normal users, and hard to discover. Due to Android's top popularity and open-source nature, it has also been the priority target of malicious Apps over other mobile operating systems. Existing techniques for malware detection have their limitations of either being too heavy-weight or requires Internet connection, and some of them may need to observe the behaviors after they have already happened. In this paper, we presented an Android Permission Model-based technique to detect malware. The main contributions of our work are summarized as below:

- A big data set of malware and benign Apps is collected and decompiled, and features are extracted and aggregated. Android permission model is studied in detail to pre-process and clean the permission data.
- The logical and statistical analysis is performed to first analyze single permission - malware correlation by paired Chi-square test. Another Chi-square test is conducted between permissions to study the internal relations between two permission combinations.
- Principal component analysis is conducted to reduce the dimension of features, help analyze the complexity and understand features better.
- Class-imbalance issue is resolved by resampling the benign Apps' data. It is explained in detail, and later experiments show positive support.

- The Linear model, tree model, neural network and Ensemble model is studied, utilized and analyzed to detect the malware in deep. Experiments are conducted for each model, ROC and Learning curve are also plotted to demonstrate the performance of the models.
- The ensemble is conducted at the end to take advantage from each of the three models, achieving 95.1% accuracy with 90.5% F-score without much overfitting.

Our future work includes:

- Deeper understanding on the logic to predict malware.
- More detailed parameter tuning for separate models.
- More diversity on Ensemble models.
- Consider the practical implementation in order to bring our research into real products.

Bibliography

[1] Wikipedia. Mobile operating system — wikipedia, the free encyclopedia, 2017. [Online; accessed 7-March-2017].

[2] Statista. Number of apps available in leading app stores as of june 2016, 2016. [Online; accessed 7-March-2017].

[3] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.

[4] Renee Shipley. The best anti-malware software of 2017, 2016. [Online; accessed 7-March-2017].

[5] Joe Hindy. 15 best antivirus android apps and anti-malware android apps, 2017. [Online; accessed 7-March-2017].

[6] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, pages 297–300. IEEE, 2010.

[7] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX security symposium*, pages 543–558, 2013.

[8] Ed Burnette. *Hello, Android: introducing Google's mobile development platform*. Pragmatic Bookshelf, 2009.

[9] Kyoochang Jeong and Heejo Lee. Code graph for malware detection. In *Information Networking, 2008. ICOIN 2008. International Conference on*, pages 1–5. IEEE, 2008.

[10] Haoran Guo, Jianmin Pang, Yichi Zhang, Feng Yue, and Rongcai Zhao. Hero: A novel malware detection framework based on binary translation. In *Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, volume 1, pages 411–415. IEEE, 2010.

[11] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of mali-

cious behavior. In *Proceedings of the 1st India software engineering conference*, pages 5–14. ACM, 2008.

[12] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333. IEEE, 2010.

[13] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[14] Min Zheng, Mingshen Sun, and John CS Lui. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 163–171. IEEE, 2013.

[15] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[16] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[17] android.com. Permissions, 2017. [Online; accessed 7-March-2017].

[18] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[19] android.com. Manifest.permission, 2017. [Online; accessed 7-March-2017].

[20] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1053–1067, 2014.

[21] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[22] Xiangyu Liu, Zhe Zhou, Wenrui Diao, Zhou Li, and Kehuan Zhang. An empirical study on android for saving non-shared data on public storage. In *ICT Systems Security and Privacy*

Protection, pages 542–556. Springer, 2015.

[23] David Barrera, H Güneş Kayacik, Paul C van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010.

[24] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[25] V Babu Rajesh, Phaninder Reddy, P Himanshu, and Mahesh U Patil. Droidswan: Detecting malicious android applications based on static feature analysis. *Computer Science & Information Technology*, page 163.

[26] Zarni Aung and Win Zaw. Permission-based android malware detection. *International Journal of Scientific and Technology Research*, 2(3):228–234, 2013.

[27] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[28] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *Intelligence and security informatics conference (eisic), 2012 european*, pages 141–147. IEEE, 2012.

[29] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[30] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.

[31] B Alll and C Tumbleson. Dex2jar: Tools to work with android. dex and java. class fil es.

[32] R Winsniewski. Apktool: a tool for reverse engineering android apk files. *URL: https://ibotpeaches. github. io/Apktool/(vi sited on 07/27/2016)*, 2012.

[33] Java Decompiler. Jd-gui.

[34] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. Curbing android permission creep. In *Proceedings of the Web*, volume 2, pages 91–96, 2011.

[35] Wikipedia. Chi-squared test — wikipedia, the free encyclopedia, 2016. [Online; accessed

7-March-2017].

[36] Jerome H Friedman. Data mining and statistics: What's the connection? *Computing Science and Statistics*, 29(1):3–9, 1998.

[37] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[38] Ron Kohavi and Foster Provost. Glossary of terms. *Machine Learning*, 30(2-3):271–274, 1998.

[39] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[40] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[41] Wikipedia. Principal component analysis — wikipedia, the free encyclopedia, 2017. [Online; accessed 7-March-2017].

[42] Tian-Yu Liu. Easyensemble and feature selection for imbalance data sets. In *Bioinformatics, Systems Biology and Intelligent Computing, 2009. IJCBS'09. International Joint Conference on*, pages 517–520. IEEE, 2009.

[43] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[44] Lior Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1-2):1–39, 2010.

[45] Francisco Pereira, Tom Mitchell, and Matthew Botvinick. Machine learning classifiers and fmri: a tutorial overview. *Neuroimage*, 45(1):S199–S209, 2009.

[46] Peter Bühlmann. Bagging, boosting and ensemble methods. In *Handbook of Computational Statistics*, pages 985–1022. Springer, 2012.

[47] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[48] Michael Collins, Robert E Schapire, and Yoram Singer. Logistic regression, adaboost and bregman distances. *Machine Learning*, 48(1-3):253–285, 2002.

[49] Jane Elith, John R Leathwick, and Trevor Hastie. A working guide to boosted regression trees. *Journal of Animal Ecology*, 77(4):802–813, 2008.

[50] MW Browne. Predictive validity of a linear regression equation. *British Journal of Mathe-*

matical and Statistical Psychology, 28(1):79–87, 1975.

[51] Strother H Walker and David B Duncan. Estimation of the probability of an event as a function of several independent variables. *Biometrika*, 54(1-2):167–179, 1967.

[52] Lior Rokach and Oded Maimon. *Data mining with decision trees: theory and applications*. World scientific, 2014.

[53] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[54] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *icml*, volume 99, pages 124–133, 1999.

[55] Mark A Friedl and Carla E Brodley. Decision tree classification of land cover from remotely sensed data. *Remote sensing of environment*, 61(3):399–409, 1997.

[56] Robert M Gray. *Entropy and information theory*. Springer Science & Business Media, 2011.

[57] Tin Kam Ho. Random decision forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pages 278–282. IEEE, 1995.

[58] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8):832–844, 1998.

[59] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[60] David Cossock and Tong Zhang. Statistical analysis of bayes optimal subset ranking. *IEEE Transactions on Information Theory*, 54(11):5140–5154, 2008.

[61] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[62] Yoav Freund and Robert E Schapire. Large margin classification using the perceptron algorithm. *Machine learning*, 37(3):277–296, 1999.

[63] Teuvo Kohonen. An introduction to neural computing. *Neural networks*, 1(1):3–16, 1988.

[64] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[65] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.