The Pennsylvania State University
The Graduate School

A MACHINE LEARNING BASED APPROACH TO APP RATING MANIPULATION DETECTION

A Thesis in
Computer Science and Engineering
by
Yang Song

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2014

The thesis of Yang Song was reviewed and approved* by the following:

Sencun Zhu
Associate Professor of Department of Computer Science and Engineering & College of Information Science and Technology
Thesis Advisor

Wang-Chien Lee
Associate Professor of Department of Computer Science and Engineering

Lee D. Coraor
Associate Professor of Department of Computer Science and Engineering
Graduate Program Chair of Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

In order to promote apps in mobile app stores, for malicious developers and users, manipulating average rating is a popular and feasible way. In this thesis, we propose a two-phase machine learning approach to detect the app rating manipulation. We give the definitions of abused app, malicious user and collusion group, and characteristics. In the first learning phase, we generate feature ranks for different app stores and find that top features match the characteristics of abused apps and malicious users. In the second learning phase, we choose top N features and train our models for each app store. With cross-validation, our training models achieve 85% f-score. We also use our training models to discover new suspicious apps from our data set and evaluate them with two criteria. Finally, we conduct some analysis based on the suspicious apps classified by our training models and some interesting results are discovered.

Keywords: machine learning, app rating manipulation, app store

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my deep gratitude to my advisor Dr. Sencun Zhu, who provides me invaluable help and guidance on my research. I would also like to thank Ph.D candidate Mr. Zhen Xie, for his advice and assistance in keeping my progress on schedule. His tool also expedites my research progress. My grateful thanks are also extended to Dr. Wang-Chien Lee for his valuable support.

Special thanks should be given to Dr. Lee Giles, who encourages me to learn Information Retrieval, Machine Learning and Data Mining.

Dedication

I dedicate this thesis to my parents and my fiancé, Yingying Hu.

Chapter 1

Introduction

The current mobile application markets such as iTunes App Store, Google Play, Amazon App Store, Windows Phone App Store, etc. provide a very convenient and efficient way to distribute mobile apps. In 2013, smartphone shipment volume reached 1.004 billion in total[1] and Android, Apple and Microsoft together gain 95.7% in smartphone platforms market share[2]. In each app store, there are hundreds of thousands apps(table 2.2 lists these numbers). In Apple's 2014 first quarter results[3], their customers have downloaded a total of 65 million apps and spent $4.7 billion on iTunes, with $2 billion being paid to developers. All these numbers show that if developers release very popular apps, they could make huge amount of profit from these apps. From computer security's perspective, those app stores become the targets of attackers for the two following reasons: 1) app stores encourage users to review apps. High average rating means app's quality is high and most users choose to download higher rating apps when they have multiple options; 2) App store providers will show various app ranking charts in the front page and the higher ranking app has, the more attentions and more downloads it will have. App store providers' ranking algorithms[4] take review rating as an

[1] https://www.idc.com/getdoc.jsp?containerId=prUS24645514

[2] http://www.comscore.com/Insights/Press_Releases/2014/1/comScore_Reports_November_2013_US_Smartphone_Subscriber_Market_Share

[3] http://www.apple.com/pr/library/2014/01/27Apple-Reports-First-Quarter-Results.html

[4] http://www.insidemobileapps.com/2012/06/29/how-rating-affects-ranking-in-search-results-and-top-charts-across-platforms/

important factor. We find some companies[5] providing app promoting services and some of them even claim that they could keep the ranks that developers want for about 24 hours.

Purifying app stores becomes app store providers' one of the primary interests. However, detecting app rating manipulation is difficult due to the following reasons: 1) massive number of accounts, reviews and apps makes it impossible to manually investigate the whole app store. Even it's possible to hire many people to manually investigate apps, discriminating malicious users from normal users is nearly impossible by reading review content because rating reflects user's opinion and opinion is always biased. 2) many researchers have studied rating manipulation on online shopping websites. But compared with traditional products, mobile apps are either more accessible and their data are more noisy. Apps are more accessible because app stores are available in many countries and developers could publish their apps world widely just by one click. For attackers, they can switch countries in app stores and attack apps across regions. Apps' data are more noisy because apps have to be run on the top of hardware(smartphone) and OS. If hardware or OS upgrades, apps have to be optimized for that updates. Failed to do so may result in lowering average rating or app crash. More importantly, number of downloads and reviews is sensitive to app price. For traditional online shopping websites, 50% percentage discount is very rare, and even so, customers still have to pay. If a paid app drops its price to free, it's very likely that much more users will download it. Of course, advertisements will introduce more noises to app's data. But ads for app have more direct effects. Facebook app reserves some space for app ads. Once users see those ads, it only takes couple seconds to complete the whole transactions when they decide to download that app.

Those difficulties and extremely noisy data lead us to seek machine learning for help. A big advantage of machine learning technique is that it provides an automatic way to figure out a model that human being is not able to conceive, and we can use this model to discover abused apps. By analyzing the data we crawled from app stores and considering characteristics of malicious user and abused app, we decide to use random forest as our learning algorithm.

Our supervised machine learning approach aims to detect abused apps directly

[5] `http://www.itunesrank.com/`, `http://www.itunesrank.com/`

instead of detecting attackers, which also can be used to discover abused apps indirectly. That's because: 1) detecting attackers is difficult due to the reasons we mentioned above; 2) apps that have been rated by attackers does not always mean they have been abused. More information is needed to confirm this relation. To build a training set, we use a strong-constraint algorithm[1] to find an initial group of abused apps for us and, meanwhile, we select benign apps manually. We use 2-step learning procedure to finalize our training models for each app store. By using cross-validation, our training models could reach 85% f-score given the fact that data are extremely noisy. We take a further step and use our training models to discover more apps from the data that are not included in training set. About 5% suspicious apps are discovered from each app store and the results are justified even we lack evidences to prove that they are abused apps.

This thesis has the following contributions:

1. We characterize abused app, malicious user and collusion group they form.

2. A multi-threaded auto-scalable crawled is introduced. It could crawl iTunes App Store, Amazon App Store and Windows Phone App Store readily.

3. We propose a novel machine learning based method to detect abused apps directly, bypassing the difficulties such discriminating attackers from normal users, data noises, etc..

4. We collect 55 features and give feature rankings for each app store. We find that top features match our definitions and characteristics of abused app, malicious user and collusion group. Not only these feature rankings are used to further improve our training model performance, but also, other researchers who want to use different learning algorithms to detect abused apps could consider using these rankings directly.

5. We analyze abused apps discovered by our training models and some interesting results are found. Other researches who want to do related research may find those results useful and even could consider transforming them as new features.

In Chapter 2, we will elaborately discuss our crawler and the data we have gathered with it. Chapter 3 emphasizes on defining, characterizing abused app,

malicious user and collusion groups. How we pick our training set is also discussed in this chapter. All 55 features are presented in Chapter 4. The machine learning approach is explained in Chapter 5. Readers also will see some interesting data analyses in Chapter 5.

Chapter 2

Data Collection

Data is the prerequisite for any machine learning research. In this chapter, we will discuss how we collect(crawl) data from various app stores in different countries. In Section 2.1, we will give an architecture overview for our app store crawler. In Section 2.2, we will discuss what app, comment and user related information we have.

2.1 Crawler Overview

For traditional or general-purpose web crawler[2], crawler is mainly responsible for exploring the webpages from initial given websites and storing the webpages it crawled for further analysis and use. Our specialized app store crawler has two following differences from general-purpose crawlers:

1. Instead of storing all the web pages that are crawled directly, we parse them on-the-fly and store only extracted information in order to minimize storage requirements.

2. We crawl three kinds of webpages in app stores: app overview pages, comment pages and user profile pages. App overview page usually contains number of ratings, average ratings, general description, app permissions, etc.. Comment page contains the names and IDs of reviewers and comments' dates and contents. iTunes(Apple App Store) also provides the corresponding app version for each comment. User profile page contains all comments

that each user leaves. For general purpose crawlers, they don't have worry about crawling different kinds of webpages(they actually do, but not with such granularity). Therefore, they only have homogenous crawling engines, which are used to send HTTP requests and download webpages. However, our crawler has four heterogenous engines as they have to parse different kinds of web pages.

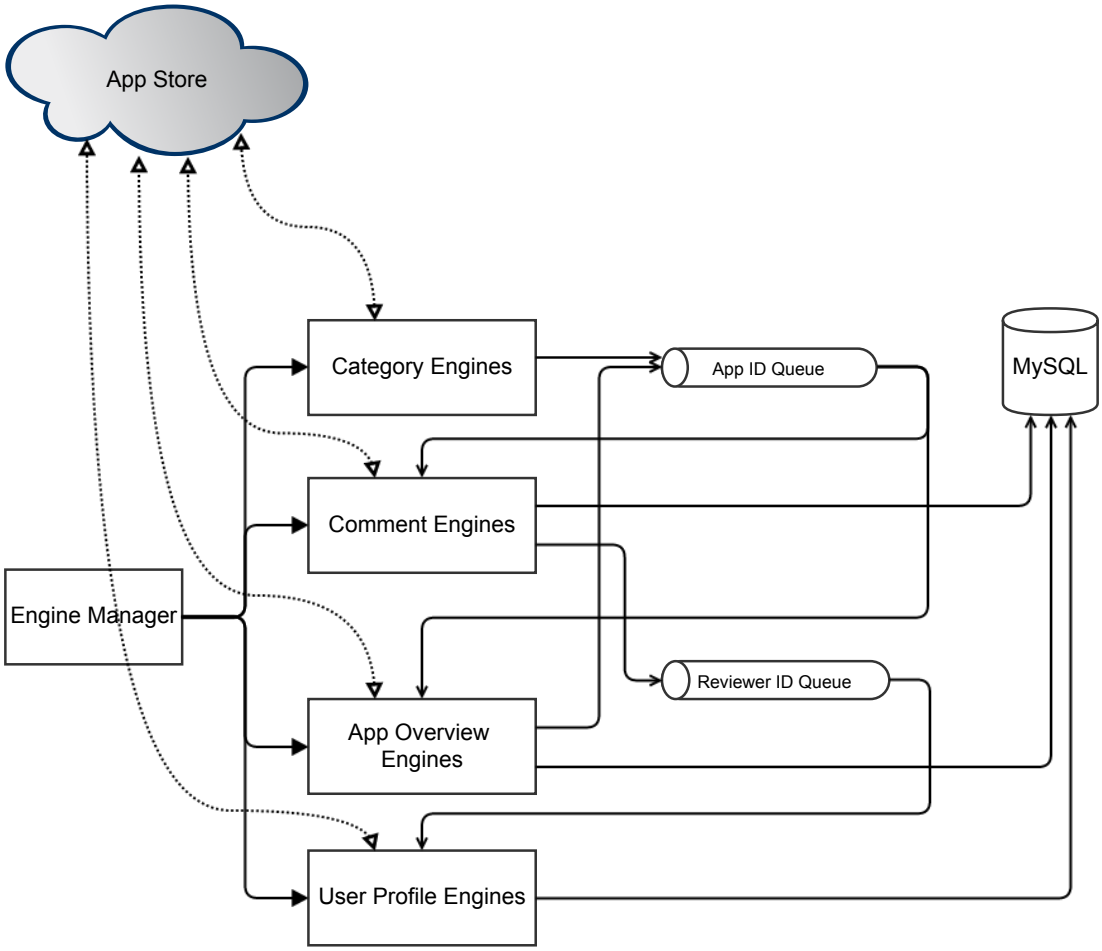

Figure 2.1: Architecture of App Store Crawler

Figure 2.1 depicts the architecture of our app store crawler. This crawler is multi-threaded and is able to scale automatically. We will use some paragraphs explaining each component in detail.

App ID Queue: a concurrent queue that receives app IDs extracted by cate-

gory engines and app overview engines. App overview engines and comment engines will fetch one app ID from it each time. App ID queue conceptually contains app IDs that have been discovered but their overview data and comments have not been crawled. To ensure that one app will only be crawled once, we also use a set(not shown in figure 2.1) to store all the app IDs that have been explored or crawled.

Reviewer ID Queue: a concurrent queue that receives reviewer(user) IDs extracted by comment engines. User profile engines will fetch one reviewer ID from it each time. Reviewer ID queue conceptually contains reviewer IDs that have been discovered but the corresponding user profile pages have not been crawled and parsed. To ensure that one reviewer can only be crawled once, we also use a set(not shown in figure 2.1) to store all reviewer IDs that have been explored or crawled.

Category Engine: to ensure accessibility and better app classification, app stores normally classify hundreds of thousands of apps into several categories. If we only use app store front page as our start page, we will encounter two problems: (1) usually app stores' front pages only contain limited number of apps; (2) some app stores, such as Amazon and Windows Phone app stores, are not separated from other webpages. Amazon app store's front page links to pages of Amazon Kindle devices and other products. Windows phone app store's front page links to pages of windows phone devices and windows phone news. To mitigate these two problems, we manually generated lists containing category webpage links of iTunes, Amazon App Store and Windows Phone App Store. Category engines will iterate these lists to extract app IDs from category web pages and insert them to app ID queue. Our category engine is smart enough to do pagination until the current web page is the last page of this category.

Comment Engine: is responsible for crawling comment pages, parsing comment information (date, title, content, comment ID and reviewer ID of comment) and inserting them into database. There could be several comment engines fetching app IDs from app ID queue concurrently. After extracting

reviewer IDs of comments, comment engine will enqueue those reviewer IDs to reviewer ID queue.

App Overviewer Engine: is responsible for crawling app overview pages, parsing app related data(name, developer, description, price, number of each rating, total number of ratings, etc.) and inserting those data into database. There could be several app overview engines fetching app IDs from app ID queue concurrently. Amazon app store is famous for their recommendation system. App overview engines will extract new app IDs from recommendation division in the app overview webpage and insert them into app ID queue after performing duplicity check with app ID set.

User Profile Engine: is responsible for crawling user profile pages, parsing user related data(date, title and content of all the comments he/she leaves) and inserting them into database. Just like app overview engine and comment engine, user profile engine could be multi-threaded as well.

Engine Manager: is the control center. Its job is to do "orchestration" on all the engines. Before starting all engines, engine manager will read a configuration file, which explicitly specifies the upper bounds of number of engines that can concurrently run.

As mentioned before, our app store crawler is able to scale automatically by inspecting the sizes of app ID queue and reviewer ID queue. If the size of queue is below a threshold, engine manager will broadcast a *shut_down* event carrying the thread ID that is supposed to be terminated.

Apart from automatic scaling, the number of threads in App Store Crawler can also be configured manually while the program is still running. To realize this feature, we use a scheduler that can reload the configuration file every 120 seconds. Once engine manager picks up the changes in the configuration files, it will create or shut down threads, depending on which kind of changes user makes.

We also realize that network connection is not as reliable as we expect, especially when we are crawling an app store whose servers reside outside of U.S. When

our crawler experiences network issues, it will retry the failed links 5 more times. If failure continues, engines will enqueue app ID or reviewer ID back to its queues and wait 2 minutes for network recovery.

Performance is evaluated by results. In Section 2.2, we will describe the data that we have crawled with our app store crawler.

2.2 Data Set Overview

Data is always the prerequisite for any machine learning research. Theoretically, the more data we have, the more confidence we have for our training model and prediction. We choose three major app stores — iTunes App Store, Amazon App Store and Windows Phone App Store – to crawl and analyze the data they have.

As shown in figure 2.1, we store all the data we have crawled into MySQL database. Table 2.1 describes how these data are organized and stored in database.

Table 2.1: Tables in MySQL Database

(a) AppData Table

app_id(primary_key)
app_name
developer_name
average_rating
total_raters
1star_num
2star_num
3star_num
4star_num
5star_num
price

(b) Comment Table

id(primary_key)
app_id
reviewer_id
date
app_version
rating
comment
comment_title
helpfulness_ratio
helpfulness_agree
helpfulness_total

(c) Reviewer Table

reviewer_id(primary_key)
app_ids
review_ratings
review_dates
size
app_versions

We also insert other information, such as app descriptions, app URL links, app permissions, etc. into database for feature research. But for now, that's all we need. We will use some space to explain some field names as we will mention those names repeatedly in Chapter 3.

AppData(table 2.1a)

total_raters: number of ratings. However, it is different from number of total comments. Usually the number of total raters is much larger than the number of total comments because many users only give ratings without comments.

star_num: number of ratings for specific rating level. Currently, all the app stores use a spectrum ranging from 1 star to 5 star to evalute app qualities. Just like total_rater, the number of ratings for a rating level usually is much larger than the number of comments for that level, just because lots of users would like to leave ratings without comments.

price: different countries use different currencies, but we don't bother to convert them since we only consider app is free or paid when we are generating features.

Comment(table 2.1b)

app_version: many apps have different versions. App qualities may vary from one version to another. In iTunes App Store, comments do not aim to apps directly, instead they aim to app versions. However, Amazon App Store and Windows Phone App Store do not expose this data to outsiders.

helpfulness: iTunes App Store and Amazon App Store allow users to agree or disagree other users' comments. This is one of many ways that they use to rank comments. For example, *2 of 5 people think this comment is helpful* means $helpfulness\_ratio = 0.4$, $helpfulness\_agree = 2$ and $helpfulness\_total = 5$.

Reviewer(table 2.1c): reviewer table is a redundant table just for faster looking up. Each row stores all the comments left by one reviewer.

We crawled iTunes App Store, Amazon App Store and Windows Phone App Store. None of them are localized — iTunes App Store spans more than 100 countries [1]. Amazon App Store supports more than 15 countries [2]. Windows

[1]Countries iTunes App Store supports: http://www.apple.com/choose-your-country/
[2]Countries Amazon App Store supports: http://www.amazon.com/gp/feature.html?docId=487250

Phone App Store supports more than 100 countries[3]. For each store, we choose U.S, China and U.K three countries to crawl, as those three countries have large population and mobile device market. Table 2.2 lists the data size we have for these three app stores each in U.S, China and U.K.

Table 2.2: Data Size Table

App store	Apps	Comments	Reviewers	Total Apps	Completeness[4]
iTunes(U.S)	23,616	18,925,438	10,328,118	1 million[5]	2.36%
iTunes(China)	21,831	9,320,807	5,568,424	1 million[6]	2.18%
iTunes(U.K)	10,579	11,761,493	6,413,303	1 million[7]	1.06%
Amazon(U.S)	128,979	1,817,808	736,666	161,181[8]	80.02%
Amazon(China)	18,697	4,023	2,086	22,695[8]	82.38%
Amazon(UK)	122,007	134,991	70,556	151,217[8]	80.58%
Windows(U.S)	57,258	393,201	N/A[9]	160,000[10]	35.79%
Windows(China)	24,064	109,237	N/A[9]	160,000[10]	15.04%
Windows(U.K)	35,852	197,021	N/A[9]	160,000[10]	22.41%

2.3 Summary

In this chapter, we briefly discussed our app store crawler and the data set we have crawled with it. Our crawler is a highly specialized crawler with multi-thread and automatic scaling features.

We use our app store crawler to crawl three major app stores — iTunes App Store, Amazon App Store and Windows Phone App Store — in U.S, China and U.K three countries. In following chapters, we will discuss how we define app

[3]Countries Windows Phone App Store supports: http://www.windowsphone.com/en-us/markets

[4]Completeness is approximated percentage of apps we have crawled in app store.

[5]This number is anounced by Apple, Inc in iPad event held in Oct, 2013.

[6]No data can be found to show the actual size of iTunes App Store in China. We assume that the number is close to 1 million.

[7]No data can be found to show the actual size of iTunes App Store in China. We assume that the number is close to 1 million.

[8]As of Jan, 22, 2014

[9]Windows App Store does not provide user profile pages.

[10]This number is claimed by Microsoft as of June, 21, 2013. http://www.wpcentral.com/160000-apps-microsoft-windows-phone-store-numbers

whose rating has been manipulated(abused app) and how we use machine learning to distinguish abused apps from normal ones.

Chapter 3

Training Set

In this chapter, we will talk about our training set. In Section 3.1, definitions of abused app and malicious user are given and the techniques we use to distinguish abused apps from normal ones will be explained in detail. In Section 3.3, we will discuss how we build our training set. To simplify our terms, in this dissertation, "user" only refers to user who rates apps; "reviewer" only refers to user who both rates apps and leaves comments.

3.1 Abused App and Malicious User

3.1.1 Definitions

For any app stores, the motivation of establishing online rating system is to help users know the *true quality* of apps. Here we give our definition for *true quality*.

Definition 3.1. *Let the number of total raters be N and $\mathbf{r} = (r_1, r_2, \ldots, r_N)$ is a vector that contains ratings of all raters.* ***True quality*** *τ is the expected value of N users' ratings, or*

$$\tau = \frac{\sum_{i=1}^{N} r_i}{N}, \tag{3.1}$$

where $N \to \infty$.

Intuitively, abused app means an app's average rating has been manipulated and rating could not correctly reflect *true quality*. There is no direct way to know

app's *true quality*. Indirectly, as largely adopted today, app stores expect the large number of users rate apps and leave comments and hopefully, the average rating could be as close to app's *true quality* as possible. From individual user's perceptive, there are two problems:

1. Evaluating user experience with number is hard and inaccurate. No global standard defines the necessary conditions of a 5-star app, neither no global standard defines the necessary conditions of a 1-star app.

2. Individually, users are all biased. 1) Different users have different perspectives. User $\mathcal{A}$ may focus on user interface, while user $\mathcal{B}$ may focus on the stability and security. It's very likely that one app is perfect in user interface, and poor in security; 2) Experience plays an important role. A user who has used more than 100 apps may not think Skype is the best app he has ever used. But for intro-level user, he might think Skype *is* the best app; 3) Apps usually support different versions of OS and hardware. An app could run perfectly on OS version 2.0 but crash on OS version 3.0.

Therefore, it's not safe to assume that average rating could accurately evaluate app's *true* quality. But in some extends, it can correctly *reflects* the quality of app. It's not likely that an app that is often crashed on any supported devices could gain 5-star average rating. Another way to look at this problem is that *true* quality of one app is the expected value of all its users' ratings, including users that will rate this app in the future(theoretically, infinite number of users could rate this app) and each user in the data we have crawled is just one random variable that could reflects the expected value more or less. According to Hoeffding inequality[3], the larger number of ratings, the closer this app's average rating will be to its *true quality*.

For this problem, Hoeffding inequality states that for any sample size N and vector of independent ratings $\Lambda = (\lambda_1, \lambda_2, \ldots, \lambda_N)$,

$$P[|E(\Lambda) - \tau| > \delta] \leq 2e^{-2\delta^2 N}, \tag{3.2}$$

where δ is the difference between average rating $E(\Lambda)$ and app's *true quality* τ.

To confidently ensure that δ is very small, the number of ratings N has to be relatively large. Therefore, we have the following theorem:

Theorem 3.1. *For a relatively large number of ratings N and each rating is independent from others, average rating $\sigma \approx \tau$, where τ is an app's true quality.*

Rating is time sensitive. On the one hand, rating will change as app's *true* quality changes. Version changes, OS updates and device update usually lead to quality changes. On the other hand, in some cases, even app's *true quality* remains the same, its rating will change if similar apps come out later but provide more features and better user interface. Here, we give the definition of environment factor:

Definition 3.2. ***Environment factor*** *is the factor that could affect users' ratings for an app but without any intentions to manipulate that app's average rating maliciously. Those factors include but are not restricted to version update, OS update, device update and release of similar apps.*

According to the discussion above, we give our definition of abused app:

Definition 3.3. *Let an app's true quality be τ and its average rating is σ. During a period of time t, when no environment factors present, if $|\tau - \sigma| > \epsilon$, where ϵ is deviation constant, then we say this app is* ***abused app****.*

3.1.2 Characterisitcs of Abused App and Malicious User

As we mentioned in Section 3.1, it's normal that users are biased. However, when a large amount of benign users rate one app, the average rating will be close to *true* quality. The definitions of benign user and malicious user are:

Definition 3.4. *A user who rates apps only based on his/her own experience is* ***benign user****.*

Definition 3.5. *A user is* ***malicious user*** *if he/she rates apps not based on his/her own experience and his/her rating deviates from the true quality of app by η, where $|\eta| > 0$.*

To effectively manipulate the ratings, it has to be many malicious users appearing in a relatively short period of time. If only a small number of malicious users rate the app, its average rating will not change abruptly. Theoretically, presence of extremely biased ratings in a very short period time is not a necessary condition for rating manipulation. However, by observation, we find that it is a normal case for rating manipulation. Figure 3.1 plots the numbers of each stars of the app with app ID 445798230 and version 2.1. What make this app suspicious(we later confirmed that this app is abused) are: 1) there is a huge spike of number of 5-star rating in week 2; 2) The spike only occurs on 5-star rating.

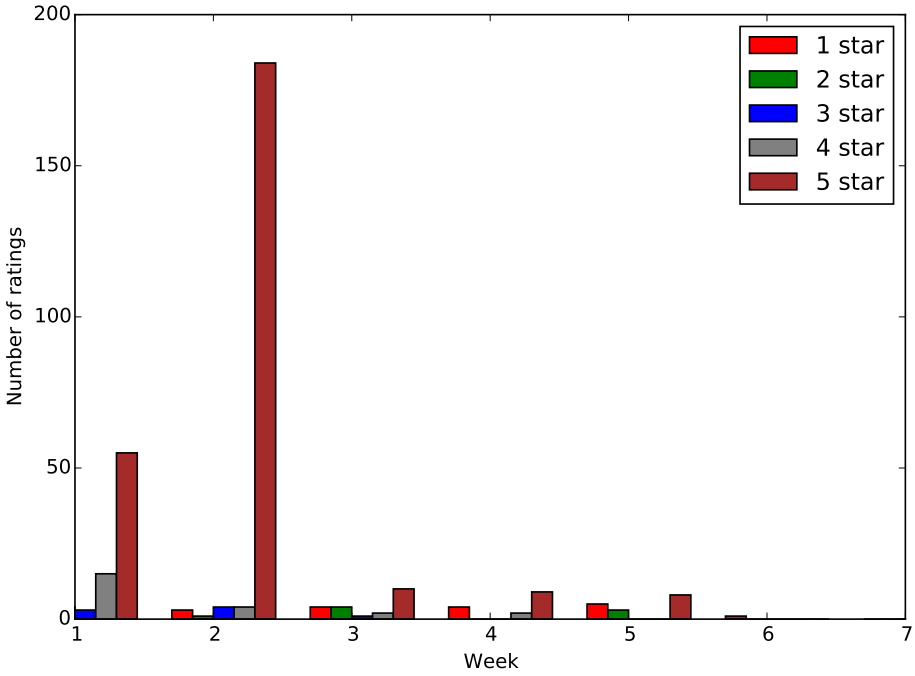

Figure 3.1: Number of Each Star (App ID: 399363156, Version: 2.1)

Mohammad, et al.[4] proposes the idea of collusion group in rating manipulation. For app's rating manipulation, [1][5] modifies the definition of collusion group and gives the following definition:

Definition 3.6. ***Collusion group*** *is an attacking group where all the members are malicious users and there exist a set of apps whose ratings have been manipulated by some of them. To be more generic, collusion group is a set of adjacent*

subgroups where each subgroup works as an unit to attack an app each time. To be considered as adjacent, two subgroups must share some raters.

Essentially, one collusion group is formed by several biclique communities[6]. It's shown that the existence of collusion group is one necessary condition for rating manipulation. Another feature of collusion group is that the rating histories of group members are similar, which means group members usually rate apps on the same day or in the same week. This feature conceptually matches the conditions of app rating manipulation — large number of malicious users and in a short period of time.

According to Definition 3.3, 3.6 and Theorem 3.1, we could have this theorem:

Theorem 3.2. *During a period of time t, an app is **abused app** if it satisfies all the following conditions:*

1. *No environment factors present.*
2. *There exists at least one collusion group.*
3. *Number of ratings N is relatively large.*
4. *Let the current average rating be $\hat{\sigma}$, and average rating without the effect of collusion group be τ. $|\hat{\sigma} - \tau| > \epsilon$.*

In fact, we are capable of estimating N by using Hoeffding inequality. Assume that we want to be sure that δ is 0.1, then we could have the following calculation:

$$P[|E(\Lambda) - \tau| > 0.1] \leq 2e^{-2(0.1)^2 N} \tag{3.3}$$

When $N = 100$, $P[|E(\Lambda) - \tau| \leq 0.1] \geq 0.7293$. When $N = 200$, $P[|E(\Lambda) - \tau| \leq 0.1] \geq 0.9634$. It means, for an app, if the number of total raters is larger than 100, we have more than 72.93% confidence to claim that its current average rating is close to its true quality; if the number of total raters is larger than 200, we have more than 96.34% confidence to claim that its current average rating is close to its true quality, assuming current average rating has not been manipulated.

Note that one of Hoeffding inequality's assumptions is ratings are independent from each other. If there are some collusion groups existing, theorem 3.1 will not hold. But consider these three situations:

1. If no collusion group exists, we are safe to assume that all the ratings are independent(it's possible that only few ratings are not). In this case, given a large number of ratings, we could precisely estimate τ;

2. If collusion groups exist and the number of ratings is small, we cannot even confidently estimate τ. Therefore, we should not label this app as an abused one;

3. If collusion groups exist and the number of ratings is large, practically, compared with number of ratings, size of collusion groups only takes a very small fraction. In this case, if we take out collusion groups from ratings, we still be able to estimate τ.

Assume no environment factors present. Let the relatively large number of total ratings of an app where collusion group does not exist be N and the current average rating be τ. In order to archive manipulated average rating $\hat{\sigma} = \tau + \epsilon_0 (\epsilon_0 > \epsilon)$, the size of collusion group is:

$$S = \frac{\epsilon_0 N}{\eta - \epsilon_0}, \tag{3.4}$$

where $\eta = \lambda - \tau$ and λ is the rating that each malicious user gives.

In order to significantly manipulate app average rating with smaller size of collusion group, η should be as large as possible, which means collusion group members usually always give 5-star rating. For app A with user rating size $N = 10000$ and average rating $\tau = 3.5$, in order to reach $\hat{\sigma} = 3.8$, the size of collusion group $S \approx 834$, assuming that every collusion group member gives 5-star rating($\lambda = 5$). If every collusion group member gives 4-star rating($\lambda = 4$), collusion group size will be $S = 5000$. This also explains the reason that collusion group usually rates one app at relatively same time: if N increases by time, S has to increase as well.

Therefore, we could have the following theorem for malicious users:

Theorem 3.3. *A user is* ***malicious user*** *if the following conditions are satisfied:*

1. *Is a member of at least one collusion group.*

2. *In all malicious events he/she participates, most of ratings are either 4-stars or 5-stars.*

3.1.3 Threat Models

We have known that to manipulate the app's average rating, collusion groups have to exist. But how are collusion groups formed? We found two possible ways to for malicious users to form a collusion group.

1. **Direct Formation** means someone *directly* hires a group of malicious users or fakes a large number of IDs to launch attacks. It also happens in social network attack and DoS attack. For iTunes app store, this kind of collusion groups sometimes have consecutive user IDs. That's because attackers may use some tools to generate those fake user IDs and iTunes generates user IDs incrementally. But this characteristic cannot be found in other app stores as they use hash code as user IDs.

2. **Indirect Formation** means someone *indirectly* hires a group of malicious users. For example, we found that some websites welcome volunteers to purchase apps and leave comments. Volunteers will get total refund and bonus after that. No fake IDs are needed in this attack, so consecutive IDs usually will not be found.

3.2 Why Use Machine Learning?

Even we know the characteristics of abused app, it's still very difficult to find out it due to the following reasons:

1. Defining a threshold of number of ratings N is hard.

2. Find exact value of ϵ is difficult.

3. Detecting a collusion group in a malicious app is unreachable. To verify a group is collusion group, we have to make sure it has attacked at least two apps. This process is time-consuming. It is possible that a collusion group only attack one app, but we will lose it since we could not verify its existence. Therefore, we need to find other features that could reflect the existence of collusion group.

4. It's impossible to rule out environment factors.

Those reasons encourage us to use machine learning to bypass the difficulties we face. The goal of machine learning is to find a hypothesis h from hypothesis set $\mathcal{H}$ that is closest to the function f, which is a function that mapping a vector $\mathbf{x}$ to y[7]. For this problem, vector $\mathbf{x}$ denotes the values of features for each app and y is a binary value indicates if this app is abused or not.

With the help of machine learning, we don't have to worry about finding exact values of N, ϵ and exact values of features describing the existence of collusion group as machine learning algorithm will choose the *best* h for us as a training model.

The way we decide the *best* training model h is to try to both minimize E_{in} and E_{out} given a training set, where E_{in} is error rate that h has in training set and E_{out} is the error rate that h has in the complete data set, no matter what evaluation method we use to calculate error rates. However, measuring E_{out} is tricky since E_{out} is the error rate in out of sample data and there is no way to know. That's why cross-validation[8] is important — only use a part of original training data as new training data and use the rest to measure E_{out}. While we may not be able to find a training model h that minimizes both E_{in} and E_{out}. It's feasible to find a training model h that minimize E_{out}(by cross-validation) and has relative small E_{in}(to avoid overfitting).

In machine learning, factors that we cannot control or expect are noises. In this problem, environment factors can be treated as noises. However, not all environment factors are uncontrollable. iTunes App Store provides version information for each comment, therefore, we may generate some features based on version.

By avoiding fitting training data perfectly(overfitting), the effects of noise to training data will be minimized[9]. This is another advantage we want to take from machine learning.

3.3 App Discrimination For Training Set

Training set is required for any machine learning problems as our final hypothesis function f has to learn from it. To guarantee the performance of our training model, our training set has two following requirements:

1. The size of our training set should be large enough.

2. Training set should be not be biased.

3.3.1 Abused Apps Detection For Training Set With Confirmation Of Existence Of Collusion Groups

We use an algorithm proposed by Zhen Xie[1] to detect abused apps for our training set. This algorithm has the following features:

1. Threshold constants such as number of ratings N, correlation coefficients between different attributes, threshold of collusion group size, etc. are predefined.

2. Collusion groups are discovered and confirmed in order to label abused apps correctly.

3. Other constraints that could indicate rating manipulation are considered.

The most important advantage of this algorithm is that it will confirm the existence of collusion groups. Therefore the true positive rate of this algorithm is very high. However, confirmation can be made only if the collusion group have attacked at least two apps, presuming that those apps have been crawled by us. Therefore, if collusion groups that have only attacked one app, or only one of apps that has been attacked by collusion groups is crawled by us, there is no way to detect neither collusion groups and abused apps. Even this is a problem, with the help of this algorithm, our training model is still able to know what abused apps look like and figure out the distinctions between abused apps and benign(normal) apps.

3.3.2 Picking Apps For Training Set Manually

Section 3.3.1 only detects abused apps for training set. In order to both increase the size of training set and keep training set as unbiased as possible, we also manually detect abused apps and pick normal apps from our data set.

The method of manually detecting abused apps is: 1) looking for rating spikes that are similar to one shown figure 3.1 in one version of app; 2) confirming that there is at least one collusion group that has attacked at least two apps; 3) taking

look at the content of reviews left by the members of collusion groups. The usual case is that malicious users tend to leave short comments containing less valuable information.

This method has both pros and cons:

Pros the algorithm we use to automatically find abused apps uses very strong constraints. Our method could find some abused apps that this algorithm missed.

Cons we could possibly missed some abused apps since the review data is too large for human being to process.

We only label abused apps and put them into training set and ignore the rests. Therefore, training set only contains abused apps that can be theoretically confirmed.

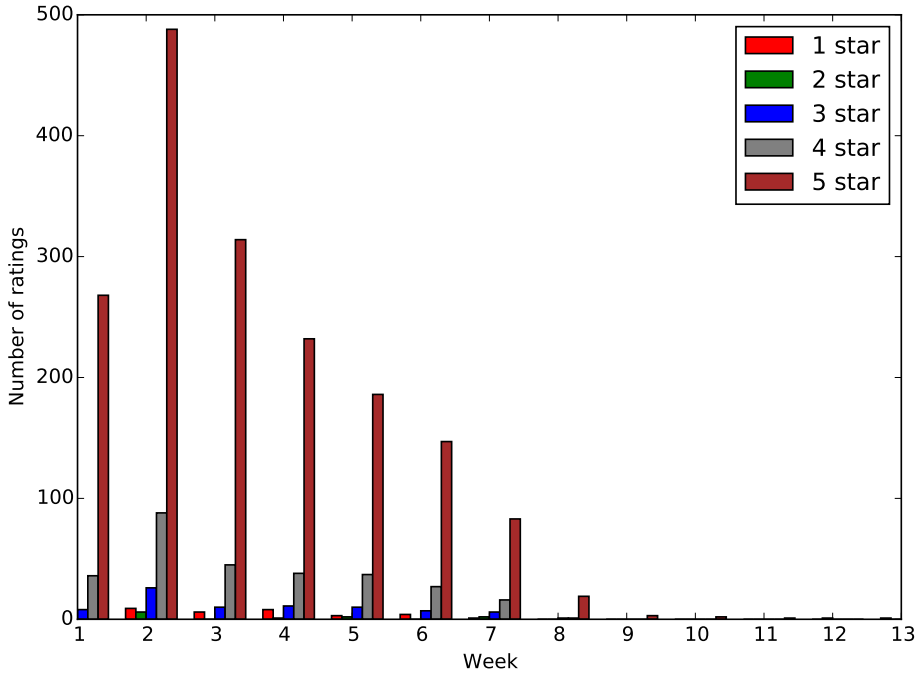

Figure 3.2: Number of Each Star (App ID: 585027354, Version: 2.0)

To pick benign apps for training set, we use the following four methods:

1. Almost all ratings are 4 stars or 5 stars. Figure 3.2 falls into this case. This app turns out to be *Google Map*. Although there is a huge spike in this

graph, it is normal as app's numbers of total ratings by week could be fitted to Poisson distribution nicely, which indicates that this spike is created by a normal user behavior. Moreover, the numbers of 1 stars and 2 stars increase or decrease as the numbers of 4 and 5 stars go up or down.

2. Apps that are publicly known due to their good qualities and their high average ratings(between 4-star and 5-star). It seems very risky to label those apps as benign ones, but is actually understandable. If an app's rating is very high, such as 4.5 and is always that high, even if collusion groups exist, it's unlikely that ϵ could be larger than deviation threshold. If the deviation threshold is $\epsilon = 0.5$, then it's impossible for this app to reach this threshold. "Publicly known" means the number of ratings N is relatively large, so it's extremely difficult to manipulate the average rating, as shown in equation 3.4.

3. Apps that are developed by some companies that have gained public trust. We select the apps developed by Apple, Google, Microsoft, Facebook, Amazon, etc.. Those companies have been playing with software and its market for a long time and users usually will download their apps just because they rely on the services these companies provide, such as Google Map, Facebook, Pages and so forth.

4. Apps whose numbers of reviews are less than 100. As we shown in Section 3.1.2, we only have 72.93% confidence to claim that app's current average rating is close to its true quality within 0.1. We select some of those apps and put them into our training set.

These four methods basically cover popular apps, unpopular apps, high average rating and low average rating apps, aiming to reduce the bias of training set. Readers may question us about the correctness of our methods, especially method 3. In Section 3.4, we will justify them.

3.3.3 Training Set Overview

Apply the methods we propose to pick our training set and we have the following data as our training sets.

Table 3.1: Training Set Table

App store	Abused Apps	Normal(benign) Apps	Total Apps
iTunes US	134	779	913
iTunes China	206	659	865
iTunes UK	91	803	894

We would like to mention that in machine learning theory, picking training data is very tricky, specially when we are able to pick data by ourselves. If the training data are biased, our training model will be biased as well. To mitigate these potential issues, when we are picking benign apps, not only did we try to make sure the apps are benign, but also, from the whole training set's perspective, make sure that we are building a mini version of iTunes app store. That's why we put popular apps, unpopular apps, high average rating apps and low average rating apps, high ranking apps, low ranking apps, apps that have been published for a very long time and newly released apps into our training set.

3.4 Ground Truth

Readers may question us about the absence of ground truth for training set and we do have the following explanations:

1. **Ground truth is not a necessary condition for machine learning problems.** Ground truth refers to the correctness (accuracy) of a test set's classification for supervised learning. However, many supervised learning do not have ground truth in the first place. Some banks use machine learning to decide whether their applicants are qualified for credit cards or not. The training set they use is gathered from historical decisions made by credit officers. However, whether an applicant is *truly* qualified for credit cards are unknown since banks could not have the complete information about applicants and applicants could lie about their annual income and other financial information. In this case, the decision sometimes is based on how much trust credit officers have for applicants.

2. **Asking ground truth for an opinion based behavior is difficult.**

Rating manipulation is similar to credit card application, whether an app is abused or not is basically based on how much trust we have for ratings(or users themselves). From app market providers' perspective, if they trust all the ratings, no app is abused; if they don't trust any ratings, all apps are abused. If some organizations or authorities could possibly give us the ground truth, it's still based on their opinions. For any online rating system research, no researchers have claimed that they have ground truth[10][11]. Theorem 3.2 and lemma 3.1 are our baselines for discriminating abused apps from benign ones.

3. **The goal of using machine learning in this problem is not ruling out human's intervention.** Since whether an app is abused or not is subjective, using machine learning aims to narrow down the scope of the highly suspicious apps instead of replacing human's work, especially when highly sensitive decisions have to be made.

4. **To avoid the arguable distrimination, we only label abused apps that we confident with.** High accuracy is good for any machine learning problems. But for abused app detection, it's more safe to assume a suspicious app as benign one than an abused one. Therefore, we also expect higher precisions(the percentage of labeling an app as abused one correctly) in cross-validation. Readers may question us about method 2 and 3 we use to pick benign apps in Section 3.3.2. But both methods show that we tend to draw a line between companies that the public trust and companies who need rating manipulation to promote their apps. Including trustworthy companies' apps into training data has another advantage: both abused apps and benign apps' rating can be affected by environment factors(defined in definition 3.2). By doing so, we could indirectly tell learning algorithm that some data influenced by environment factors are noise and should not be taken in to consideration while it is learning the training data and making classification.

3.5 Summary

In this chapter, we give the definitions of environment factor(noise), abused app, malicious user and collusion group. We also characterize abused app, malicious user and collusion group. The difficulties that we will encounter when using explicit algorithm to detect abused apps based on those characteristics lead us to seek machine learning for help. The questions about the absence of ground truth are answered and we describe how to build our trustworthy training set both automatically and manually before we present our training set data. In Chapter 4, we will discuss all the features we gather and extract from the data we crawled in detail.

Chapter 4

Features

Feature is a numeric describing data from one or more aspects. We transform the data we crawled from app stores into different features so that learning algorithms are able to learn from the data. In this chapter, we will discuss all the features we select.

4.1 Primary Features

We define primary features as the features that can be collect from the raw data itself without further processing and transformation. Since we our classification target is app, the primary features are selected from table 2.1a. For better understanding, we will use some space to elaborate them in the pattern $feature\_name(feature\_ID)$.

average_rating(1) : average of all ratings.

total_rater(2) : number of all ratings.

1star_num(3) : number of all 1 star ratings.

2star_num(4) : number of all 2 star ratings.

3star_num(5) : number of all 3 star ratings.

4star_num(6) : number of all 4 star ratings.

5star_num(7) : number of all 5 star ratings.

price(8) : 1 if this is a paid app; otherwise 0.

Readers may be curious about the motivation of including feature 3–8 because it does not make many senses to put them into feature set. In Chapter 5, we will see that the learning algorithm we use is able to rank these features based on their importances. Therefore, we are safe to put all the features that could describe the app data into feature set.

4.2 Advanced Features

Besides from the primary features that are ready to be used without any pre-processing, we want to transform the comment and reviewer data into more advanced features.

num_dev(9) : a developer may develop more than one apps. This feature describe how many apps one developer has developed.

To describe the reviewers' (users who leave comments) behavior, we define the following terms:

Definition 4.1. *If a reviewer gives 4 or 5 stars for all the apps that he/she has rated, he/she is a* ***positive reviewer****; if a reviewer gives 1 or 2 stars for all the apps that he/she has rated, he/she is a* ***negative reviewer****.*

Many reviewers only rate app once, so it might be risky to label those reviewers as positive reviewers or negative reviewers. To avoid this potential pitfalls, we define the following terms:

Definition 4.2. *If a* ***positive reviewer*** *rate 3 or more apps, he/she is* ***extremly positive reviewer****; if a negative reviewer rate 3 or more apps, he/she is* ***extremely negative reviewer****.*

num_pos_reviewer(10) : number of positive reviewers of this app.

perc_pos_reviewer(11) : percentage of positive reviewers, or $\frac{num\_pos\_reviewer}{total\_number\_of\_reviewer}$. Note that $total\_number\_of\_reviewer$ is not $total\_rater$ as reviewers leaves comments but raters do not.

num_neg_reviewer(12) : number of negative reviewers of this app.

perc_neg_reviewer(13) : percentage of negative reviewers, or $\frac{num\_neg\_reviewer}{total\_number\_of\_reviewer}$.

num_extr_pos_reviewer(14) : number of extremely positive reviewers of this app.

perc_extr_pos_reviewer(15) : percentage of extremely positive reviewers of this app, or $\frac{num\_extr\_pos\_reviewer}{total\_number\_of\_reviewer}$.

num_extr_neg_reviewer(16) : number of extremely negative reviewers of this app.

perc_extr_neg_reviewer(17) : percentage of extremely positive reviewers of this app, or $\frac{num\_extr\_pos\_reviewer}{total\_number\_of\_reviewer}$.

Helpfulness might be useful as well.

helpfulness_ratio_avg(18) : as we mentioned in Section 2.1, helpfulness ratio is $\frac{number\_of\_people\_agree}{number\_of\_people\_agree+number\_of\_people\_disagree}$. This is the average of helpfulness ratio, or $\frac{\sum helpfulness\_ratio}{number\_of\_comment}$.

num_helpfulness(19) : number of comments whose helpfulness ratios are greater than 0.

perc_helpfulenss(20) : percentage of comments whose helpfulness ratios are greater than 0, or $\frac{num\_helpfulness}{number\_of\_comment}$.

Variance is able to detect how much fluctuation a feature has. We aggregate comment data by week. We think it's an appropriate granularity since we only have data divided in days and usually rating manipulation spans weeks. The term "rating" in the following paragraphs is only the rating with comment, since only comment has date.

var_num_comment_by_week(21) : variance of number of comments by week.

var_avg_rating_by_week(22) : variance of average ratings by week.

var_perc_rating_by_week(23, 24, 25, 26, 27) : variance of percentage of 1, 2, 3, 4 and 5 star ratings by week.

var_perc_pos_reviewer_by_week (28): variance of percentage of positive reviewers by week.

var_perc_neg_reviewer_by_week (29): variance of percentage of negative reviewers by week.

We notice that when the app is just released, version updates, etc. there will be some spike in terms of number of ratings. Poisson distribution is often used to describe those spike. Usually to let user know about new apps, app stores put them into new category, which is located in an obvious place on the front page. A few weeks or days later, other new apps come and replace those old ones. Another case is, when an app is updated, people who have used it tends to download and rate it or leave comment on it again. Commercials and ads may attract user's attention, etc. All those could lead to spikes and number of comment usually follow the poisson distribution.

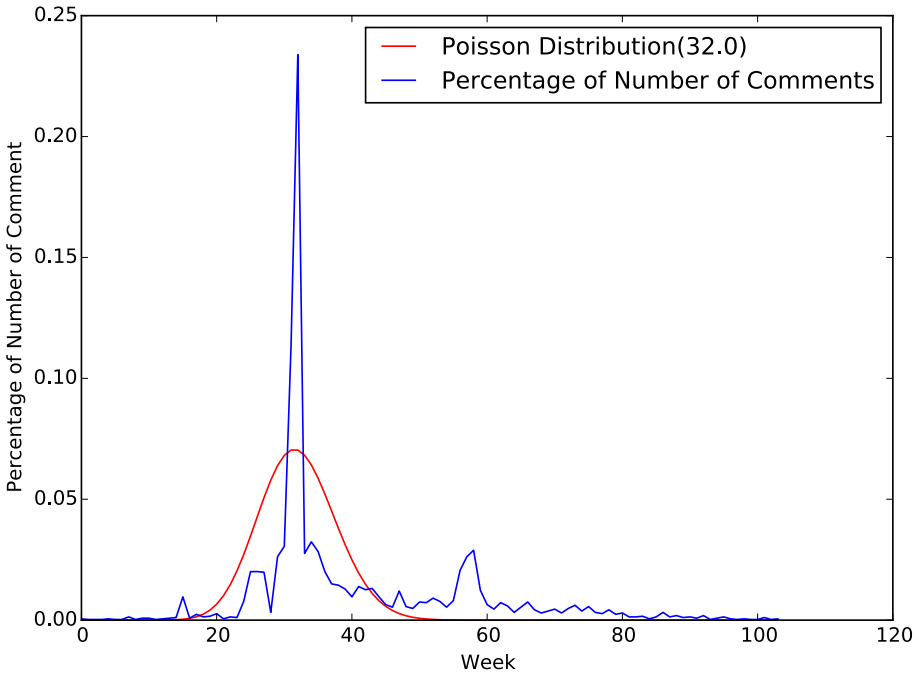

Figure 4.1: Poisson Fitting(app ID: 460351323; Store: iTunes China)

Figure 4.1 and 4.2 depict the poisson distributions that we use to fit the data. Readers may see that poisson distributions we found out does not fit the data

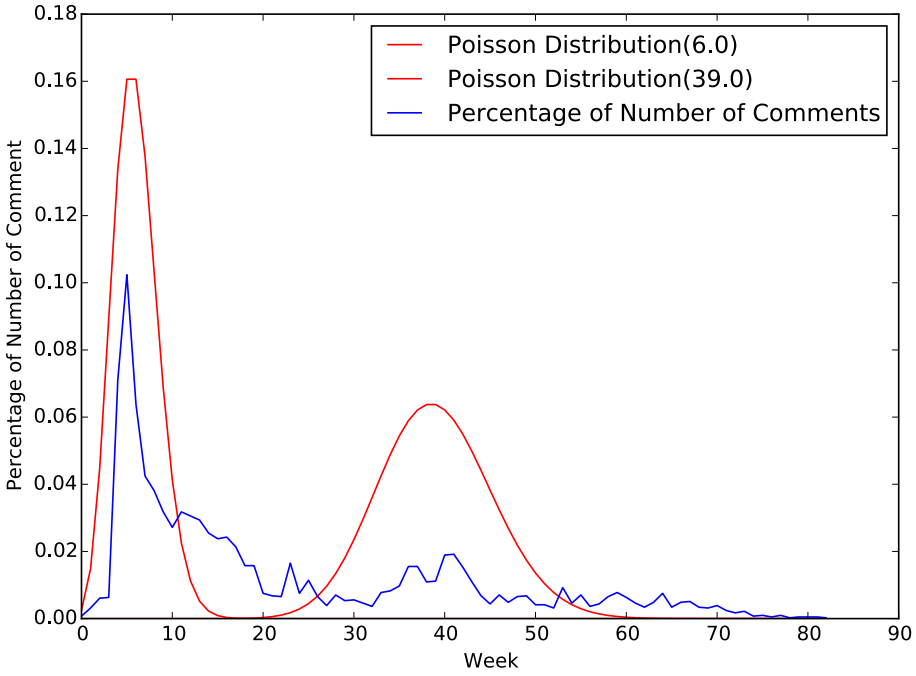

Figure 4.2: Poisson Fitting(app ID: 521134765; Store: iTunes China)

perfectly. Since we want to use poisson distribution to find the locations of spikes, so it does not matter. However, the tricky part of the fitting is about the granularity we choose. If we choose a very fine granularity, then it may end up with plenty spikes. If we choose a coarse granularity, we may miss some spikes. To mitigate this problem, we choose 20% largest values from the data set and use them as initial values to find poisson distributions that fit the data with least square error measurement. That's the reason that there are only two poisson distributions in figure 4.1. After locating all the poisson distributions, we are able to use those data as features.

poisson_num_peaks(30) : number of poisson distributions we find.

poisson_first_peaks(31) : relative position of first poisson distribution(the one with smallest λ), or $\frac{min(\lambda)}{number\_of\_weeks}$.

poisson_last_peaks(32) : relative position of last poisson distribution(the one with largest λ) or $\frac{max(\lambda)}{number\_of\_weeks}$.

We define the following terms to capture special weeks:

Definition 4.3. *If during one week, all the ratings are 4 or 5 stars, this week is* ***positive week****; if all the ratings are 1 or 2 stars, this week is* ***negative week****.*

Then we select following features:

num_week(33) : number of weeks of this app's lifespan(or till now).

num_pos_week(34) : number of positive week.

num_neg_week(35) : number of negative week.

perc_pos_week(36) : percentage of positive week, or $\frac{num\_pos\_week}{num\_week}$.

perc_neg_week(37) : percentage of negative week, or $\frac{num\_neg\_week}{num\_week}$.

max_pos_week(38) : largest number of continuous weeks that all of them are positive week.

perc_max_pos_week(39) : percentage of largest number of continuous weeks that all of them are positive week, or $\frac{max\_pos\_week}{num\_week}$.

max_neg_week(40) : largest number of continuous weeks that all of them are negative week.

perc_max_neg_week(41) : percentage of largest number of continuous weeks that all of them are negative week, or $\frac{max\_neg\_week}{num\_week}$.

Even version is an environment factor, we want to capture its affect. We firstly treat each version of app as an individual app, then normalized it.

var_perc_pos_reviewer_by_week_by_version(42) : summation of variances of percentage of positive reviewers by week then normalized by number of versions, or $\frac{\sum_{i=1}^{n} var\_perc\_pos\_reviewer_i}{n}$, where n is number of versions.

var_perc_pos_reviewer_by_week_by_version(43) : summation of variances of percentage of negative reviewers by week then normalized by number of versions, or $\frac{\sum_{i=1}^{n} var\_perc\_pos\_reviewer_i}{n}$, where n is number of versions.

var_num_reviewer_by_week_by_version(44) : summation of variances of number of reviewers by week then normalized by number of versions, or $\frac{\sum_{i=1}^{n} var\_num\_reviewer_i}{n}$, where n is number of versions.

var_avg_rating_by_week_by_version(45) : summation of variance of average ratings by week then normalized by number of versions, or $\frac{\sum_{i=1}^{n} var\_avg\_rating_i}{n}$.

var_perc_1_star_rating_by_week_by_version(46) : summation of variance of percentage of 1 star ratings by week then normalized by number of versions, or $\frac{\sum_{i=1}^{n} var\_perc\_1\_star\_ratings_i}{n}$, where n is number of versions.

var_perc_2_star_rating_by_week_by_version(47) : summation of variance of percentage of 2 star ratings by week then normalized by number of versions, or $\frac{\sum_{i=1}^{n} var\_perc\_2\_star\_ratings_i}{n}$, where n is number of versions.

var_perc_3_star_rating_by_week_by_version(48) : summation of variance of percentage of 3 star ratings by week then normalized by number of versions, or $\frac{\sum_{i=1}^{n} var\_perc\_3\_star\_ratings_i}{n}$, where n is number of versions.

var_perc_4_star_rating_by_week_by_version(49) : summation of variance of percentage of 4 star ratings by week then normalized by number of versions, or $\frac{\sum_{i=1}^{n} var\_perc\_4\_star\_ratings_i}{n}$, where n is number of versions.

var_perc_5_star_rating_by_week_by_version(50) : summation of variance of percentage of 5 star ratings by week then normalized by number of versions, or $\frac{\sum_{i=1}^{n} var\_perc\_5\_star\_ratings_i}{n}$, where n is number of versions.

We found that in normal apps, the increment of number of reviewers will lead to increment of number of each stars. Zhen[12] points out that, in normal apps, correlation coefficient between number of reviewer and average rating in each week should be close to 0, which means these two factors should have no apparent relation and should have no effect to each other. Therefore, we could have the following features:

coef_pos_neg_rating_by_week(51) : correlation coefficient between numbers of 1, 2 star ratings and 4, 5 star ratings by week.

coef_1_5_num_rating_by_week(52) : correlation coefficient between numbers of 1 star ratings and 5 star ratings by week.

coef_2_5_num_rating_by_week(53) : correlation coefficient between numbers of 2 star ratings and 5 star ratings by week.

coef_3_5_num_rating_by_week(54) : correlation coefficient between numbers of 3 star ratings and 5 star ratings by week.

coef_avg_num_rating_by_week(55) : correlation coefficient between average rating and number of raters by week.

4.3 Summary

In this section, we basically elaborate all the features that we collect and build from our data. Two kinds of features are included: 1) app's basically features and 2) app's advanced features describing its ratings, comments and reviewers information. In Section 5, we will discuss features' rankings by importance and how we use machine learning algorithm to generate training model from training data and those features.

Chapter 5

Training With Random Forest

5.1 Random Forest

Random Forest[13] is an ensemble[14] machine learning algorithm proposed by Leo Breiman. Generally speaking, random forest is a meta estimator that fits a number of decision tree classifier on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. "Forest" refers to decision tree[15], where each node corresponds to one of the input variables; each leaf represents the target variable's value. "Random" refers to the method of choosing m variables from total M variables randomly for each node of the tree and sample training set for each tree randomly[16]. The algorithm of how each tree is grown as follows:

1. Sample N training cases randomly from original training set as new localized training set for growing the tree, where N is the size of original training set.

2. For each node in the tree, m input variables are selected from total M variables and $m \ll M$.

3. Each tree is grown to the largest extend possible.

The number of estimators(trees) is not specified. Increasing number of estimators does not always guarantee better accuracy[17], so we will try different numbers of trees when we are training our models. We will discuss that in Section 5.3.

Among many popular machine learning algorithms such as SVM[18], neural network[19], deep learning[20], etc., we choose random forest as learning algorithm due to the following several reasons:

1. It runs efficiently on large data set.

2. It gives estimates of what variables are important in the classification. We will use this advantage to analyze our features in Section 5.2.

3. The algorithm itself does not overfit. Random forest use "bagging"[21] to select a subset of training data to grow a tree and use the rest of them to do internal validation.

4. It could ensembles many decision trees and each tree could decide the exact values of N, ϵ and other undetermined values in Chapter 3. From this perspective, using random forest as learning algorithm makes more sense.

5.2 Feature Importances

As we mentioned in Section 5.1, random forest could estimates the importances of each features. Feature importances are generated during the training process. Therefore, it's not a separated procedure from training. But analyzing features can be very helpful when we want to know more about the data and figure out which features contribute more in the training model.

A popular feature importance measuring method is proposed by Leo Breiman[13]. The essence of this method is to rank the error rate of each forest grown with different composition of features by calculating gini index[22].

5.2.1 US iTunes App Store Feature Ranking

Figure 5.1 shows the feature ranking by importance in US iTunes app store. All the importance values of features are normalized. The higher ranking the feature has, the more error the training model will generate without this feature. By depicting the feature ranking, we are able to see which features play important role during abused app detection and we can also give some reasonable explanations of the rank of first 10 features.

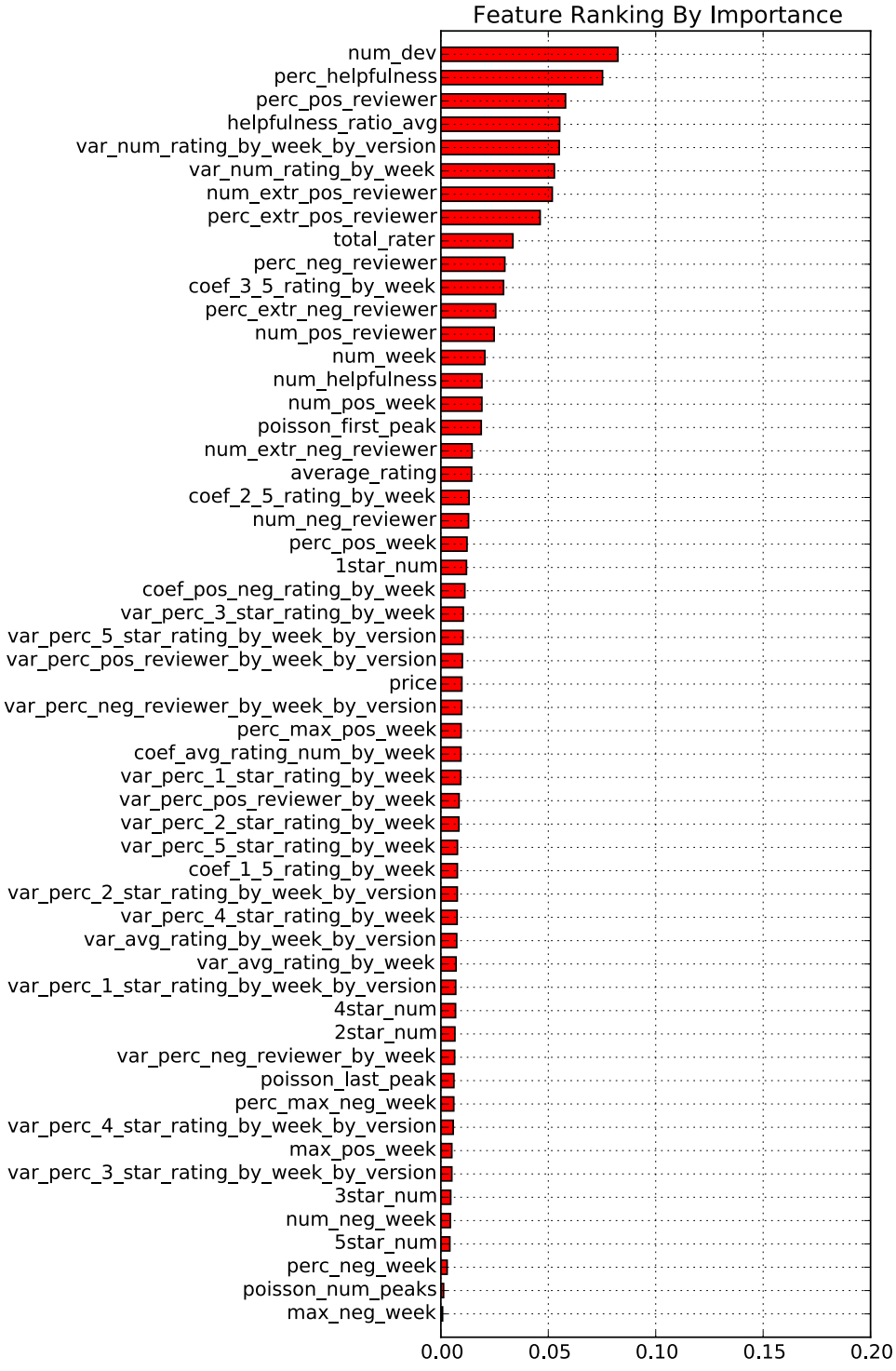

Figure 5.1: US iTunes app store feature ranking by importance

1. **num_dev**: in US iTunes app store, abused apps are usually developed by small companies who developed a few apps. We manually inspect the developer websites of some abused apps in the training set and find that usually those websites are ill-designed and poor user interface. Those companies are not well-known and are recognized as small companies.

2. **perc_helpfulness**: users in U.S app store usually like to give feedback on other users' comments. Helpfulness is a very effective way to reflect the usefulness of comments. To speed up their attacking process, attackers usually leave some simple and uninformative comments, which draw normal user's less attention. Moreover, iTunes will rank the comments not only by comment dates, but also by helpfulness ratio. Therefore, the more informative comments will gain more feedbacks.

3. **perc_pos_reviewer**: as we suggest in Section 3.1.2, to manipulate the average rating as effective as possible, malicious users tend to give high ratings(4 or 5) each time. This feature is a strong indication of existence of malicious users and collusion groups.

4. **helpfulness_ratio_avg**: besides *perc_helpfulness*, this feature describes the percentage of agrees. An uninformative comments gain, even thought some feedbacks, usually negative feedback. In other words, informative and genuine comments will have higher average helpfulness ratio but rake comments that attackers leave will have lower value.

5. **var_num_rating_by_week_by_version**: apps with low value of this feature will be in safe area. High value of this feature could be suspicious as the existence of collusion groups could result in this, even we still need other features to clarity our suspicion because environment factors are strong noises. Advertising or dropping price could also lead to this high variance.

6. **var_num_rating_by_week_by_version**: similar to **var_num_rating_by_week_by_version** but ignore app version factor.

7. **num_extr_pos_reviewer**: similar to *perc_pos_reviewer*. Note that two features together indicate the high number of reviewers, which is a necessary

condition of abused app.

8. **perc_extr_pos_reviewer**: similar to *perc_pos_reviewer*. But this feature is a more strong indication of existence of malicious users and collusion groups since *extreme positve reivewer* is reviewer who has given 4 or 5 stars at least 3 times.

9. **total_rater**: usually large number of total raters leads to large number of reviewers, which is a necessary condition of abused app.

10. **perc_neg_reviewer**: lower value of this feature makes apps more suspicious. We found that, in most cases, malicious users and collusion groups usually promote apps instead of demote other developers' apps. Lower value of this feature suggests the larger number of normal users(neither are negative users nor positive users) and positive users.

5.2.2 China iTunes App Store Feature Ranking

Figure 5.2 shows normalized feature ranking by importances in China iTunes app store. We will give explanations of top 10 features.

1. **perc_pos_reviewer**: as discussed in Section 5.2.1, high value of this feature strongly indicates the existence of malicious users and collusion groups.

2. **var_num_rating_by_week_by_version**: apps whose numbers of ratings by week are stable are usually benign ones.

3. **perc_extr_pos_reviewer**: similar to **perc_pos_reviewer**.

4. **var_num_rating_by_week**: Usually both popular apps and abused apps share this feature if this value is relatively large. But apps with low value of this feature are normal ones.

5. **total_rater**: identical to that in Section 5.2.1.

6. **num_pos_reviewer**: similar to **perc_pos_reviewer**.

7. **num_extr_pos_reviewer**: similar to **num_pos_reviewer**.

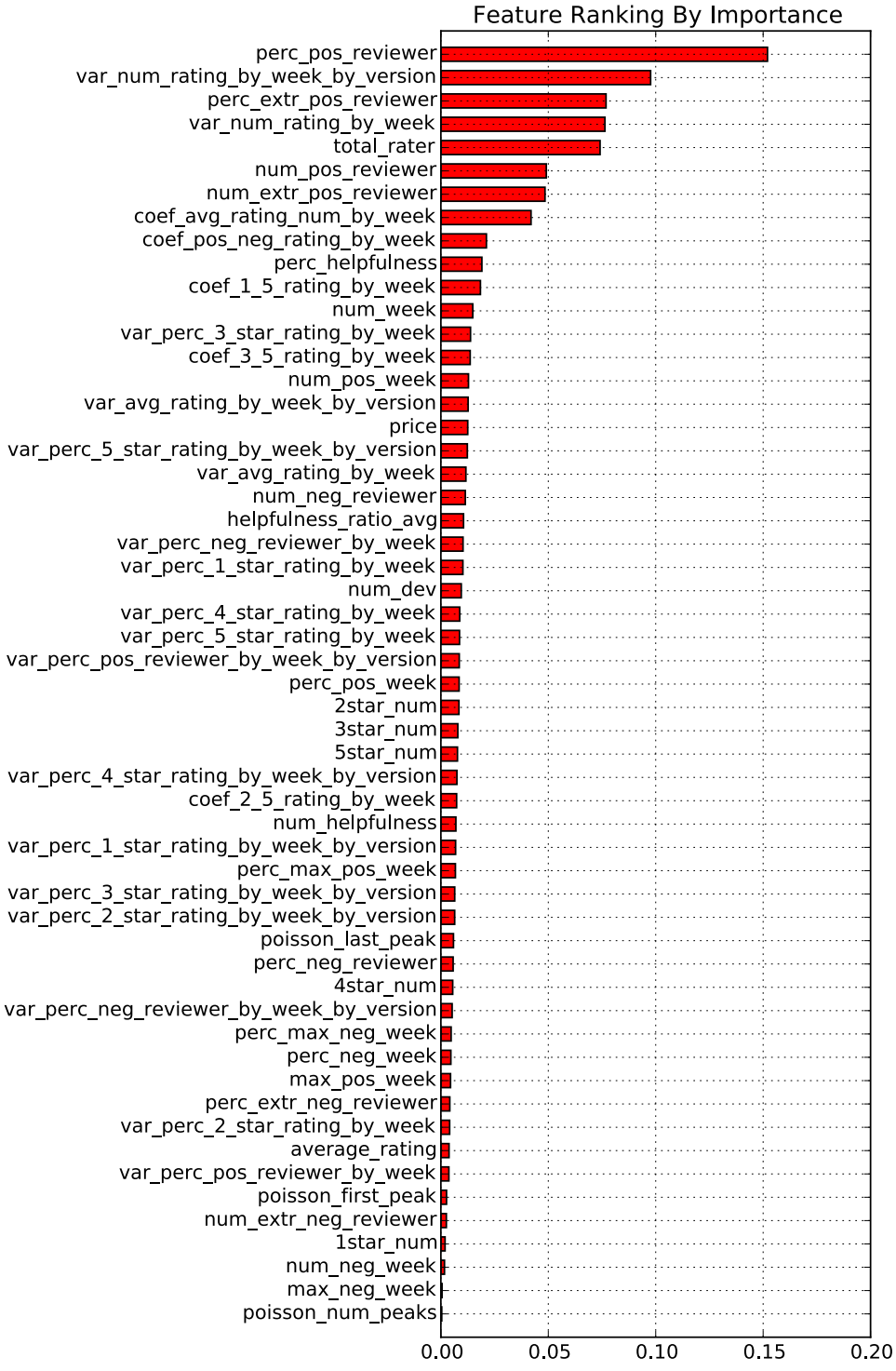

Figure 5.2: China iTunes app store feature ranking by importance

8. **coef_avg_rating_num_by_week**: as suggested[12], the value of this feature should be close to 0. High correlation between average rating and number of reviewers indicats that there are malicious users and collusion groups manipulating average rating.

9. **coef_pos_neg_rating_by_week**: similar to **coef_1_5_rating_by_week**.

10. **perc_helpfulness**: identical to this feature in Section 5.2.1.

5.2.3 UK iTunes App Store Feature Ranking

Figure 5.3 shows normalized feature ranking by importances in UK iTunes app store. We will give explanations of top 10 features.

1. **perc_helpfulness**: same as this feature in Section 5.2.1.

2. **num_dev**: same as this feature in Section 5.2.1.

3. **helpfulness_ratio_avg**: similar to **perc_helpfulness**.

4. **perc_pos_reviewer**: same as this feature in Section 5.2.1 and Section 5.2.2.

5. **num_extr_pos_reviewer**: identical to this feature in Section 5.2.2.

6. **var_num_rating_by_week**: even high value of this feature cannot increase the possibility of labeling an app as abused app, low value of this feature could rule out some benign apps and narrow down our scope.

7. **var_num_rating_by_week_by_version**: same as this feature in Section 5.2.1.

8. **num_helpfulness**: the more number of helpfulnesses, the more useful comments an app has, because a large number of useful comments indicates the large portion of benign users.

9. **total_rater**: identical to this feature in SectionSection 5.2.1 and 5.2.2

10. **poisson_first_peak**: most apps have one peak of number of downloads during the first few weeks. However, if this peak occurs later, this app becomes suspicious.

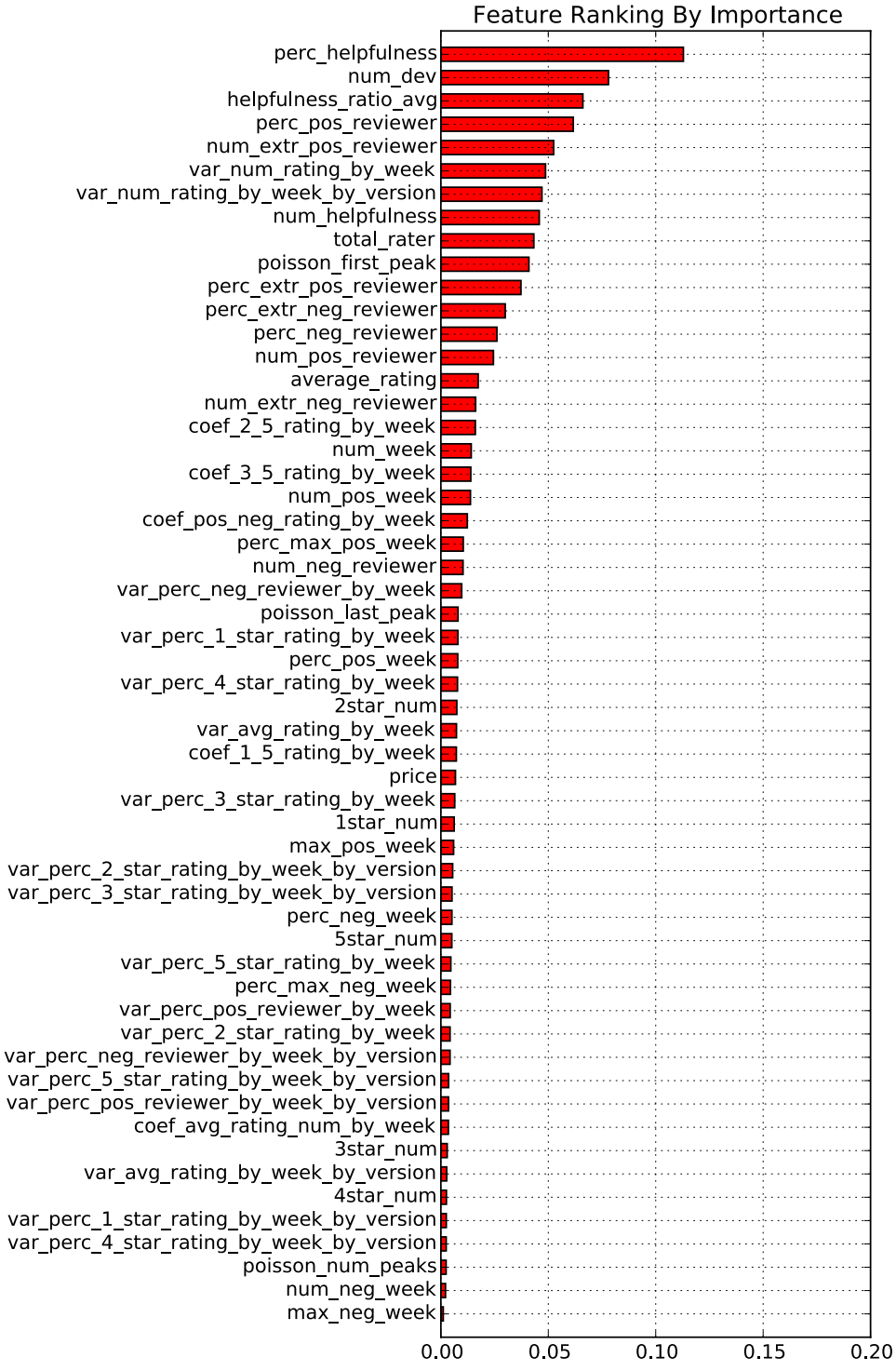

Figure 5.3: UK iTunes app store feature ranking by importance

5.2.4 Feature Analysis

When we put feature ranking from those three countries together, we are able to find both similarities and differences. In this section, we will dive into those feature and analysis those similarities and differences.

5.2.4.1 Similarities

Even we create training sets from each of dataset separately, we still find that some features always rank higher than others, which means those features play significant roles in abused app detection. The followings are those similarities with their analysis.

1. High values of **num_pos_reviewer**, **perc_pos_reviewer**, **num_extr_pos_reviewer** and **perc_extr_pos_reviewer** strongly indicate the existence of malicious users and collusion group. And this can been seen from all three dataset. This also confirms our analysis in Chapter 3 that to effective manipulate an app's average rating, malicious users tend to give 5 star, or at least 4 star and their group size should be large enough so that their rating could go against the rest of benign users.

2. As we suggest in theorem 3.2, only with large number of **total_rater**(ranks 9 in iTunes US app store, 5 in iTunes China app store, 9 in iTunes UK app store), we could possibly label those apps as abused ones. If number of **total_rater** of one app is too small, we lose our confidence in estimating app's true quality, therefore we shall label this app as normal app.

3. Correlation coefficient related features, such as **coef_1_5_rating_by_week**, etc. and variance related features, such as **var_num_rating_by_week**, etc. rank in top 15 in each country. Outlier values of these features directly indicate the existence of collusion groups, as by definition, members of collusion group manipulate average rating during a short period of time in order to effectively change average rating and reduce financial cost.

4. Helpfulness related features are important in abused app detection. Normal users will read the comments and possibly leave some feedbacks, while mali-

cious users tend to publish new reviewers in order to manipulate the average ratings.

5.2.4.2 Differences

Differences seem very obvious. Each dataset has its own data and we create different training sets from each of them. Therefore, feature rankings are different. But feature **num_dev** draws our attention. As we described in Chapter 4, **num_dev** is the number of applications that one developer develops. Big companies such as Google, Gameloft, EA, have published many apps. Therefore they have higher value of **num_dev**.

However, this feature is critical only in US and UK iTunes store but it ranks 43rd in China app store. Our reasonable guess is that in US and UK app stores, it's more likely that small companies abuse their apps instead of big companies; in China app store, even some big technology companies will manipulate their apps' average ratings. This guess is confirmed by our abused app data in the training set as we found several abused apps are developed by biggest technology companies in China.

5.3 Training

The official page of random forest algorithm[1] states that random forest does not overfit and we can freely choose the number of trees($n\_estimator$). However, Mark R. Segal found that this algorithm will overfit for some noisy datasets[23]. Moreover, as random forest will rank all the features by importance during the training process, it is very convenient to select top n features. In this section, we will evaluate our training model after discussing our expected results.

5.3.1 Learning Goals

For every machine learning problem, the ultimate goals are:

1. High precision and recall for training set.

[1] `http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm#remarks`

2. Accurate prediction in out-of-sample data.

Usually item 1 is reasonable to reach if the training model is good enough. But item 2 is relatively difficult due to the lack of out-of-sample data. Sometimes, even we do have out-of-sample data, evaluation is still impossible as, for supervised learning, those data are not labeled. In this specific training problem, our goals are:

1. while pushing precision and recall, precision is supposed to be better than recall. Precision is defined as

$$precision = \frac{\sum true\_positive}{\sum test\_outcome\_positive}. \tag{5.1}$$

In this problem, precision indicates the confidence of abused apps we classify. Recall is defined as

$$recall = \frac{\sum true\_positive}{\sum total\_positive}. \tag{5.2}$$

In this problem, recall indicates coverage of our results. Usually, higher precision will lead to lower recall, and vice versa[24]. For abused app detection, we emphasize on higher precision since we want to make sure we minimize the probability of labeling benign apps as abused apps and we could tolerant to missing several abused apps.

2. narrowing down the scope of abused app detection instead of declaring abused apps. For app market providers, labeling an abused app is very serious and sensitive and it always involves business level decisions. We hope that our machine learning results could help app market providers narrow down the scope of abused app detection – only have to inspect abused apps we classify.

5.3.2 Learning Method

Generally speaking, we use two-phase training in our method. Figure 5.4 depicts the whole procedure of our machine learning approach. **Raw data** is the data we crawled from each app store and it has been discussed in Section 2.2. **Feature generator** is a program we use to generate all the features discussed in Chapter

4. The output is **data with features**, which will be split into **tranining set** and **residual data**. How we choose this **training set** has been discussed in Chapter 3. As we mentioned in Section 5.1, random forest algorithm is able to rank all the features by their importance. **Random forest(a)** is used to generate this ranking. The whole training is done with scikit-learn[25].

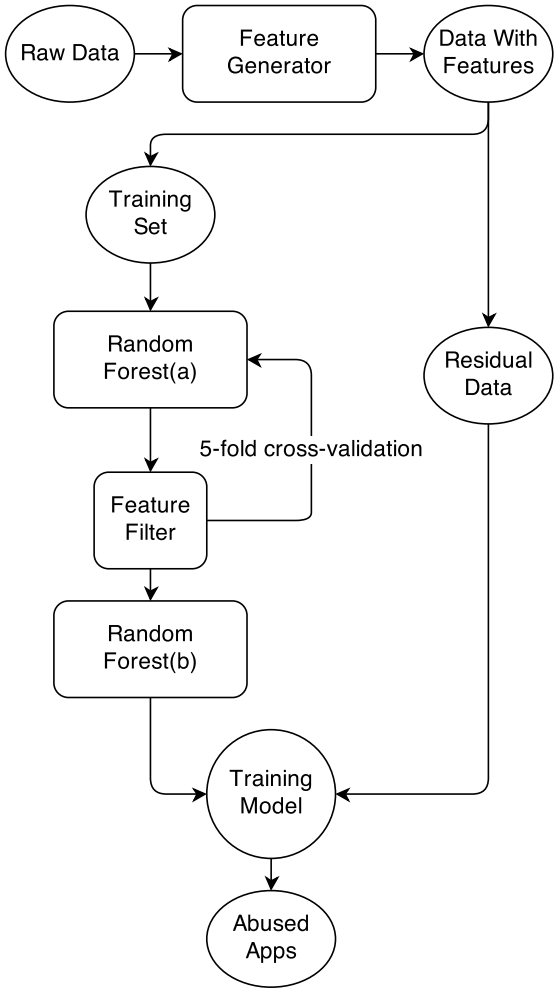

Figure 5.4: Learning Procedure

To improve our training model[26], we do not use the training model generated by **random forest(a)**. Instead, we use **feature filter** to work with **random forest(b)** to select top n features – n is chosen by comparing the evaluations performed with 5-fold cross-validation. We use algorithm 1 to select the best top n features.

Algorithm 1 Feature Filter

```
1: procedure FeatureFilter(feature_ranking)
2:     f_score ← list()
3:     for n ← len(feature_ranking), n >= 10, n ← n − 1 do
4:         feature_list ← top_n_feature(feature_ranking)
5:         f_score_local ← list()
6:         for n_estimator ← 61, n_estimator ≤ 101, n_estimator ← n_estimator + 2 do
7:             rfc ← RandomForestClassifier(n_estimator)
8:             avg_f_score ← 5_fold_cross_validation(rfc, feature_list)
9:             Inserts avg_f_score into f_score_local_gets
10:        end for
11:        max_f_score ← max(f_score_local)
12:        Inserts (max_f_score, n) into f_score
13:    end for
14:    Returns best f_score and its n
15: end procedure
```

The inner loop will iterate different number of trees($n\_estimator$) in random forest algorithm in order to find the best number of trees for it, as we mentioned in the beginning of Section 5.3. $n\_estimator$ is always odd number in order to make sure classification results are deterministic since each tree takes a vote for the target value. Outer loop will select top n(n from 10 to 55 in this case) and pick the best f-score from the inner loop.

There is a theory behind the feature filtering step. VC-dimension[27] is introduced to measure the complexity of the size of hypothesis set $\mathcal{H}$, denoted by $|\mathcal{H}|$. The more features that we use to training our model, the larger $|\mathcal{H}|$ is, so is the VC-dimension. However, practically speaking, the smaller VC-dimension will result in under-fitting and the larger VC-dimension will results in over-fitting. From random forest's perspective, the more feature we introduce, the more complicated each decision tree is. In this case, we might be end up with using too many features to learning training set and these features will form a very specific concept, which has poor generalization. If we don't include enough features, the decision tree is very simple. In this case, those features will form a very board concept that will cover many negative instances. Feature filter can solve this problem by iterating all the top N features and find a training model that is neither under-fitting nor

over-fitting.

When we get the best testing results(highest f-score in this case), we select this training model as our final hypothesis h, as we discussed in Section 3.2, and we use h to classify new abused apps from **residual data**.

5.3.3 Results

Figure 5.5 depicts the precision, recall and f-score varies from top 10 features to top 55 features(all features) in iTunes US app store. We could have the following observations:

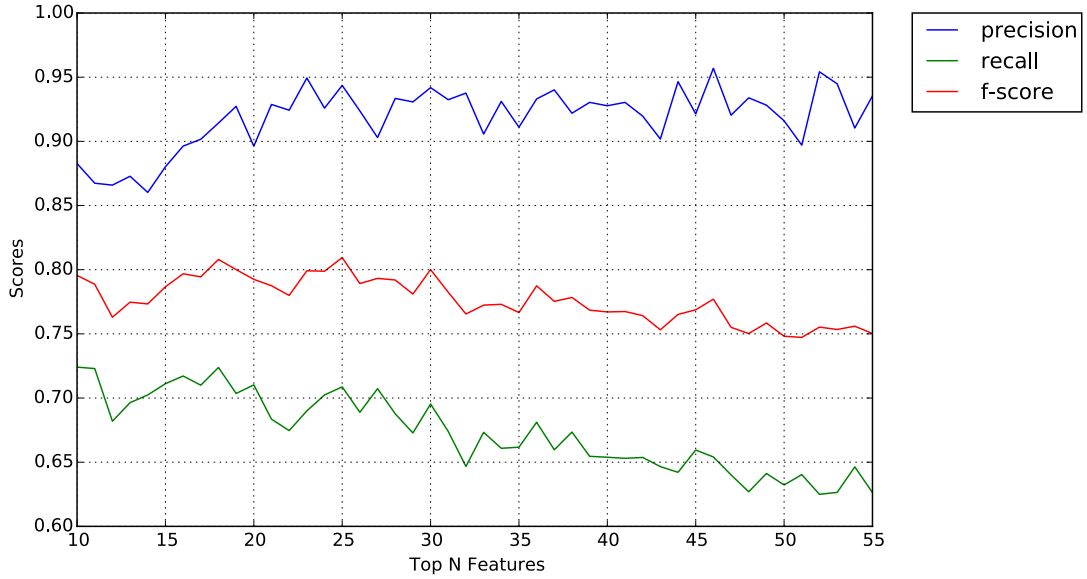

Figure 5.5: iTunes US App Store: Evaluation for Top n Features

1. Generally, as we increase the number of top features, precision increases slowly while recall and f-score decrease relatively fast.

2. Precision is always better than recall, which meets our requirement.

3. When $n = 18$, both f-score and recall reach their peaks.

Figure 5.6 and figure 5.7 are our evaluations for iTunes China app store and iTunes UK app store, respectively. Even the plot is very spiky for iTunes China

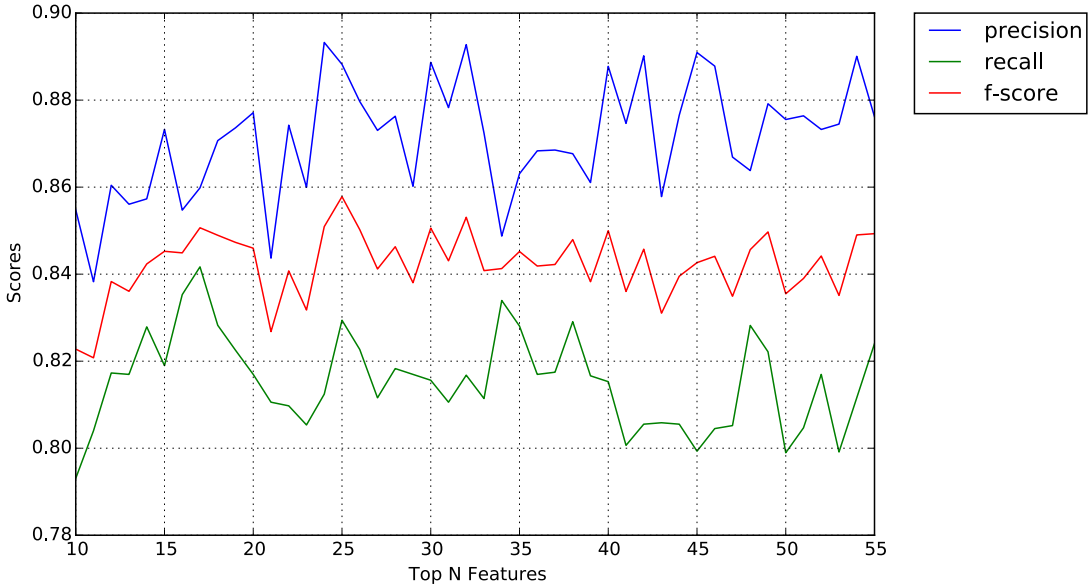

Figure 5.6: iTunes China App Store: Evaluation for Top n Features

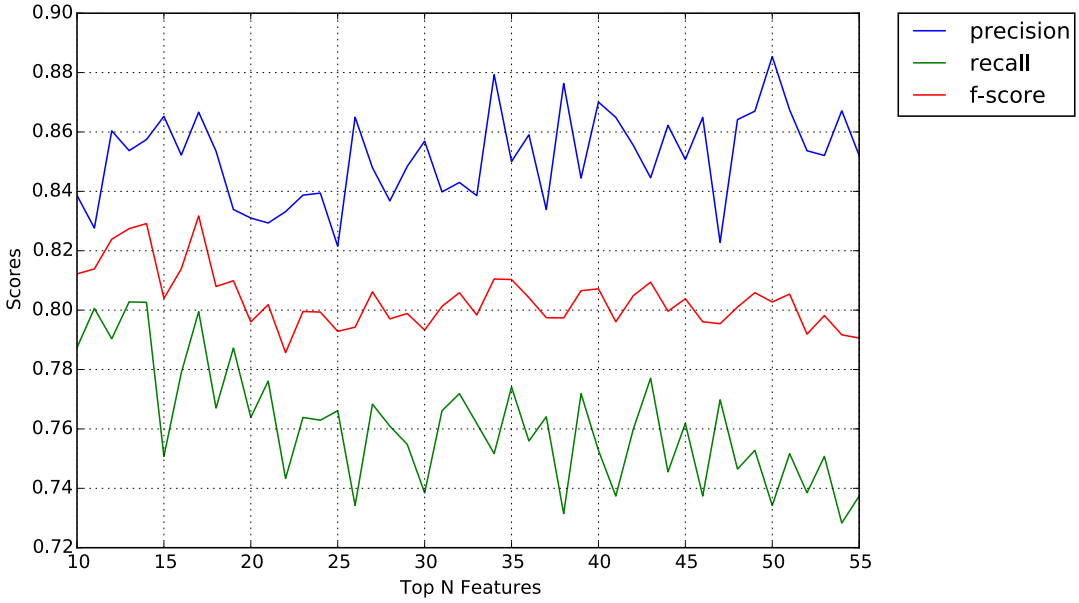

Figure 5.7: iTunes UK App Store: Evaluation for Top n Features

app store, we still have the same observations for both UK and China app stores as for US app store. For figure 5.7, even f-score($n = 10$) is slightly better than f-score($n = 15$), we are still willing to choose latter one as we refer high precision to high recall. Therefore, from both figures, we choose top 25 features for iTunes China app store and top 14 features for iTunes UK app store.

Table 5.1 summarizes our results.

Table 5.1: Traning Model Evaluation Table

App Store	Precision	Recall	F-score	Top n Features
iTunes US	91.43%	72.38%	80.52%	18
iTunes China	88.82%	82.94%	85.58%	25
iTunes UK	85.37%	80.28%	82.27%	14

After we getting our final training models(hypothesizes) h for each app store, we can use these training model to classify apps in **residual data**. We call the newly found 'abused app' as *suspicouse app* as we have no evidence to prove those apps are abused ones. Table 5.2 summarizes our results.

Table 5.2: Classified Suspicious Apps In Each App Store

App Store	# Suspicious Apps	# Residual Apps	Coverage(%)
iTunes US	1103	22703	4.86%
iTunes China	1123	21056	5.33%
iTunes UK	930	9777	9.51%

From table 5.2, we can see that our coverage in iTunes UK is relatively larger than other app stores. Intuitively, label 9.51% percentage apps as abused app (suspicious) in one app store is questionable. But our explanation is that we tend to crawl popular apps from app store. In other words, apps we missed are more likely to be unpopular ones and few users will review them. In this case, if they came to our training model and get classified, the possibility of being labeled as normal apps is high. Consider that, from app store provider's perspective, popular apps are expected to be noticed more easily and from crawler's perspective, apps that could be fetched with fewer hops are more easily to be collected by crawler. Same applies to iTunes US and iTunes China app stores: if we use these coverage to estimate the suspicious apps in the whole app store, there could be around 50,000

suspicious apps. But we could conclude that, if apps in app store are difficult for crawler to find, generally, they are difficult for human beings to find as well. Therefore, we are safe to say that, if we have crawled all the data in app stores, the actual number of suspicious apps our training model could label will not be much larger than the number in table 5.2.

5.3.4 Classification Evaluation

The question may be asked after we label some abused(suspicious) apps in **residual data** is that how to validate those apps or how suspicious those apps could be. Though direct validation is difficult as we have no evidence to show if those apps are abused or not, here, we present two additional features to evaluate our results.

5.3.4.1 Consecutive Reviewer IDs

In Section 3.1.3, we point out that some abused apps may contain consecutive IDs. Here we define **consecutive user IDs** as if two IDs rate one apps in the same day and difference between IDs is less than 1000.

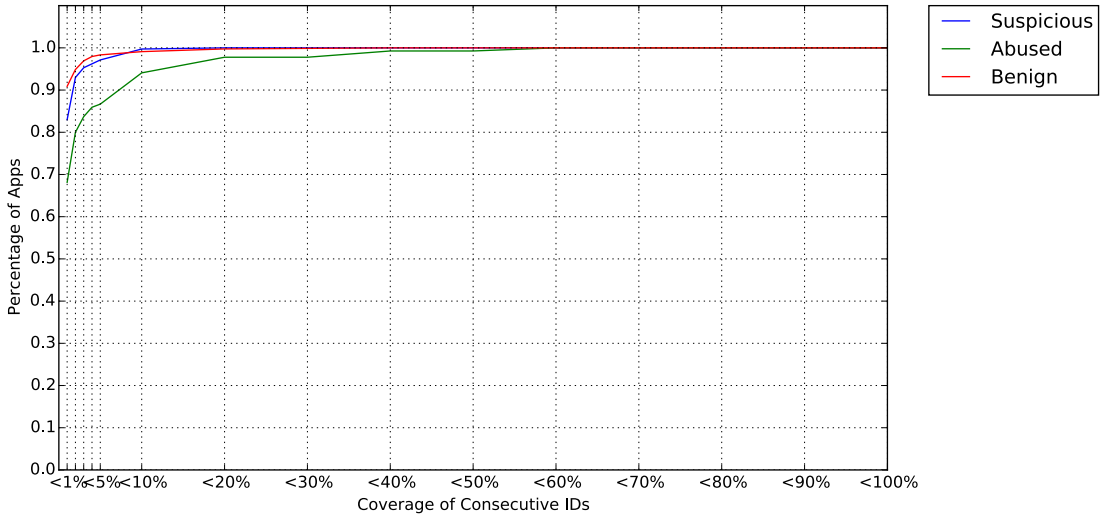

Figure 5.8: Consecutive User IDs in iTunes US App Store

Figure 5.8, 5.9 and 5.10 depict situations of existence of consecutive IDs in iTunes US, iTunes China and iTunes UK app stores, respectively. In each figure,

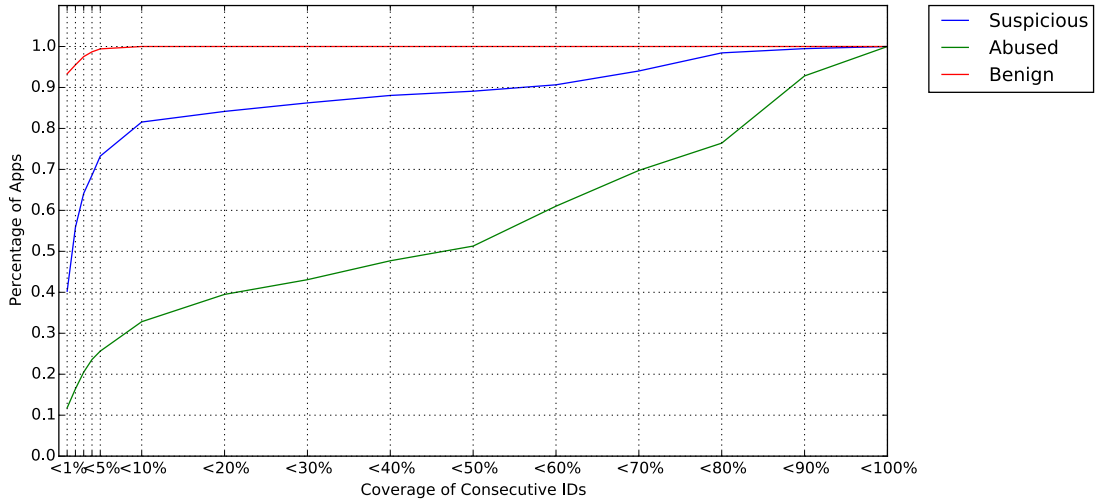

Figure 5.9: Consecutive User IDs in iTunes China App Store

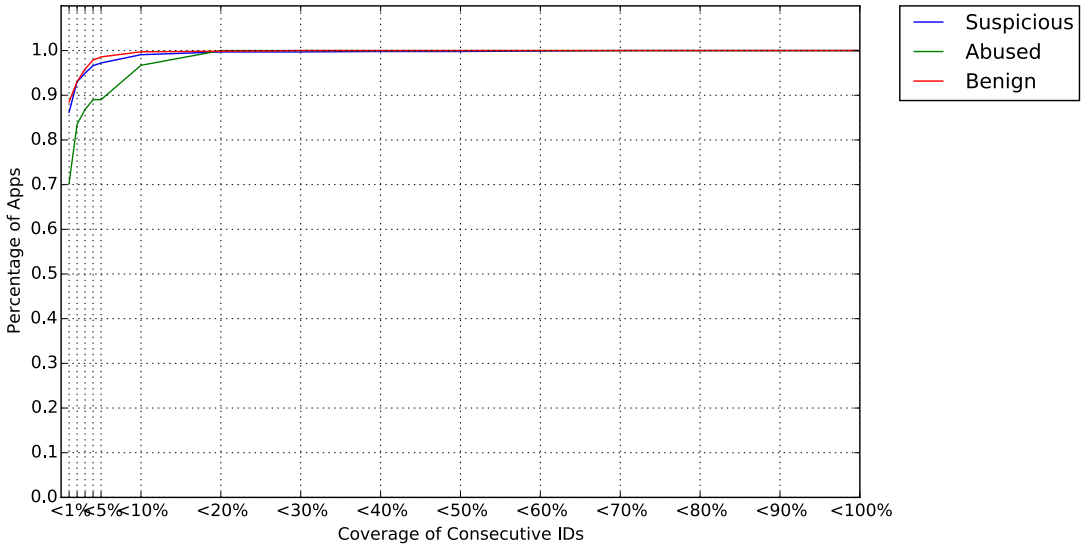

Figure 5.10: Consecutive User IDs in iTunes UK App Store

x-axis represents the coverage of consecutive IDs, which is defined as percentage of consecutive user IDs over number of users who leave comments for this app. For instance, '<10%' means that coverage of consecutive IDs is less than 10%. Y-axis represents the percentage of those apps over the number of apps in **residual data**. In the legend, 'suspicious' refers to the abused apps we classified by our training

model; ‘abused’ refers to the abused app in our training set; ’benign’ means the normal apps in our training data.

We easily see that some apps do contain more consecutive user IDs than normal apps do. Our training set contains more those apps than what we classified. That’s because when we are picking abused apps for our training set, the algorithm we use will explicitly find collusion groups that have attacked more than one apps. It’s conceivable that attackers will use same fake user IDs to launch another attack, which is much easier and more feasible than finding the same group of real users to attack another app. A group of fake users often have consecutive IDs, so those abused apps attacked by fake users can be more easily detected by our algorithm and put them into our training set and those abused apps attacked by real users remain undetected and discovered by our training models later.

Put these three figures together, we discover that, in iTunes US and UK, most abused apps are attacked by **indirect formation** groups, while in iTunes China, we are safe to draw conclusion that more than 50% of abused apps are attacked by **direct formation** groups(defined in Section 3.1.3).

5.3.4.2 Review Density

Another feature we use to evaluate our classification results is **review density**, which is the percentage of review area. Figure 5.11 is the number of reviews vs. week, where blue area is defined as review area and total area is represented by the square.

Usually, abused apps should have low review density. However, we notice that most apps, no matter they are benign or abused, will gain lots of reviews in the first several weeks. That’s because usually app stores will put new apps in a specific section in the front page so that users may be able to notice these new apps. To mitigate this issue, we remove the first 5 weeks data for each app.

Figure 5.12 depicts box plots of review density for iTunes US, China and UK app stores. We can see that, compared with benign apps, abused apps usually have lower review density. The suspicious apps that we classified have similar review density distribution to that of abused apps. Review densities of benign apps in China and UK are higher than those in US, which means US benign apps’ review distributions are less spiky, in general.

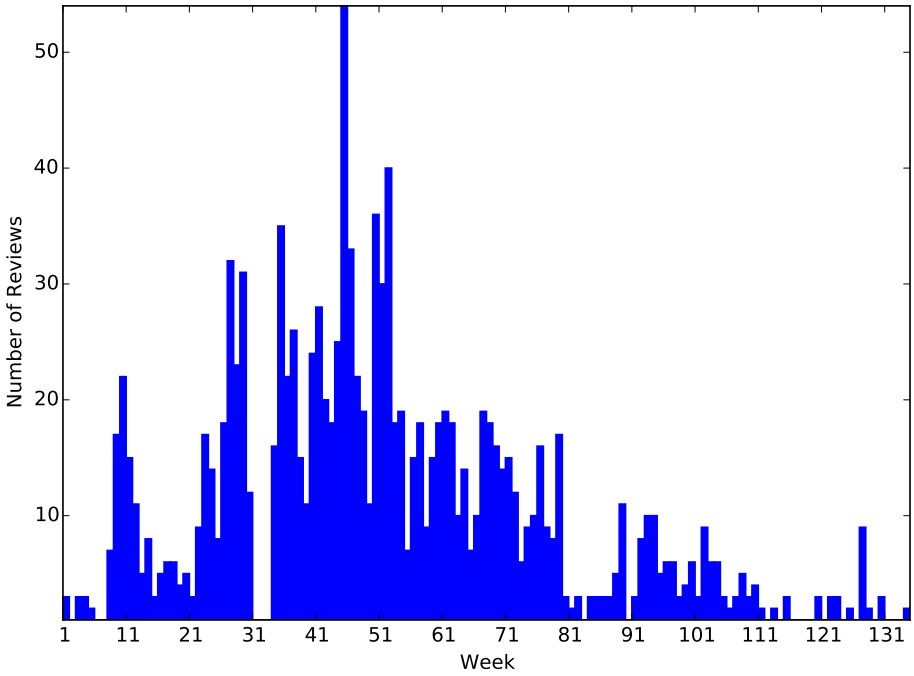

Figure 5.11: Review Density Exmple

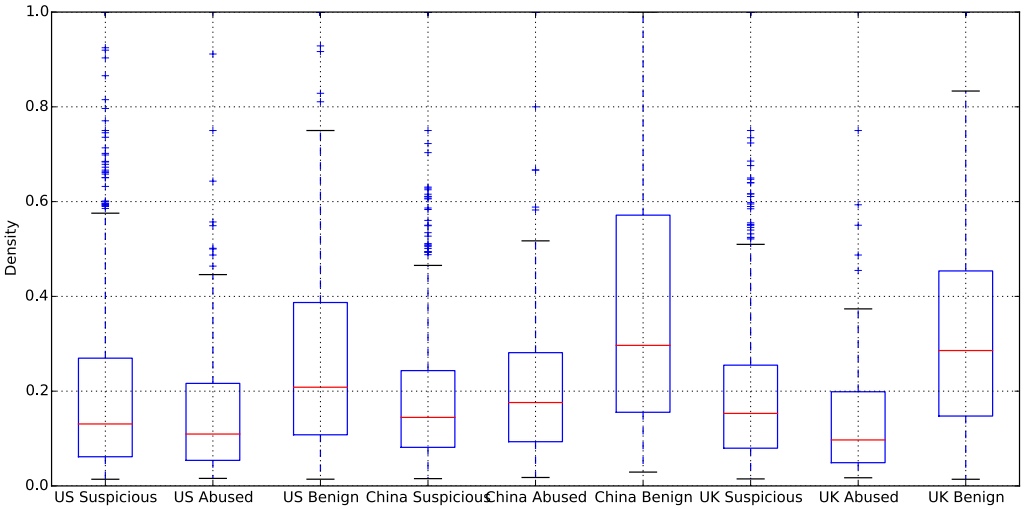

Figure 5.12: Review Density for iTunes US, China, UK App Store

There are some outliers in both suspicious and abused apps, whose review densities are relatively large. Nevertheless, it's superficial to say these outliers are false positive. As we cut out the first 5-week data for each app, there might be attacking behavior hiding in the first 5 weeks, which will greatly reduce the review density if we put those data back. Another possible case is, if abused apps are successfully promoted, users will easily find these apps in app store, which leads to many reviews.

We also randomly picked some suspicious apps we found and manually investigated their rating distributions. Even theoretically, it is very difficult for human being to process those large amount of data and find useful information from it, we did notice some skeptical data and figure 5.13 depicts one of them.

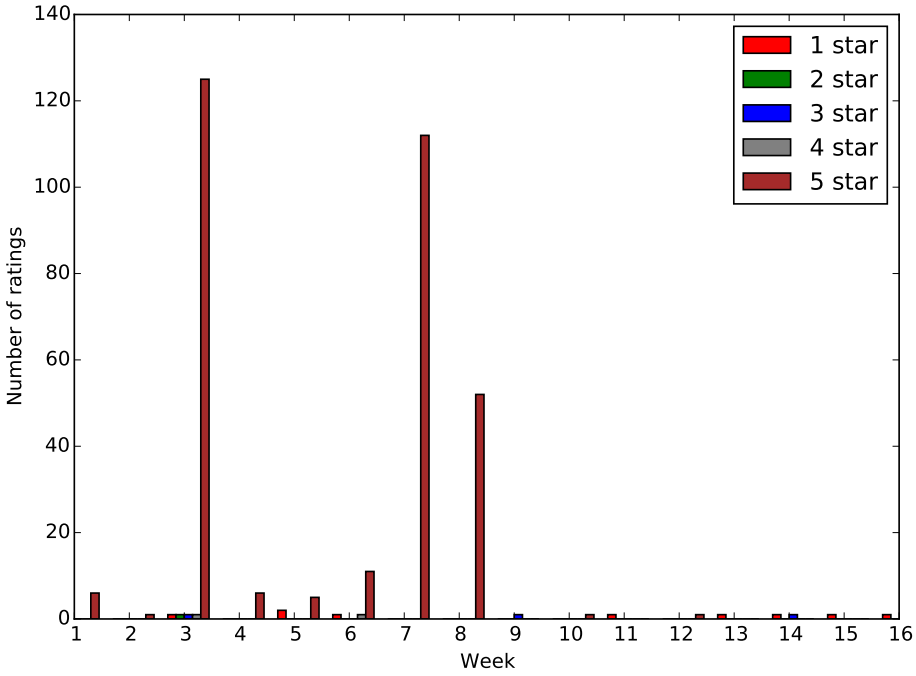

Figure 5.13: Suspicious App (app ID: 593313544)

5.4 Data Analysis

We have done some data analysis during our training and result evaluation, but it's worth analyzing the data we have from different perspectives.

5.4.1 Categories of Abused App

Figure 5.14 depicts the percentage of apps of each category in three app stores. Note that abused apps here include abused apps in our training set and suspicious apps classified by our training models.

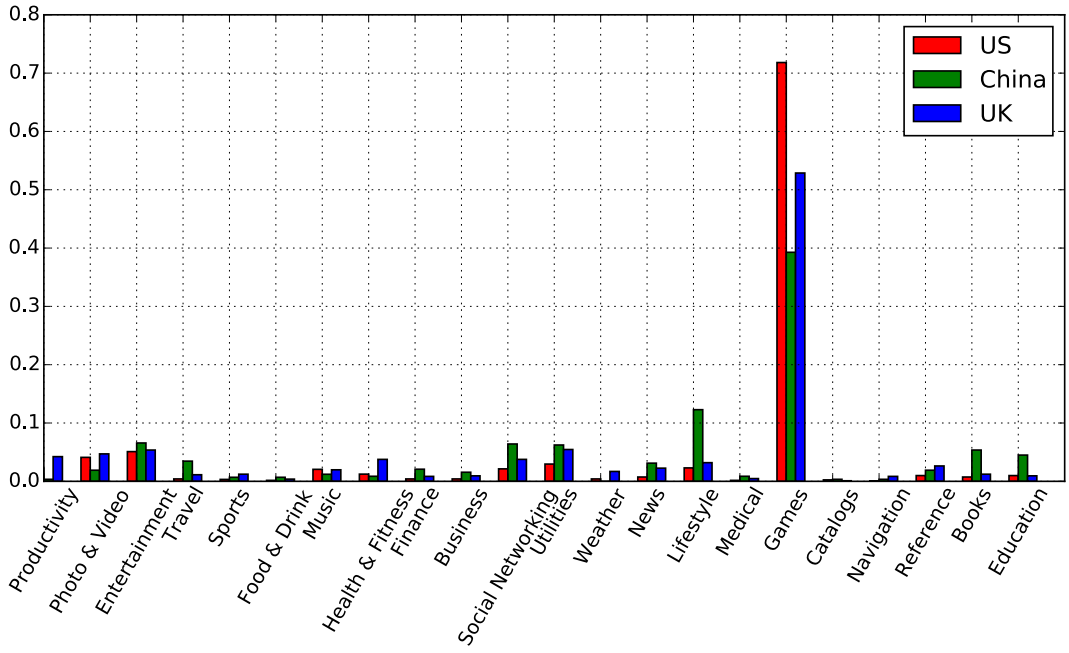

Figure 5.14: iTunes Abused Apps Categories

Even we use different training sets to train our models, in different countries, game apps are more likely to be abused than any other categories. In iTunes US and UK app store, game apps take up more than 70% and 50%, respectively. Either because of there are most games than other apps or games in iTunes app store always draw more attention than any other categories simply because games have more varieties and many users would like to explore new games and spend money on them. In this case, promoting a game app could attract more users' attentions than promoting a sport app. In China, more(in terms of percentage) lifestyle apps are abused. It might because many coupon companies and e-commerce companies are competing for market shares.

5.4.2 Review Content Analysis

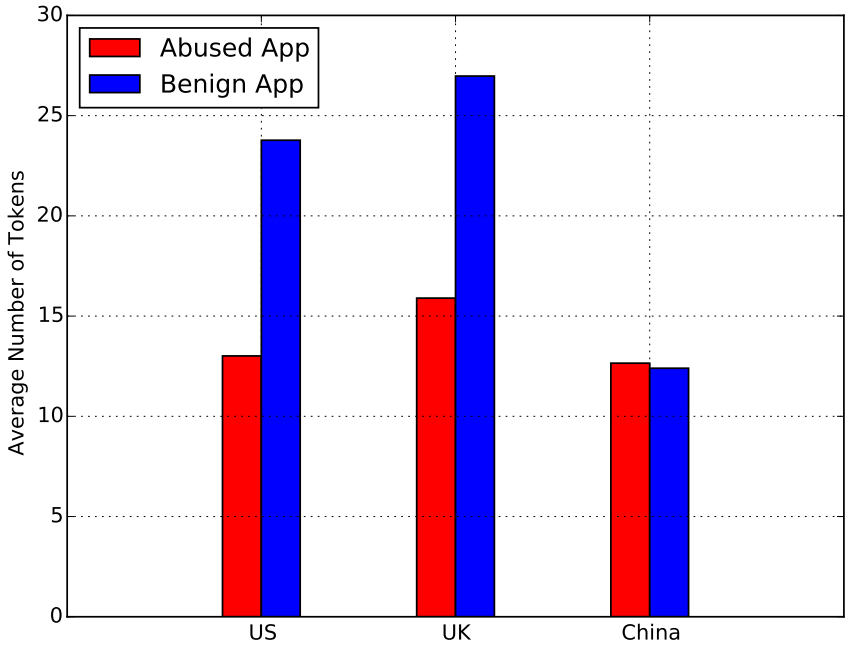

Figure 5.15: Average Review Content Lenght in iTunes US, UK and China App Store

If collusion groups focus on manipulating average scores, to reduce the cost, malicious users usually leave simple and short reviews. Abused apps here include both abused apps in the training set and suspicious apps we classified with our training model. Benign apps are the rest of apps. Figure 5.15 depicts the average length(number of tokens) of review contents for abused apps and benign apps in iTunes US, UK and China app stores. The method used to calculate review length is to tokenize each review content and title without punctuations. For English, number of tokens is number of words, while for Chinese, it is the number of characters. Some comments contain several languages, but with Lucene's[28] help, our tool is still able to count the number of tokens correctly.

We have the following three take aways from this figure:

1. For US and UK users tend to write longer reviews than users in China. Even English and Chinese are quite different, by manually inspecting review

contents, we found that US and UK users tend to describe the apps they purchased in great detail and share more personal experience. But China users tend to give a brief judgement and usually won't share user experience.

2. In iTunes US and UK app stores, compared with benign apps, the average length(number of tokens) of comments for abused apps is nearly half of that for benign apps, which means malicious users do tend to leave simple and short comments.

3. In iTunes China app store, we can see that the average lengths are same for both of abused and benign apps. That's because even benign users in China leave short and simple reviews.

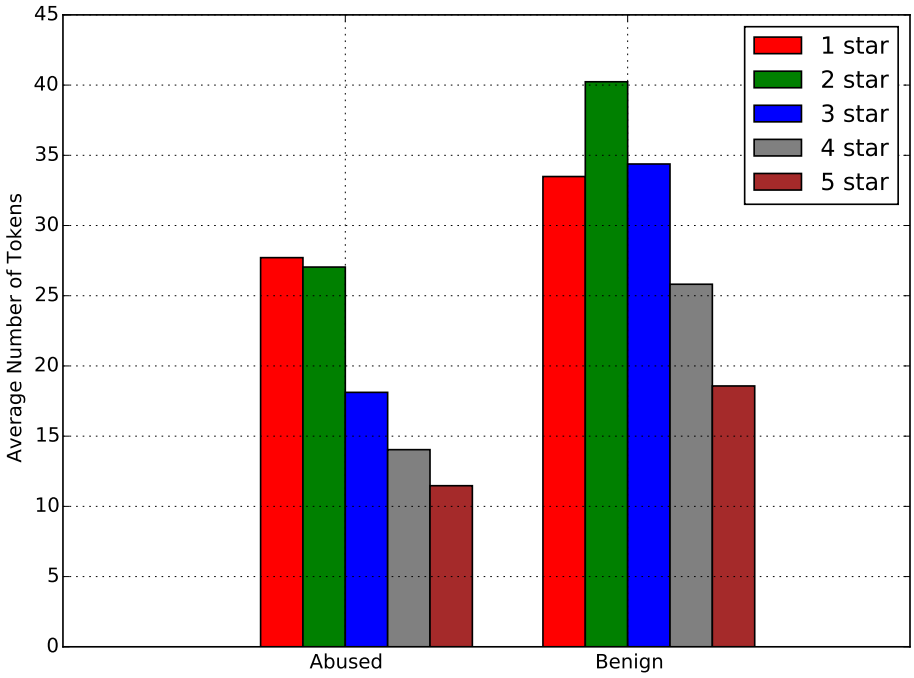

Figure 5.16: Average Review Content Lengths in iTunes US App Store

We also calculate the average review lengths of each star in these three app stores. By inspecting figure 5.16, 5.17 and 5.18, we have the following observations:

1. The average lengths of higher rating levels(4 and 5 stars) are lower than those of lower rating levels(1 and 2 stars). The higher rating level, the shorter its

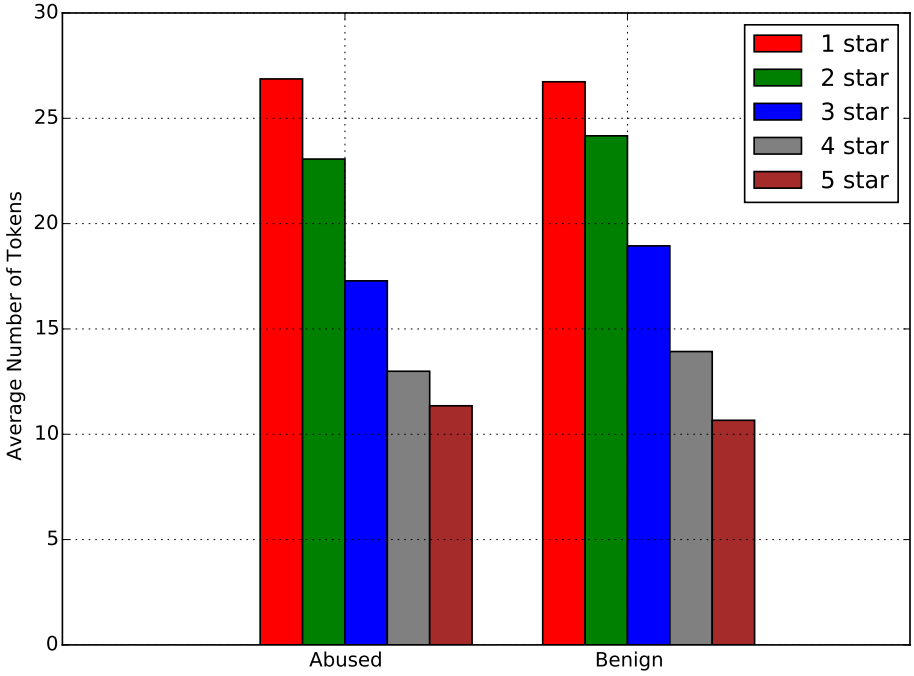

Figure 5.17: Average Review Content Lengths in iTunes China App Store

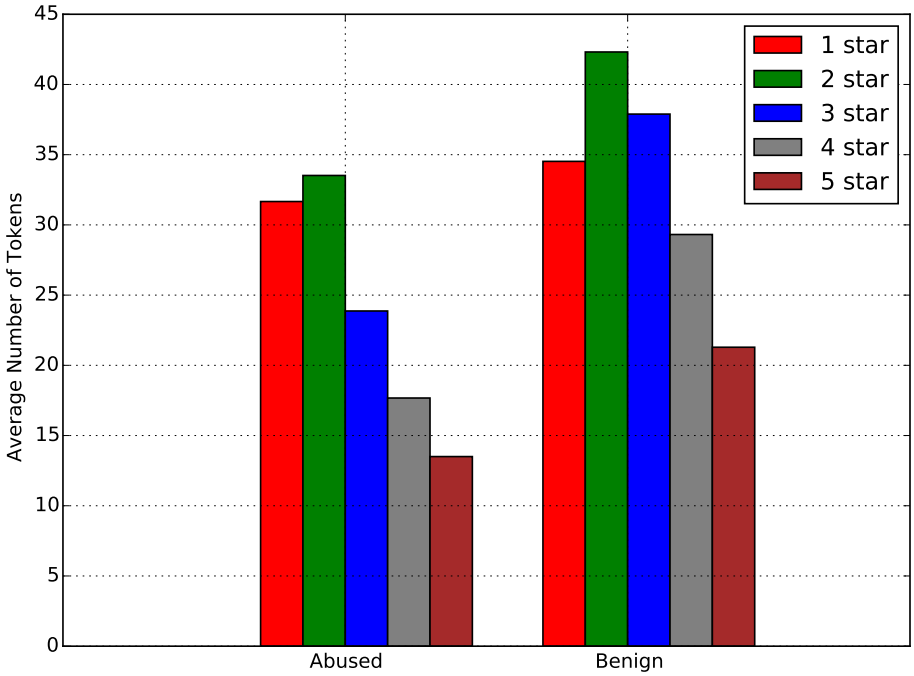

Figure 5.18: Average Review Content Lengths in iTunes UK App Store

reviews are. That's because the higher rating apps get, the less bugs they have and users usually use more words to complain and report bugs in lower level reviews.

2. For iTunes US and UK app stores, average lengths for abused apps are always lower than benign apps. Our explanation is that there are some app-promoting websites[2] and apps[3] are made for recruiting users to review apps and pay them. Intuitively, if users are paid to review an app, their reviews' quality usually is low, which means review contents are short, simple, sometimes meaningless. Those websites do not require users to leave high ratings, therefore, we can see that the average lengths of each rating level for abused apps are always lower. However, this observation does not fit iTunes China app store. First of all, we have not found any similar app-promoting websites in China. Secondly, those websites use Paypal to refund users, but Paypal is not popular in China. Therefore, we think users in China may find difficulty to get paid from those websites.

5.5 Summary

To summarize, we briefly introduce random forest algorithm, which is chosen to be our machine learning algorithm. Random forest runs internal validation so that it could rank features by their importance. We depict the rankings for each app stores and give the explanations of high ranking features. After filtering out some low ranking features, second random forest is used to finalize our training models, which are applied to label new abused apps. New abused apps are evaluated by another two additional criteria and they are merged with our training data in order to do analyses on abused app categories and review contents.

[2] `https://promodispenser.com/`, `https://giftmeapps.com/`, etc.
[3] `https://itunes.apple.com/app/id688637547?mt=8`

Chapter 6

Conclusion And Future Work

In this thesis, we propose a machine learning based approach to detect app rating manipulation. The main purpose of this approach is to provide a way to narrow down the scope of abused app detection, from the whole app store to a relative small size of them.

We firstly discuss our crawler, which helps us gather data from various app stores in different countries. After that, we give the definitions of abused app, malicious user and collusion group and we use an strong-constraint algorithm to detect abused apps with high precision. Even this algorithm potentially could miss some abused apps, it still provides us some abused apps so that we could build training sets. We extract 55 features from our data and use random forest algorithm to rank their importances, so that we are able to figure out which features are critical in abused app detection. It turns out that the feature rankings we get match our definitions and characteristics of abused app and collusion group, which means our features can be used to separate abused apps from benign apps. To improve training performance in terms of speed and accuracy, we select top n features for each app store and apply our training models to the data that we crawled but are not used during the learning procedure, in order to discover more abused apps. Even we lack evidences to prove that average ratings of the newly discovered abused apps have been manipulated, by using two more features(consecutive user IDs and review density), it's reasonable to believe that our training models have practical merits.

Readers may reckon that purchasing apps in app stores is similar to online shop-

ping. We recognize their similarities. They both contain many reviews and rating information and suffer from rating manipulation. However, from our perspectives, they are quite different. App is more sensitive as its quality will change if version changes, hardware and software upgrades, which means data in app stores contain more noises. It's also very normal that one app is perfect on one device but poorly optimized for other devices, which leads to quite different reviews. Moreover, app can be downloaded in and distributed to any countries, but online shopping is usually geographically restricted, which means threat models in different countries might be different. Those two differences bring us more obstacles in app rating detection problem. Therefore, currently, the best we can do is trying to narrow down the scope of investigation done by human, but improving the confidence of our classification will be our future work.

In Section 5.4, we analyze review contents superficially, but more work can be done from the following two perspectives:

1. Use unsupervised learning[29] to detect collusion groups. Since we found that app rating manipulation is always done by large number of malicious users in a relatively short period of time, unsupervised learning is able to divide all reviews into different clusters. Therefore, we may find useful information from those clusters.

2. Nature language processing could be introduced to analyze the review contents. We could use sentimental analysis to judge if reviewer's comment matches its rating score, which might be a good way to classify review as malicious one or benign one. Also, by removing stop words, we will be able to analyze important word frequencies and other stuffs. Although some researchers have done similar researches before[30][31][32], analyzing review contents along is less convincing.

Bibliography

[1] Xie, Z. and S. Zhu *Toward Large-Scale Abused Apps Detection in Mobile App Stores*, *Tech. rep.*

[2] Manning, C. D., P. Raghavan, and H. Schütze *Introduction to information retrieval*.

[3] Hoeffding, W. (1963) "Probability inequalities for sums of bounded random variables," *Journal of the American statistical association*, **58**(301), pp. 13–30.

[4] Allahbakhsh, M., A. Ignjatovic, B. Benatallah, E. Bertino, N. Foo, et al. (2013) "Collusion Detection in Online Rating Systems," in *Web Technologies and Applications*, Springer, pp. 196–207.

[5] Salehi-Abari, A. and T. White (2009) "On the impact of witness-based collusion in agent societies," in *Principles of Practice in Multi-Agent Systems*, Springer, pp. 80–96.

[6] Lehmann, S., M. Schwartz, and L. K. Hansen (2008) "Biclique communities," *Physical Review E*, **78**(1), p. 016108.

[7] Abu-Mostafa, Y. S., M. Magdon-Ismail, and H.-T. Lin *Learning From Data*, chap. The Learning Problem, pp. 3–5.

[8] Kohavi, R. (1995) "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection," Morgan Kaufmann, pp. 1137–1143.

[9] Abu-Mostafa, Y. S., M. Magdon-Ismail, and H.-T. Lin *Learning From Data*, chap. Overfitting, pp. 123–126.

[10] Liu, Y., Y. Yang, and Y. L. Sun (2008) "Detection of collusion behaviors in online reputation systems," in *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, IEEE, pp. 1368–1372.

[11] Bansal, T., B. Chen, and P. Sinha "FastProbe: Malicious User Detection in Cognitive Radio Networks Through Active Transmissions," .

[12] Xie, Z. and S. Zhu *GroupTie: Toward Hidden Collusion Group Discovery in App Store*, *Tech. rep.*

[13] Breiman, L. (2001) "Random forests," *Machine learning*, **45**(1), pp. 5–32.

[14] Dietterichl, T. G. (2002) "Ensemble learning," *The handbook of brain theory and neural networks*, pp. 405–408.

[15] Safavian, S. R. and D. Landgrebe (1991) "A survey of decision tree classifier methodology," *Systems, Man and Cybernetics, IEEE Transactions on*, **21**(3), pp. 660–674.

[16] Breiman, L. (1993) *Classification and regression trees*, CRC press.

[17] Oshiro, T. M., P. S. Perez, and J. A. Baranauskas (2012) "How many trees in a random forest?" in *Machine Learning and Data Mining in Pattern Recognition*, Springer, pp. 154–168.

[18] Cortes, C. and V. Vapnik (1995) "Support vector machine," *Machine learning*, **20**(3), pp. 273–297.

[19] Hagan, M. T., H. B. Demuth, M. H. Beale, et al. (1996) *Neural network design*, Pws Pub. Boston.

[20] Bengio, Y., A. Courville, and P. Vincent (2014) "Representation learning: A review and new perspectives," .

[21] Dietterich, T. G. (2000) "An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization," *Machine learning*, **40**(2), pp. 139–157.

[22] Strobl, C., A.-L. Boulesteix, A. Zeileis, and T. Hothorn (2007) "Bias in random forest variable importance measures: Illustrations, sources and a solution," *BMC bioinformatics*, **8**(1), p. 25.

[23] Segal, M. R. (2004) "Machine learning benchmarks and random forest regression," .

[24] Buckland, M. K. and F. C. Gey (1994) "The relationship between recall and precision," *JASIS*, **45**(1), pp. 12–19.

[25] Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. (2011) "Scikit-learn: Machine learning in Python," *The Journal of Machine Learning Research*, **12**, pp. 2825–2830.

[26] Guyon, I. and A. Elisseeff (2003) "An introduction to variable and feature selection," *The Journal of Machine Learning Research*, **3**, pp. 1157–1182.

[27] Vapnik, V., E. Levin, and Y. Le Cun (1994) "Measuring the VC-dimension of a learning machine," *Neural Computation*, **6**(5), pp. 851–876.

[28] Hatcher, E., O. Gospodnetic, and M. McCandless (2004), "Lucene in action," .

[29] Barlow, H. B. (1989) "Unsupervised learning," *Neural computation*, **1**(3), pp. 295–311.

[30] Ott, M., Y. Choi, C. Cardie, and J. T. Hancock (2011) "Finding deceptive opinion spam by any stretch of the imagination," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, Association for Computational Linguistics, pp. 309–319.

[31] Fu, B., J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh (2013) "Why people hate your app: making sense of user feedback in a mobile app store," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, pp. 1276–1284.

[32] Jindal, N. and B. Liu (2008) "Opinion spam and analysis," in *Proceedings of the 2008 International Conference on Web Search and Data Mining*, ACM, pp. 219–230.