9-2016

Predicting crashing releases of mobile applications

Xin XIA

Emad SHIHAB

Yasutaka KAMEI

David LO
Singapore Management University, davidlo@smu.edu.sg

Xinyu WANG

Published in ESEM '16: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement: Ciudad Real, Spain, September 8-9, 2016.
http://doi.org/10.1145/2961111.2962606

Predicting Crashing Releases of Mobile Applications

Xin Xia[1], Emad Shihab[2], Yasutaka Kamei[3], David Lo[4], and Xinyu Wang[1*]
[1]Zhejiang University, Hangzhou, China; [2]Concordia University, Montreal, Canada
[3]Kyushu University, Nishi-ku, Japan; [4]Singapore Management University, Singapore
{xxia, wangxinyu}@zju.edu.cn, eshihab@cse.concordia.ca,
kamei@ait.kyushu-u.ac.jp, davidlo@smu.edu.sg

ABSTRACT

Context: The quality of mobile applications has a vital impact on their user's experience, ratings and ultimately overall success. Given the high competition in the mobile application market, i.e., many mobile applications perform the same or similar functionality, users of mobile apps tend to be less tolerant to quality issues.
Goal: Therefore, identifying these crashing releases early on so that they can be avoided will help mobile app developers keep their user base and ensure the overall success of their apps.
Method: To help mobile developers, we use machine learning techniques to effectively predict mobile app releases that are more likely to cause crashes, i.e., crashing releases. To perform our prediction, we mine and use a number of factors about the mobile releases, that are grouped into six unique dimensions: complexity, time, code, diffusion, commit, and text, and use a Naive Bayes classified to perform our prediction.
Results: We perform an empirical study on 10 open source mobile applications containing a total of 2,638 releases from the F-Droid repository. On average, our approach can achieve F1 and AUC scores that improve over a baseline (random) predictor by 50% and 28%, respectively. We also find that factors related to text extracted from the commit logs prior to a release are the best predictors of crashing releases and have the largest effect.
Conclusions: Our proposed approach could help to identify crash releases for mobile apps.

CCS Concepts

•**Software and its engineering** → *Software testing and debugging;* •**General and reference** → *Empirical studies; Evaluation;*

Keywords

Mobile Applications, Crash Release, Prediction Model

1. INTRODUCTION

Mobile apps are some of the most popular software systems today. Recent market studies show that the Apple App Store has more than 1.5 million apps and Google Play has over 1.6 million apps [2]. This rapid development of mobile app markets has brought huge benefits. At the same time, competition in the mobile app industry is very high since many apps provide similar functionality, e.g., there are more than 250 weather forecast applications in Google Play[1].

Although mobile apps are considered to be software systems, they are very different from traditional "shrink wrapped" software such as Mozilla Firefox or Microsoft Windows [47]. For example, most mobile apps tend to be small in nature and are developed by smaller and less experienced teams [50]. More importantly, the role of mobile apps has evolved from simple games that entertain us to playing a critical role in our daily lives [40]. For example, mobile apps are increasingly being used in mission-critical systems such as security, healthcare and the financial sector. Therefore, the quality of mobile apps is of critical importance.

At the same time, due to the fact that these mobile apps are released through app stores, mobile apps tend to have more releases compared to traditional applications. To further enhance the competitiveness and attract more users, developers continually evolve their applications and frequently release new versions. For example, the `gothfox_ Tiny-Tiny-RSS` app released 280 versions in the 3 year period from September 2011 to July 2014. Other apps have even shorter release cycles. However, in many cases, mobile developers may release versions of the app that are of poor quality. Prior work showed that some of the most negatively impacting releases, are *crashing releases*, i.e., releases that cause the app to crash [23]. These crashing releases can cause users to uninstall an app and potentially give it a negative rating [16], which in turn impacts the app's revenues. Therefore, identifying crashing releases early on, can help warn mobile app developers about a potential crashing version before it is released and reduce the number of crashing releases. Notice our prediction is used to prioritize the releases rather than replace testing.

In this paper, we perform an empirical study on crashing releases of mobile apps. We perform our study on 10 open source mobile apps from the from F-Droid repository[2]. The 10 apps have 2,638 releases, of which 13% were crashing releases. For each app, we mine its commit data to identify crashing releases. Since our goal is to predict crashing releases, we extract 20 different factors from the mobile app repositories that are grouped into six unique dimensions: complexity, time, code, diffusion, commit, and text. Using the extracted factors, we leverage machine learning techniques to build models that can effectively predict crashing releases. We compare our approach with two baselines: random prediction and majority prediction. Our results show that we are able to achieve an im-

*Corresponding Author

[1]https://play.google.com/store/search?q=weather%20forecast
[2]http://f-droid.org/

Makes stats and browser deck-aware, and a fix crash caused by deck selector

Figure 1: Example commit log between Releases 2.2alpha86 and 2.2alpha87 of the *ankidroid_Anki-Android* mobile app.

Prevent crashes in onPostExecute() by not calling onCreateDialog(). Force startLoadingCollection() to re-sync with AnkiDroidApp.getCol()

Figure 2: Example commit log between Releases 2.3alpha16 and 2.3alpha17 of the *ankidroid_Anki-Android* mobile app.

provement over random prediction by 50% and 28% in terms of F1 and AUC score, and an improvement over majority prediction by 39% in terms of AUC score, respectively. Note that the precision and recall for the majority prediction baseline are both 0, and thus the F1-score for majority prediction cannot be calculated. In addition, we analyzed the factors that best indicate crashing releases and find that factors in the text dimension, i.e., text in the commit messages, are the best predictors.

The main contributions of the paper can be summarized as follows:

1. We propose the problem of crashing release prediction for mobile apps. To the best of our knowledge, this is the first work to predict crashing releases for mobile apps. We envision that our approach can be used to predict the releases that are highly likely to be crashing so that they can be prioritized for testing.
2. We develop factors that can be easily mined from mobile app repositories to predict crashing mobile app releases. We use the developed factors to accurately predict crashing releases and investigate the factors that best indicate crashing releases.
3. We perform an empirical study on 10 open source mobile apps and our experimental results show that our approach achieves an improvement over two other baseline approaches.

The remainder of the paper is organized as follows. We describe the motivation of our study in Section 2. We present our empirical study data and empirical study setup in Section 3 and 4, respectively. We present our empirical study results in Section 5. We discuss additional points on the benefits and limitations of our approach in Section 6. We discuss related work in Section 7. We conclude and mention future work in Section 8.

2. MOTIVATION

Although our intuition and experience tells us that mobile app developers have crashing releases and care about minimizing such crashing releases, we wanted to make sure that this was a real problem that mobile app developers face and care about (note that prior work only showed that crashing releases negatively impact users [23]). Hence, we approached the task of investigating whether crashing releases is a real problem in two ways. First, we examined the commit logs and found examples where developers explicitly mention that crashing releases occur and are swiftly fixed. Second, we emailed a number of developers to get their opinion on the importance of this problem. We elaborate on each of these two tasks in the following subsections.

Examples of Crashing Releases in Commit Logs. Figure 1 presents example commit logs between Releases 2.2alpha86 and 2.2alpha87 of the *ankidroid_Anki-Android* mobile app. Figure 2 presents example commit logs between Releases 2.3alpha16 and 2.3alpha17 of the *ankidroid_Anki-Android* mobile app. We note that in both cases presented in the Figures, the developers are mentioning fixes for crashes that were introduced by earlier versions. Obviously these crashes were critical enough for the developers to not only fix them, but to fix them quickly. In the example shown in Figure 1, the fix to the crashing release was performed in 5 days and in the example shown in Figure 2, the fix was committed in 4 days.

Developer's Opinions of Crashing Releases. As mentioned earlier, we also asked mobile developers at Hengtian[3] and the developers of the mobile app projects `ankidroid _Anki-Android` and `qii_weiciyuan`. We sent emails to 25 developers asking them a very simple question:

Do you think it is necessary to predict the crash releases in mobile apps? If so, why?

In total, 6 developers replied to our email. All the 6 developers, denoted as D1 to D6, agreed that predicting crashing releases is an important problem. We summarize their replies as follows:

1. **Release Cycles**: Since the release cycles for mobile apps are very short (e.g., one release per week (D5)), "*we do not have enough time to test all aspects of the apps in each release*" (D2). "*If there is a tool that can tell us whether the release has a high likelihood of containing a crash, we will pay more attention to the release and spend more time inspecting the code*" (D4).
2. **Inspection Cost**: In traditional desktop applications, for each release, it may modify a large number of LOCs (e.g., more than 1M LOCs). In such cases, predicting crashing releases for desktop applications is not necessary *since developers will need to inspect a large number of LOCs* (D3). However, *"in mobile apps, in each release our team only modified several thousand LOCs. If a tool can tell us there is a crash in the release, it is possible for us to manually inspect the code"* (D3). Note *"it is impossible for us to inspect the code in all of the releases since we have too many releases, and practically speaking, we can only focus on the problematic releases"* (D6).
3. **Supplementary Means to Testing:** Eventhough some apps may have received enough testing, *some of the crashes are still hard to detect* (D1). A crash release prediction tool *can be a supplementary means to testing* (D1). After the testers complete the testing, the tool can be used to ensure the quality of the released version.

3. EMPIRICAL STUDY DATA

This section details the data used in our study. First, select the relevant mobile applications (application collection). Then, we identify the releases of the collected applications (release identification) and use heuristics to determine the crashing releases (crash release determination). We detail each step of the data collection below.

Mobile App Collection: To perform our study, we need to obtain a number of mobile applications. To do so, we collected the mobile applications available on the Free and Open Source Software (FOSS) repository F-Droid. F-Droid is widely used in a number of software engineering studies (c.f., [1, 26, 29, 36]). Each mobile application available on F-Droid has a corresponding wiki page which

[3]Hengtian is a large software outsourcing company in China, with more than 2,000 employees. More details can be found in http://www.hengtiansoft.com/?lang=en

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="org.andstatus.app"
        android:versionName="13.0" android:versionCode="113"
        android:installLocation="internalOnly">
```

Figure 3: The manifest file of andstatus.

provides details about the app. For us, wiki pages are very important since they provided us with the number and dates of the releases for each app. In total, we crawled the data for 900 mobile apps. This data included: 1) the wiki page of the app, 2) the link to the source code repository, and 3) the source code of the app.

Since some of the mobile apps did not provide all of the information we required (e.g., there was no link to the source code repository available), we ended up with a total of 632 apps. Of these 632 apps, 466 mobile apps are managed by git, 40 are managed by mercurial, and 126 are managed by svn. Since some of the tools we used (e.g., the tools used to extract factors from commits) only support Git repositories [49], we only consider the apps managed by Git, i.e, 466 apps.

Release Identification: According to the Google Play developer documentation[4], when a new version is released, the version attribute in the manifest file is also changed. More specifically, the android:versionCode and the android:versionName attributes must be changed. Figure 3 presents an example manifest file from the andstatus app. The version code is "113", and the version name is "13.0". With this information, we mine the commit logs in the source code repository, and collect the dates of commit logs. We use the dates from the commits and the dates for the version name or version code in the manifest file (which we obtain from the wiki page), to determine which commits belong to a release.

To filter out toy examples and non-active apps, we manually check the status of apps by reading their release notes and checking their download records. We select apps that existed in GitHub for at least 2 years, since our prediction models require historical data of the mobile apps. After this step, we are left with a total of 22 apps. Next, we further filtered out the apps that have less than 100 releases, since we require enough instances of crashing releases and non-crashing releases to train our models. This step further narrowed the number of apps to 10. Although our requirement of having more than 100 releases may seem restrictive, it is important to note that our approach is most useful for apps with a large number of releases (apps with a small number of releases can afford to check every release).

Crashing Release Determination: Once we selected the 10 mobile apps, we extracted all releases for these apps. Then, for each release, we extract all its associated commits along with the commit messages from the Git repo of the project. To identify the crashing' releases, similar to prior work, we use keywords found in the commit messages [25,45,52]. In particular, we looked for the keywords "crash", "crashed", "crashing", and "crashes". Note that keyword searches may increase the number of false positives and false negatives. To reduce the number of false positives, for each identified crashing release, the first and third author manually checked the commit logs and modified source code, and we only find 5 releases are misclassified. To reduce the number of false negatives, for these non-crash releases, the first and third author also randomly choose and manually checked 30% (i.e., 688)of the releases from these non-crash releases, and we find all of these releases labeled as non-crash releases are correctly labeled. It is important to note here that if we find these keywords in release x for example, then we mark release $x-1$ as the crashing release. This is because developers will often mention that they are fixing a crash in a previous release. Based on our observations, in practice crashing releases are often followed by another release very swiftly. This is done to minimize the negative impact of the crashing release. Using the method above, we identify all the potential crashing releases. Next, we manually check the commit logs, fixed bugs, release notes and user reviews of the crashing releases and their subsequent releases to makes sure the releases identified as crashing in fact are.

[4]https://developer.android.com/tools/publishing/preparing.html

Table 1: Statistics for the collected mobile applications. The last row shows the total number of releases, the total number of crashing releases, and the percentage of crash releases.

Project	Time Period	# R	# C	% C
ankidroid_Anki-Android	2009/06–2014/09	597	97	16%
bpellin_keepassdroid	2009/01–2013/09	149	23	15%
BrandroidTools_OpenExplorer	2011/12–2013/05	156	28	18%
freezy_android-xbmcremote	2009/08–2013/12	392	19	5%
gothfox_Tiny-	2011/09–2014/07	280	36	13%
guardianproject_Gibberbot	2010/07–2014/09	233	26	11%
mariotaku_twidere	2012/04–2014/05	262	34	13%
mtotschnig_MyExpenses	2011/05–2014/09	241	34	14%
qii_weiciyuan	2012/07–2014/07	205	39	19%
WSDOT_wsdot-android-app	2010/05–2014/07	123	8	7%
Total		**2,638**	**344**	**13%**

Table 1 presents the statistics for each mobile app in our data set. The columns correspond to the time period, the number of releases (# Releases), the number of crash releases (# Crash Releases), and the percentage of the crash releases (% Crash Releases) for the 10 mobile applications. In total, we collect 2,638 releases across the 10 mobile applications, and among them, 344 releases were labelled as crashing releases, which accounts for 13% of the releases, on average. Note that the number of crash releases is much less than the number of the non-crash releases. We refer to this phenomenon as the class imbalance phenomenon [18]. Due to the class imbalance phenomenon, we anticipate that predicting crashing releases with high accuracy will be a difficult task.

4. EMPIRICAL STUDY SETUP

Now that we have observed that the percentage of crashing releases is not negligible, the goal of our study is to be able to effectively predict crashing mobile releases early on so they can be avoided. We extract a number of factors about the mobile releases and use them to perform our prediction. We are interested in knowing how effective the extracted factors are at predicting crashing releases. Furthermore, we determine the best indicators of crashing releases and examine their effect. We formalize our study in the following research questions:

RQ1: Can we effectively predict crashing mobile releases?

RQ2: What factors are the best indicators of a crashing mobile release? What is the relationship of these factors with a crashing mobile release?

4.1 Factors Studied

To predict whether a release is a crashing release, we consider 20 factors grouped into six unique dimensions: complexity, time, code, diffusion, commit, and text. These factors are derived from the source control repository data of a mobile application. Table 2 presents the summary of the factors used in our study. We use these 20 carefully developed factors because either 1) prior work showed that they perform well in predicting defects [17, 19, 21, 35, 58, 60] or 2) intuitively, the factors made sense to include when predicting a crashing mobile release. That said, we are the first to investigate the effectiveness of these factors in predicting crashing mobile app releases and encourage future factors to be developed to help predict crashing mobile app releases. Next, we discuss each dimension in detail.

Complexity Dimension: If the source code in a release has high complexity (e.g., high number of data flows in an applications), the code will be harder to maintain, which may increase the chance of

Table 2: Factors used to identify crashing releases

Dimension	Name	Definition	Rational
Complexity	Cyclomatic	The number of branching paths within code in all the source code files in a release.	High complexity indicates that the code may be difficult to modify and has a higher chance of containing/introducing a defect [42, 54].
Time	PreDays	The number of days since the previous release.	The shorter the number of days passed from the previous releases, the high chance the current release is a non-crashing release since developers need to fix the crashes appeared in the previous release.
Code	LA	Number of lines added in a release	The higher the number of lines added, the more likely a defect can be introduced, increasing the chance of a crash release [39, 41].
	LD	Number of lines deleted in a release	The higher the number of lines deleted, the higher chance that a previous bug recurs, or important functionality is removed [39, 41].
	SIZE	Total number of lines of code in the current release	An increase in the SIZE causes an increase in complexity, and high complexity may result in crash releases [39, 41].
	SAME	Number of source code files that are modified by both the current and the previous release	The higher the number of the same source code files that are modified, the higher the chance that the current release fixes existing defects in the previous release.
	CUR_file	Number of source code files in the current release	An increase in the number of source code files between two releases (CUR_file - PREV_file) may indicate that the current release adds more functionality and/or performs many fixes, which may result in crash release.
	PREV_file	Number of modified source code files in the previous release	
Diffusion	Top_NS	Number of unique subsystems changed between two releases	The more functionalities there are in an application release, the more prone it is to fail. We use the subsystem as a proxy for a feature. Releases that contain many modifications at the subsystem level are more likely to be crash releases.
	Bottom_NS	Number of unique subsystems changed between two releases	
	NF	Number of unique files that have changed between two releases	Releases that touch too many files have a higher chance of being a crash releases.
	File_entropy	Distribution of modified files across the release	Releases with high entropy are more likely to be crash releases, since a developer needs to inspect large number of scattered changes (at file or code levels) across each file.
	Churn_entropy	Distribution of modified code across the application	
Commit	NC	Number of commits	The more number of commits, the more functionality added and/or defects fixed. This may increase the chance of a crash release.
	NFC	Number of commits which fix bugs	The higher the number of commits that fix defects, the higher the chance that a future defect may be introduced, leading to a higher chance of a crash release [25, 45].
Text	Fuzzy_score	Fuzzy set scores of commit logs	The text content of commit logs between the previous release and the current release. We represent the text of the commit logs using fuzzy set scores [65], naive Bayes scores [34], naive Bayes multinomial scores [34], discriminative naive Bayes multinomial scores [53], and complement naive Bayes scores [46]. A higher score means the release is more likely to be a crash release.
	NB_score	Naive Bayes scores of commit logs	
	NBM_score	Naive Bayes Multinomial scores of commit logs	
	DMN_score	Discriminative naive Bayes Multinomial scores of commit logs	
	COMP_score	Complement naive Bayes scores of commit logs	

a crashing release. Also, the defect prediction literature showed that complexity (e.g., cyclomatic complexity) is a good predictor of defect-prone modules [42, 54]. We expect that the complexity dimension can be used to determine the likelihood of a crashing release. As shown in Table 2, we propose McCabe's cyclomatic complexity in the complexity dimension. McCabe's cyclomatic complexity can be measured directly from the source code in the current release using standard code analysis tools. In our study we used the Understand tool[5] to determine the code complexity.

Time Dimension: If the time period between the previous release and the current release is short, the current release has a higher chance to be a non-crash release since it may fix the crashes appeared in the previous release. We expect that the time dimension can be used to determine the likelihood of a crashing release. As shown in Table 2, we propose the number of days since the previous release (PreDays) to make up time dimension, PreDays is computed by counting the number of days between the previous and current release.

Code Dimension: If a release has large changes done to its source code, then it has a higher probability to introduce more defects, which in turn may cause a crashing release [42]. Previous defect prediction studies show that the size of changes (e.g., number of lines of code added or deleted) is a good indicator of defects [39, 41]. Also, if the current release modifies many of the same source code files as the previous release, this may be an indication that a lot of fixing is being done, which may indicate that the current release is a good release. We expect that the code dimension can be used to determine the likelihood of a crashing release. As shown in Table 2, we propose 6 factors which make up the code dimension. These factors can be measured directly from the source control repository by comparing the difference between 2 releases.

Diffusion Dimension: In general, a highly distributed release is more difficult to understand, and requires more work to inspect all the locations that are changed. Also, previous defect prediction studies found that the number of subsystems touched is an indicator of defects [38], and scattered changes are good indicators of defects [17]. We expect that the diffusion dimension can be used to determine the likelihood of a crash release. As shown in Table 2, we propose 5 factors that make up the diffusion dimension.

We use the top directory name and bottom directory name as the subsystem name to measure Top_NS and Bottom_NS, respectively. For example, if a commit changes a file in the path "src/app/easytoken/MainActivity.java", then its top directory name is "src/", and the bottom directory name is "src/app/easytoken/". For the i^{th} release, we compute the set of top directory name Top_i and bottom directory name $Bottom_i$. Then, for two consecutive releases (i^{th} and $(i+1)^{th}$ releases), Top_NS= $|Top_i \cap Top_{i+1}|$, and Bottom_NS =$|Bottom_i \cap Bottom_{i+1}|$.

To measure the file and churn entropy, we use the measures similar to the measures proposed by Hassan [17]. Entropy is computed as: $H(P) = -\sum_{k=1}^{n}(p_k \times log_2\ p_k)$. In the above equation, n is the number of files changed in the release, $p_k \geq 0$ is the probability for file k, and p_k satisfies $(\sum_{k=1}^{n} p_k) = 1$. p_k can be computed at different levels of granularity: to compute file entropy, p_k is the proportion that file k is modified in the changes[6] in a release, to compute churn entropy, p_k is the proportion that the number of lines of code in file k that is modified in the changes in a release. Entropy aims to measure the distribution of the release across the different files or the lines of code in the files. The higher the entropy, the larger the spread of a release.

Commit Dimension: If a release has a large number of commits, we conjuncture that the release has a high probability to be a crashing release. This belief is driven by the intuition that more commits mean more changes (e.g., bug fixes, new functionalities) to the application, which may introduce more problems (e.g., bugs) in the

[5]https://scitools.com/

[6]In a release, there are multiple commits, and each commit corresponds to a change.

release. As shown in Table 2, we propose 2 factors which make up the commits dimension: the number of commits (NC), and the number of bug fixing commits (NFC). NC is computed by counting the number of commits in the current release, and NFC is computed by counting the number of commits which contains the strings "fix", "'error", "fault", "crash", "issue", or "bug" [25, 45, 52].

Text Dimension: In the text dimension, we extract a number of textual factors from commits, and we convert the terms that appear in commit logs into numerical values. In total, we compose 5 different factors in the text dimension, which correspond to the textual scores based on 5 different classifiers.

During the release of an application, many commits may be submitted to fix defects or implement new features. Since prior work showed that text descriptions can help with bug prediction [60], we believe that analyzing the natural language description in the commit logs may help us identify crashing releases. We refer to the factors extracted from the commit logs as textual factors. To extract the textual factors, we first tokenize the text in the commit logs into words, phrases, symbols, or other meaningful name element tokens. Next, we remove the stop-words such as "I", "the", "he", which carry little value to discriminate crashing releases. Then, we perform stemming on the tokens, which reduces inflected (or sometimes derived) tokens to their stem, base or root form (e.g., "write", "wrote", and "written" are reduced to the root form "writ"). We use the resulting textual tokens and count the number of times each token appears in the commit logs of a release.

Our textual dataset contained a large number of tokens. Since a high number of tokens (factors) can cause, what is known as the curse-of-dimensionality [15], we follow previous studies (e.g., [60]) and we convert the textual factors into numerical values, which associate each textual feature with a value that it indicates a crashing releases and another value that the textual feature is indicative of a non-crashing release. For example, the word bug may have a score x that it indicates a crashing release and a score y that it indicates a non-crashing release.

To come up with the scores for the different textual features, we first divide our data into a training set and a testing set. Then, the training data set is split into two training subsets by leveraging stratified random sampling, so that the distribution and number of non-crashing and crashing releases in both training subsets is the same [60]. Next, we extract the textual factors from the training subsets, and create word frequency tables based on the extracted factors. We train a classifier with the first training subset, and use it to obtain the textual scores on the second training subset. We also train a classifier with the second training subset, and use it to obtain the textual scores on the first training subset. We use this strategy to avoid the textual classifiers from being biased toward their training sets [60]; otherwise, our models may lead to optimistic (unrealistic) values for the textual scores. In the prediction phase, for a new release, we leverage the text mining classifiers that are built on all of the training releases to compute the values of the textual factors.

In this paper, we use 5 types of textual classifiers to calculate the scores of the textual features, which are: fuzzy set classifier [65], naive Bayes classifier [34], naive Bayes multinomial classifier [34], discriminative naive Bayes multinomial classifier [53], and complement naive Bayes classifier [46], and we denote the textual scores using these 5 classifiers as fuzzy_score, NB_score, NBM_score, DMN_score, and Comp_score respectively. In the following paragraphs, we present the details of the 5 classifiers.

Fuzzy Set Classifier: A fuzzy set classifier considers the concurrence of words and labels (i.e., crashing or non-crashing release) as good indicators of identifing crashing releases. If a word always appears in a crashing release in the training set, then for a new release, if the same word appears, the release will be assigned a high probability of being a crashing release. We denote the label of the i^{th} release as l_i, and we represent the textual factors in the i^{th} release as a vector of weights denoted by $R_i = \langle f_{t_1,i}, f_{t_2,i}, \cdots, f_{t_m,i} \rangle$ ($f_{t_n,i}$ denotes the n^{th} factor of R_i), where $f_{t_n,i} = 1$ represents that the word token t_n appears in the commit logs of the i^{th} release; and $f_{t_n,i} = 0$ else wise. Based on these notations, we define a label-term affinity score as follows:

DEFINITION 1. *(***Fuzzy Label-Word Affinity Score.***) Consider a historical release data collection D, and a set of labels L. For each label $l \in L$, and word token $t \in D$, the fuzzy label-word affinity score of l and t, denoted as $Aff(l,t)$, is computed as $Aff(l,t) = \frac{n_{l,t}}{n_l + n_w - n_{l,t}}$. $n_{l,t}$ denotes the number of releases whose commit logs contain the word token t and are of the label l, n_l denotes the number of releases that have the label l and n_t denotes the number of releases that contain the word token t.*

In our paper, we have two labels, crashing release C and non-crashing release NC. Thus, for each word token $t \in D$, we have two fuzzy label-word affinity scores, $Aff(C,t)$, and $Aff(NC,t)$. For a new release $R_{New} = \langle f_{t_1,new}, f_{t_2,new}, \cdots, f_{t_m,new} \rangle$, we define the label-release affinity score as follows:

DEFINITION 2. *(***Label-Release Affinity Score.***) For a new release R_{New}, the label-release affinity score $Fuzzy(l, New)$ is computed as the combinations of the label-word affinity scores $Aff(l,t)$ of its associated terms w: $Fuzzy(l, New) = 1 - \prod_{t \in New}(1 - Aff(l,t))$. t denotes the word tokens that appear in the commit logs of the new release.*

For a new release, we have 2 label-release affinity scores, i.e., $Fuzzy(C, New)$ and $Fuzzy(NC, New)$, the fuzzy score (Fuzzy _score) to be a crash release for fuzzy set classifier is $Fuzzy(New) = \frac{Fuzzy(C,New)}{Fuzzy(C,New)+Fuzzy(NC,New)}$.

Naive Bayes Classifier: In Naive Bayes classifier [34], we consider the number of times a word as good indicators to identify crashing releases. Previous studies show that Naive Bayes is a fast and effective text classification technique [34]. Similar to the fuzzy set classifier, we represent the textual factors in the i^{th} release as a binary vector denoted by $R_i = \langle f_{t_1,i}, f_{t_2,i}, \cdots, f_{t_m,i} \rangle$ ($f_{t_n,i}$ denotes the n^{th} factor of R_i), where $f_{t_n,i} = 1$ represents that the word token t_n appears in the commit logs of the i^{th} release; and $f_{t_n,i} = 0$ else wise. For each word token t_n, we count the number of times that t_n appears in the crashing releases as $Crash_n$, and the number of times that t_n appears in the non-crashing releases as $NonCrash_n$. Then, the probability that w_j is related to crashing releases as $P(t_n) = \frac{Crash_n}{Crash_n + NonCrash_n}$.

After the naive Bayes classifier is built, we compute the NB_score of the commit logs in a new release New as $NB(New) = \frac{\prod P(t_n)}{\prod P(t_n) + \prod (1 - P(t_n))}$, where t_n refers to the word in the commit logs of New.

Naive Bayes Multinomial Classifier: Naive Bayes multinomial (NBM) is one of the variants of naive Bayes algorithm which builds a classifier based on multinomially distributed data [34]. McCallum et al. empirically find that NBM performs better than Naive Bayes in text classification tasks [34]. Different from Naive Bayes, NBM leverages the word frequency information to perform text classification, i.e., we represent the textual factors in the i^{th} release as a vector of weights denoted by $R_i = \langle w_{t_1,i}, w_{t_2,i}, \cdots, w_{t_m,i} \rangle$, where $w_{t_n,i}$ is the number of times the word token t_n appears in the commit logs of the i^{th} release. We use the implementation of naive Bayes multinomial on top of Weka [14]. For a new release R_{New}, we denote the likelihood score to be a crash release of the naive Bayes multinomial classifier as NBM_score $NBM(New)$.

Discriminative Naive Bayes Multinomial Classifier: Discriminative naive Bayes multinomial (DMN) is one of the variants of naive Bayes algorithm multinomial which learns parameters by discriminatively computing frequencies from data [53]. Su et al. empirically find that compared with Naive Bayes, DMN converges quickly, and does not suffer from the over-fitting problem [53]. We use the implementation of DMN on top of Weka [14]. For a new release R_{New}, we denote the likelihood score to be a crash release of the discriminative naive Bayes multinomial classifier as DMN_score $DMN(New)$.

Complement Naive Bayes Classifier: Complement Naive Bayes Classifier (COMP) is one of the variants of Naive Bayes multinomial (NBM) algorithm which uses various transformation approaches in information retrieval community to improve the prediction performance for textual data [46]. Different from Naive Bayes multinomial that assumes the text follows multinomial distribution. COMP proposes the usage of heuristic solutions (i.e., term frequency, document frequency, and document length transformation) to model better text [46]. We use the implementation of complement naive Bayes on top of Weka [14]. For a new release R_{New}, we denote the likelihood score to be a crash release of the complement naive Bayes classifier as Comp_score $COMP(New)$.

4.2 Prediction Model

For each of our empirical study projects, we use our proposed factors to train a Naive Bayes classifier to predict whether a release will be a crashing release or not. We also compare our prediction model with four other classifiers namely: decision tree, kNN, and Random Forest.

Naive Bayes (NB): We use the Naive Bayes classifier for two purposes: to convert textual information into numerical values (i.e., to obtain the probability that the textual contents in the commit logs are related to crash release), and to build a prediction model that predicts if a release will be a crashing release or not.

Decision Tree (DT): C4.5 is one of the most popular decision tree algorithms [15]. A decision tree algorithm classifies data points by comparing their factor with various conditions captured in the nodes and branches of the tree.

K-Nearest Neighbor (kNN): K-nearest neighbor is an instance-based algorithm for supervised learning, which delays the induction or generalization process until classification is performed [15]. We use the Euclidean distance as the distance metric, and since the performance of kNN may be impacted by different values of k, we set k from 1 to 10, and report the best performance (in terms of F1-score) among the 10 values of k.

Random Forest (RF): Random forest is a kind of combination approach, which is specifically designed for the decision tree classifier [5]. The general idea behind random forest is to combine multiple decision trees for prediction. Each decision tree is built based on the value of an independent set of random vectors. Random forest adopts the mode of the class labels output by individual trees.

4.3 Dealing with Data Imbalance

As can be noted, the number of crashing releases is much less than non-crashing releases (i.e., we have a major class imbalance in the data). For example, in `WSDOT_ wsdot-android-app`, only 9% of the releases are crashing releases. In this paper, we perform both over- and under-sampling to alleviate the imbalance issue. In over-sampling, we increase the number of the minority class instances (in our case, crashing release). And in under-sampling, we decrease the number of the majority class instances (in our case, non-crashing releases). Previous studies (e.g., [8]) recommend that we perform both under- and over-sampling, since under-sampling may lead to useful data being discarded and over-sampling may lead to over-fitted models. We perform both over- and under-sampling on the training data and predict using a non-balanced test data set (i.e., we do not resample the testing data). We use the resampling algorithm in Weka [14] to do the over- and under-sampling on the training data, and we use the option "-B 1.0" (ensure the class distribution is uniform in the output data), and "-Z 100" (the final sample size is the same as the original dataset). After the resampling, the number of crash releases and non-crash releases are the same in the training set.

4.4 Experimental Evaluation

There are four possible outcomes for a release in the test data: a release is classified as a crashing release when it truly is a crashing release (true positive, TP); it can be classified as a crashing release when it is actually a non-crashing release (false positive, FP); it can be classified as a non-crashing release when it is actually a crashing release (false negative, FN); or it can be classified as a non-crashing release and it truly is a non-crashing release (true negative, TN). Based on these possible outcomes, precision, recall and F1-score are defined as:

Precision: is the proportion of releases that are correctly labeled as crashing releases among those labeled as crashing releases, i.e., $P = TP/(TP + FP)$.

Recall: is the proportion of crashing releases that are correctly labeled, i.e., $R = TP/(TP + FN)$.

F1-score: is a summary measure that combines both precision and recall - it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision), i.e., $F = (2 \times P \times R)/(P+R)$. F1-score is the harmonic mean of precision and recall, which is used in many software analytics studies [19, 24, 43, 56, 62].

AUC: In addition to the F1-score, we also use the Area Under the Receiver Operating Characteristic Curve (AUC) to evaluate the effectiveness of our approach. AUC is a commonly-used measure to evaluate classification performance, and many other software engineering studies also use AUC as an evaluation metric [27, 28, 48, 55]. The larger the AUC is, the better is the performance of a classification algorithm.

5. EMPIRICAL STUDY

In this section, we answer the research questions posed earlier[7].

RQ1: Can we effectively predict crashing mobile releases?

Motivation: In order to avoid a negative user experience and poor ratings, we would like to effectively identify crashing releases early on so they can be avoided (or more quality assurance efforts can be targeted towards such releases). Therefore, we use our proposed factors and build prediction models to examine whether it is feasible to build accurate models that help to predict crashing releases.

Approach: We implement our proposed approach on top of the Weka tool [14]. We use 100 times stratified 10-fold cross validation to estimate the accuracy of our models. In stratified 10-fold cross validation we randomly divide the dataset into ten folds. Of these ten folds, nine folds are use to train the classifier, while the remaining one fold is used to evaluate the performance. The class distribution in the training and testing datasets is kept the same as the original dataset to simulate real-life usage of the algorithm. We run 10-fold cross validation 100 times to further reduce the bias due to training set selection. To evaluate their performance, we use the precision, recall, F1, and AUC metrics. The reported performance

[7]The datasets used in this paper can be downloaded in https://www.dropbox.com/s/hr6amcyssuj194c/dataset.zip?dl=0

Table 3: Precision, recall, and F1-score for our approach (our) compared with the baseline approaches. The best F1-score and AUC values are highlighted in bold.

Projects	Naive Bayes Model (our)				Random Prediction Model				Majority Prediction Model			
	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC
ankidroid_Anki-Android	0.18	0.52	**0.26**	**0.50**	0.16	0.50	0.25	0.50	0.00	0.00	NA	0.49
bpellin_keepassdroid	0.18	0.61	**0.28**	**0.61**	0.15	0.50	0.24	0.50	0.00	0.00	NA	0.45
BrandroidTools_OpenExplorer	0.22	0.79	**0.34**	**0.70**	0.18	0.50	0.26	0.50	0.00	0.00	NA	0.46
freezy_android-xbmcremote	0.21	0.58	**0.31**	**0.70**	0.05	0.50	0.09	0.50	0.00	0.00	NA	0.47
gothfox_Tiny-Tiny-RSS-for-Honeycomb	0.15	0.56	**0.24**	**0.60**	0.13	0.50	0.20	0.50	0.00	0.00	NA	0.46
guardianproject_Gibberbot	0.12	0.73	**0.20**	**0.52**	0.11	0.50	0.18	0.50	0.00	0.00	NA	0.45
mariotaku_twidere	0.20	0.50	**0.29**	**0.63**	0.13	0.50	0.21	0.50	0.00	0.00	NA	0.46
mtotschnig_MyExpenses	0.17	0.68	**0.27**	**0.59**	0.14	0.50	0.22	0.50	0.00	0.00	NA	0.46
qii_weiciyuan	0.41	0.67	**0.51**	**0.78**	0.19	0.50	0.28	0.50	0.00	0.00	NA	0.48
WSDOT_wsdot-android-app	0.19	0.63	**0.29**	**0.78**	0.07	0.50	0.12	0.50	0.00	0.00	NA	0.39
Average	**0.20**	**0.62**	**0.30**	**0.64**	**0.13**	**0.50**	**0.20**	**0.50**	**0.00**	**0.00**	**NA**	**0.46**

of the models are the average of the 100 times stratified 10-fold cross validation. These metrics are compared to the performance of our baseline model.

Here, we choose two baseline models: random prediction, and majority prediction. In random prediction, it randomly predicts crash releases. The precision for random prediction is the percentage of crash release in the data set. Since the random prediction model is a random classifier with two possible outcomes (e.g., crash/non-crash release), its recall is 0.50. Majority prediction always predicts the label of an instances as the majority class (in our case, majority prediction will predict every release as a non-crash release). In majority prediction, its TP will be 0 since none of the releases are predicted as crash releases, and thus its precision and recall are both 0, which in turns cause the F1-score to be NA. To compute the improvement of our approach over the baseline approaches, we denote the evaluation metric score of our approach as our, and the score of the baseline approach as $baseline$. Then, the improvement is computed as $\frac{our-baseline}{baseline} \times 100\%$.

Results: Table 3 presents the precision, recall, F1, and AUC scores for our approach compared with the baseline approach for the 10 mobile apps, respectively. On average across the 10 projects, our approach achieves precision, recall, F1-score, and AUC scores of 0.20, 0.62, 0.30, and 0,64, respectively. Our approach improves over the random prediction by 54%, 24%, 50%, and 28% in terms of precision, recall, F1-score, and AUC scores, respectively. Also, compared with majority prediction, considering its precision, recall, and F1 scores are 0, 0, and NA, our approach improve the AUC score of majority prediction by 39% on average across the 10 projects.

We notice for freezy_android-xbmcremote and WSDOT_wsdot-android-app, the improvements of our approach over random prediction are 251% and 156% in terms of F1-score. We find that only 5% and 7% of releases are crash releases in these two datasets, i.e., these two datasets are more imbalanced than the other datasets. This indicates that our approach is best suited for highly imbalanced cases. To test whether the improvement of our approach over random prediction on highly imbalanced datasets (i.e., freezy_android-xbmcremote and WSDOT_wsdot-android-app) are higher than these on the other datasets is statistically significant, we apply Wilcoxon Rank Sum test [61], and the p-value we achieve is 0.0222. This indicates that the improvement is statistically significant at the confidence level of 95%. Moreover, we also use Cliff's delta [6], which is a non-parametric effect size measure that quantifies the amount of difference between the two groups. The Cliffąs delta is 0.875, which corresponds to a large effect size.

Considering that only a small proportion of releases are crash release (on average across the 10 apps, 13% of the releases are crash releases), the data imbalance phenomenon makes the prediction of crashing releases a hard task. Some previous studies on software engineering (c.f., [57, 59, 63]) also achieved similar F1-scores, and in the future, we plan to improve the F1-score of our approach further by designing a better algorithm and considering more factors.

Table 4: Top-5 factors which achieve the highest IncNodePurity (Inc.) and effect scores

Factor	andkidroid.		Factor	bpellin.	
	Inc.	Effect		Inc.	Effect
Fuzzy_sc.	7.46	1.00	DMN_sc.	1.46	0.80
Cyclomatic	7.22	0.23	Cyclomatic	2.29	0.44
DMN_sc.	6.88	0.21	PreDays	1.42	0.22
LA	4.79	0.21	CUR_file	1.30	0.23
CUR_file	4.30	0.22	NBM_sc.	1.29	0.23

Factor	Brandroid.		Factor	Freezy.	
	Inc.	Effect		Inc.	Effect
NBM_sc.	2.28	1.00	DMN_sc.	1.99	1.00
Cyclomatic	1.78	0.99	LA	1.69	0.00
CUR_file	1.65	0.99	Cyclomatic	1.64	0.00
SAME	1.29	1.00	Fuzzy_sc.	1.11	0.02
LA	1.28	1.00	SAME	0.81	0.00

Factor	gothfox.		Factor	guardian.	
	Inc.	Effect		Inc.	Effect
Fuzzy_sc.	3.80	0.93	NBM_sc.	2.06	0.99
LA	2.47	0.43	LA	1.96	0.99
NBM_sc.	2.32	0.43	Chrun_en.	1.54	0.99
Chrun_en.	1.75	0.44	CUR_file	1.50	0.98
LD	1.62	0.44	Fuzzy_sc	1.25	0.86

Factor	mariotaku.		Factor	mtotschnig.	
	Inc.	Effect		Inc.	Effect
NBM_sc.	2.91	0.02	CUR_file	2.33	0.85
Fuzzy_sc.	2.55	0.02	DMN_score	2.03	0.70
SAME	1.91	0.02	Chrun_en.	1.79	0.88
SIZE	1.84	0.02	Cyclomatic	1.76	0.87
LD	1.56	0.02	LA	1.67	0.82

Factor	qii_weiciyuan.		Factor	WSDOT.	
	Inc.	Effect		Inc.	Effect
DMN_sc.	2.62	0.74	DMN_sc.	0.92	0.04
Cyclomatic	5.79	0.06	NBM_sc.	1.33	0.00
NBM_sc.	1.66	0.01	Fuzzy_sc	0.82	0.00
Fuzzy_sc.	1.59	0.01	Cyclomatic	0.60	0.00
Chrun_en.	1.14	0.01	Comp_sc.	0.50	0.00

RQ2: What factors are the best indicators of a crashing mobile release? What is the relationship of these factors with a crashing mobile release?

Motivation: In addition to identifying crashing releases with high accuracy, we are interested in knowing what factors are good indicators of crashing releases and the relationship of the different factors with crashing releases. Knowing which and by how much each factor relates to a crashing release helps practitioners determine what factors they should carefully consider when determining which releases have a higher chance of being a crashing release.

Approach: To determine the important factors in determining a crashing release, we use IncNodePurity scores in the output of the R random forest library [11]. IncNodePurity refers to the mean decrease in node impurity, i.e., a higher IncNodePurity score means that the corresponding factor plays an more important role in the built model. Since our goal here is understanding and not prediction, we use the entire dataset to build the random forest models since we only need to use IncNodePurity scores outputted by random forest.

In addition, similar to prior work [37, 51], we measure the relationship (i.e., effect) of each factor with a crashing release. To do so, we create a model where factors are set to their median values, we predict the likelihood score for the release to be a crashing re-

lease, and denote these likelihoods as $base$. Then, for each factor f, we create a model where we double the median of the factor f, while keeping all other factors at their median values. We compute the likelihood score for the release to be a crashing release, and denote the score as f_{crash}. We measure the effect of the factor f by using $eff(f) = \frac{(f_{crash} - base)}{base}$. The effect of a factor can be positive or negative. A positive effect indicates that a higher level of a factor corresponds to an increase in the likelihood of a release being a crashing release, while a negative effect indicates that a higher level of a factor corresponds a decrease in the likelihood of a release being a crashing release.

Results: Table 4 presents top-5 factors which achieve the highest IncNodePurity scores of factors as assigned by the random forest, and the effect scores. We notice for different datasets, the most important factors are different. For example, the top-2 important factors are fuzzy_score and Cyclomatic in ankidroid_Anki-Android, and the top-2 important factors are NBM_score and fuzzy_score in mariotaku_twidere.

As for the relationship of the different factors with the prediction of a crash release, we find that all of the 20 factors have the non-negative effect with the crash releases. For BrandroidTools_OpenExplorer and guardianproject_Gibberbot, we notice the effect scores are much higher than the others, which means that in these apps a higher level of a factor corresponds to an increase in the likelihood of a release being a crashing release. For freezy_android-xbmcremote, mariotaku_twidere, and WSDOT_wsdot-android-app, the effect scores are much lower than the others, which means the impact of these factors to identify crash releases is low.

Considering both the importance (i.e., IncNodePurity) and effect of the factors, we find the factors in the text dimension such as Fuzzy_score, NB_score, NBM_score, DMN_score, and Comp_score show higher IncNodePurity and effect score compared to other factors. For example, DMN_score shows the highest importance and effect scores for freezy_android-xbmcremote and qii_weiciyuan.

To investigate why factors in the text dimension are more important and effective than the other factors, we also manually analyze the text in the commit logs. For example, in ankidroid_Anki-Android, the Fuzzy_score is one of the most important factors. For each term, we get its Fuzzy_score, and we rank these scores from high to low. We find the top-5 terms with the highest Fuzzy_scores are "screen", "loading", "UI", "button", and "view", and these terms are all related to problems in UI.

Similarly, in BrandroidTools_OpenExplorer, the NBM_score is one of the most important factors. We also rank the terms according to their NBM_scores, and we find the top-5 terms with the highest NBM_scores are "dialog", "translation", "icon", "page", and "box". The terms "dialog", "icon", "page", and "box" are related to UI issues, and the term "translations" is related to the language translation. We further investigate the patches for the crash releases, we find in BrandroidTools_OpenExplorer, a number of crash releases are caused by the enhancement or modification of UI, and also a number of crash releases are caused due to language translation (such as translate the language to French, Japanese, or Korean).

For the other apps, we also check the factors in the text dimension, and we find that a number of crashes are caused by UI enhancement or modification. In practise, users cannot bear the crashes in UI, and our finding is in consistent with the practical user experience.

Notice that in our paper, to ensure our prediction model is not biased by the large number of textual features, we only use 5 numeric textual factors (we initially start with 20 factors) by using 5 different text mining techniques. Moreover, besides the text factors, other factors such as Cyclomatic and LA are also important factors to predict crashing releases. For example, Cyclomatic and LA appear in the top-5 most important factors for 6 out of 10 apps.

Table 5: Prediction models by using different classifiers. NB=Naive Bayes, RF=Random Forest, C4.5=Decision Tree. The best precision, recall, F1 and AUC scores are in bold.

Algo.	ankidroid				bpellin.			
	Prec.	Recall	F1	AUC	Prec.	Recall	F1	AUC
NB	0.18	**0.52**	**0.26**	0.50	**0.18**	**0.61**	**0.28**	**0.61**
kNN	0.17	0.51	0.26	0.50	0.16	0.53	0.25	0.50
RF	**0.22**	0.28	0.24	**0.57**	**0.18**	0.30	0.23	0.49
C4.5	0.18	0.36	0.24	0.51	0.14	0.35	0.20	0.45

Algo.	Brandroid.				freezy.			
	Prec.	Recall	F1	AUC	Prec.	Recall	F1	AUC
NB	0.22	**0.79**	0.34	**0.70**	**0.21**	**0.58**	**0.31**	**0.70**
kNN	0.19	0.57	0.29	0.51	0.07	0.53	0.12	0.62
RF	0.21	0.29	0.24	0.61	0.11	0.11	0.11	0.57
C4.5	**0.26**	0.50	**0.35**	0.62	0.05	0.16	0.08	0.51

Algo.	gothfox.				mariotaku.			
	Prec.	Recall	F1	AUC	Prec.	Recall	F1	AUC
NB	0.15	0.56	0.24	0.60	0.20	0.50	0.29	0.63
kNN	0.16	**0.69**	0.26	0.56	0.18	**0.59**	0.27	0.63
RF	**0.28**	0.36	**0.32**	**0.65**	**0.33**	0.44	**0.38**	**0.69**
C4.5	0.22	0.39	0.28	0.59	0.21	0.41	0.28	0.62

Algo.	mtotschnig.				qii_wei.			
	Prec.	Recall	F1	AUC	Prec.	Recall	F1	AUC
NB	0.17	**0.68**	0.27	0.59	**0.41**	0.67	**0.51**	**0.78**
kNN	0.17	0.57	0.26	0.62	0.29	**0.69**	0.41	0.70
RF	0.18	0.21	0.19	0.59	0.38	0.49	0.43	0.77
C4.5	**0.23**	0.47	**0.31**	**0.62**	0.38	0.59	0.46	0.67

Algo.	WSDOT.				Average.			
	Prec.	Recall	F1	AUC	Prec.	Recall	F1	AUC
NB	0.19	0.63	0.29	0.78	0.20	**0.62**	**0.30**	**0.64**
kNN	**0.23**	**0.75**	**0.35**	**0.85**	0.17	0.60	0.26	0.60
RF	0.30	0.38	0.33	0.82	**0.23**	0.30	0.26	0.63
C4.5	0.25	0.38	0.30	0.65	0.21	0.39	0.27	0.58

6. DISCUSSION

Comparison Using Different Classifiers: Besides the Naive Bayes classifiers, there are other popular machine learning algorithms that can be used to predict whether a release is a crashing release or not. In this section, we compare the performance of three other classifiers namely: kNN, decision tree and random forest. Table 5 presents the precision, recall, F1, and AUC scores achieved by these classifiers. We notice different classifiers show different performance in different apps. For example, random forest shows the best F1 and AUC scores in gothfox_Tiny-Tiny-RSS-for-Honeycomb and mariotaku_twidere, and kNN shows the best F1 and AUC scores in WSDOT_wsdot-android-app. On average across the 10 apps, we find that Naive Bayes achieves the best performance. Thus, in practice, we recommend developers to use Naive Bayes when predicting crashing releases.

Longitudinal Data Setup: To investigate whether our tool can be used to solve the problem in the same setting as the one in practice, we performed an experiment using a longitudinal data setup (i.e., not using cross validation). We sorted the releases in the temporal order they are published. Then we built a prediction model by using the first 70% of the releases, and predict the labels for the remaining 30% of releases. Note that in WSDOT_wsdot-android-app, there are no crashing releases in the latest 30% of releases, thus the F1 and AUC scores are NA. Table 6 presents the precision, recall, F1, and AUC scores for Naive Bayes in longitudinal data setup. We see that Naive Bayes, in longitudinal data setup, achieves similar F1 and AUC scores as Naive Bayes in cross-validation setting. The average F1 and AUC scores for Naive Bayes in longitudinal data setup are 0.28 and 0.62, while these scores for Naive Bayes in cross-validation are 0.30 and 0.64.

Table 6: Precision, Recall, F1, and AUC scores for Naive Bayes in longitudinal data setup.

Project	Precision	Recall	F1	AUC
ankidroid_Anki-Android	0.11	0.22	0.15	0.50
bpellin_keepassdroid	0.31	0.79	0.45	0.56
BrandroidTools_OpenExplorer	0.23	1.00	0.37	0.81
freezy_android-xbmcremote	0.06	0.88	0.12	0.74
gothfox_Tiny-Tiny-RSS-for-Honeycomb	0.15	0.50	0.23	0.57
guardianproject_Gibberbot	0.18	0.58	0.28	0.59
mariotaku_twidere	0.28	0.65	0.39	0.62
mtotschnig_MyExpenses	0.22	0.53	0.31	0.55
qii_weiciyuan	0.33	0.15	0.21	0.60
Average.	**0.21**	**0.59**	**0.28**	**0.62**

Precision vs. Recall: The goal of our approach is to flag crashing releases to improve the quality of mobile apps. As with any classification technique, there is a tradeoff between precision and recall. As we have shown, our approach easily outperforms the baseline approaches in terms of F1-measure and AUC, however, it is important to note that we tend to achieve better recall than precision. The reason for favouring recall over precision is due to the fact that crashing releases are very negatively perceived by users, hence, it is important that we avoid crashing releases at any cost. That said, it is important to note that one can easily modify the threshold used in our technique (currently we set the threshold to be 0.5), to achieve a higher precision at the cost of lower recall. Depending on individual/project-specific circumstances, users of our tool can adjust the threshold based on how much recall or precision matter to them.

Usefulness of Our Proposed Tool: After we completed the study, we also sent a version of this paper to mobile app developers in a company, called Hengtian, to inquire about the usefulness of our approach and the factors used to perform our prediction. In total, we sent the paper to 20 developers, and we received 12 responses (denoted as R1 to R12). The majority of the developers (11 out of the 12) consider the proposed factors and machine learning techniques to be practical and applicable in a real mobile development environment. For example, R2 states: *"we can easily extract these factors from the commits in a release, and also the naive Bayes classifier is understandable"*. However, R5 has some questions about how machine learning techniques could work in practice. Moreover, 10 out of the 12 developers agree that the results show that our proposed approach is useful. For example, R9 states that *"although F1-score is not high, the recall score is good, in practice we are interested to find all the crash releases as possible, thus recall is more important than precision. The proposed tool could enhance the confidence about the quality of a release"*. On the other hand, two developers (R5 and R7) wish the F1-score would be higher. Therefore, in the future we plan to continue investigating other factors that can help improve our prediction accuracy.

Threats to Validity: Threats to internal validity refers to errors in our code and experiment bias. We have double checked our code, however, there may exist some errors that we did not notice. Also, to identify the crash releases, we looked for the keywords "crash", "crashed", "crashing", and "crashes". In such a way, we may increase the number of false positives and false negatives. To reduce the number of false positives, for each identified crashing release, we manually checked the commit logs and modified source code. To reduce the number of false negatives, for these non-crashing releases, we also randomly choose and manually check 30% of the releases from these non-crashing releases, and we found all of these releases labeled as non-crashing releases were correctly labeled.

Set and selection bias is another threat to internal validity. To mitigate this bias, we run the 10-fold cross-validation 100 times, and present the average performance. To identify crashing releases, we manually check the commit logs, bug reports, code change, and user reviews. Due to the manual process, some releases may be wrongly identified as non-crashing/crashing releases.

Threats to external validity relates to the generalizability of our results. We have analyzed 2,638 releases from 10 mobile applications. In the future, we plan to reduce this threat further by analyzing even more releases from additional mobile applications.

Threats to construct validity refers to the suitability of our evaluation measures. We use F1 and AUC scores which is also used by past studies to evaluate the effectiveness of various software engineering studies [19, 24, 27, 28, 43, 48, 55, 56, 62].

7. RELATED WORK

Prediction in Software Engineering: There have been a number of studies that use prediction in software engineering (SE). Kim et al. proposed the change classification problem, which predicts whether a change is buggy or clean [24]. Jiang et al. proposed PCC, which built a separate prediction model for each developer to predict software defects [19]. Bhattacharya et al. proposed a set of graph-based metrics to predict the number of defects in a release [3]. Dhaliwal et al. [7] used crash report data to assist in the bug fixing of Mozilla Firefox bugs. Francese et al. use requirements measures to predict software project and product measures in the context of Android mobile apps [12]. Yu and Yeung use install and uninstall log in a mobile app store to estimate the value of the apps [64]. Finkelstein et al. extract a set of features from release notes available in app store, and they find that there is strong correlation between customer rating and the rank of apps in terms of the number of downloads [10]. Ferrucci et al. perform a replicated study on the effectiveness of functional and code size measures to apps [9]. Similar to the prior work, we also perform prediction in SE context. However, our work focuses on a related but different problem: different from prior defect prediction work, we evaluate the quality of a mobile application at the release level. In particular, we predict which mobile releases have a high likelihood of being a crashing release. To the best of our knowledge, our work is the first to predict crashing mobile app releases.

Mobile Applications: There have been a number of studies in the area of mobile apps. Martin et al. perform a comprehensive survey on app store analysis for software engineering [33]. Minelli and Lanza studied the difference between the development of mobile application and traditional software systems [36]. They develop a web-based software analytics platform named SAMOA. Harman et al. found that there are strong correlations between the rating of mobile apps and the ranking of app downloads by mining the reviews from App stores [16]. Joorbachi et al. investigated the challenges in mobile application development [20]. They found that dealing with multiple mobile platforms is one of the most challenging aspects of mobile development. Martin et al. find that the release context plays a role in whether a release is impactful and the type of impact it has [31, 32]. Nayebi et al. perform two surveys to understand the common release strategies used for mobile apps, the rationale behind them and their perceived impact on users [44]. Martin et al. propose the app sampling problem, and study its effects on sets of user review data [30]. Guerrouj et al. study the impact of app churn on the app rating by analyzing 154 free apps, and they find that high app churn leads to lower user ratings [13]. Most related is our prior work, where we performed an empirical study on the reasons behind low ratings and complaints that were posted on mobile app reviews [22, 23]. We found that a high percentage of complaints and bad reviews were posted after new releases. The work presented in this paper is motivated by the findings of our prior findings, however, our study is orthogonal since our goal is to predict crashing mobile app releases.

8. CONCLUSION AND FUTURE WORK

In this paper, we perform a study on the prediction of crashing releases in mobile applications. We first collect 2,638 releases from 10 mobile applications in F-Droid. Then, we extract various factors from the collected releases. Based on the factors, we propose the usage of a Naive Bayes model to predict crashing releases. To investigate the performance of our proposed approach, we perform experiments on the 2,638 releases. On average across the 10 apps, our approach can achieve F1 and AUC scores to 0.30 and 0.64, which improves the random prediction by 50% and 28%, and improves the AUC scores of majority prediction by 30%, respectively.

Our work is one of the first works focusing on the prediction of crashing releases. Although the performance of our approach is not perfect, we hope that our work will inspire other researchers to develop more advanced techniques to identify crashing releases, since it is clearly a problem that negatively impacts both, mobile users and developers. In the future, we plan to explore more factors and evaluate our approach with datasets from more mobile applications, develop a better technique that can further improve the prediction performance, and evaluate the performance of our proposed tool by deploying it in practice. We also plan to investigate how the integration of bug localization techniques into our technique can help developers in the fixing process of crashing releases. Moreover, we plan to perform an empirical study on developers' perception on crash release prediction. Recently, Bowes et al. investigated the usefulness of mutation metrics for fault prediction [4]. In the future, we plan to investigate if these metrics are useful to predict crashing releases.

Acknowledgment. This research was partially supported by JSPS KAKENHI Grant Number 15H05306, the NSFC Program (No.61572426), and National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2015BAH17F01.

9. REFERENCES

[1] K. Allix, Q. Jerome, T. F. Bissyande, J. Klein, R. State, and Y. le Traon. A forensic analysis of android malware–how is malware written and how it could be detected? In *COMPSAC*, 2014.

[2] Android, Apple, Google, Microsoft, AppBrain, BlackBerry, V. sources (WindowsCentral.com), and A. (n.d.). Number of apps available in leading app stores as of july 2015.

[3] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *ICSE*, 2012.

[4] D. Bowes, T. Hall, M. Harman, Y. Jia, F. Sarro, and F. Wu. Mutation-aware fault prediction. In *ISSTA 2016*.

[5] L. Breiman. Random forests. *Machine learning*, 2001.

[6] N. Cliff. *Ordinal methods for behavioral data analysis*. 2014.

[7] T. Dhaliwal, F. Khomh, and Y. Zou. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In *ICSM 2011*.

[8] A. Estabrooks and N. Japkowicz. A mixture-of-experts framework for learning from imbalanced data sets. In *Advances in Intelligent Data Analysis*. 2001.

[9] F. Ferrucci, C. Gravino, P. Salza, and F. Sarro. Investigating functional and code size measures for mobile applications: A replicated study. In *Product-Focused Software Process Improvement*.

[10] A. Finkelstein, M. Harman, Y. Jia, F. Sarro, and Y. Zhang. Mining app stores: Extracting technical, business and customer rating information for analysis and prediction. *RN*, 13:21, 2013.

[11] A. S. Foulkes. *Applied statistical genetics with R*. 2009.

[12] R. Francese, C. Gravino, M. Risi, G. Scanniello, and G. Tortora. On the use of requirements measures to predict software project and product measures in the context of android mobile apps: a preliminary study. In *EUROMICRO-SEAA 2015*.

[13] L. Guerrouj, S. Azad, and P. C. Rigby. The influence of app churn on app success and stackoverflow discussions. In *SANER 2015*.

[14] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 2009.

[15] J. Han and M. Kamber. *Data Mining, Southeast Asia Edition: Concepts and Techniques*. 2006.

[16] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: Msr for app stores. In *MSR*, 2012.

[17] A. E. Hassan. Predicting faults using the complexity of code changes. In *ICSE 2009*.

[18] N. Japkowicz and S. Stephen. The class imbalance problem: A systematic study. *Intelligent data analysis*, 2002.

[19] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *ASE 2013*.

[20] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *ESEM*, 2013.

[21] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *TSE*, 2013.

[22] H. Khalid. On identifying user complaints of ios apps. In *ICSE*, 2013.

[23] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile app users complain about? a study on free ios apps. *IEEE Software*, 2015.

[24] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *TSE*.

[25] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *ICSE*, 2007.

[26] D. E. Krutz, M. Mirakhorli, S. A. Malachowsky, A. Ruiz, J. Peterson, A. Filipski, and J. Smith. A dataset of open-source android applications. In *MSR*.

[27] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *MSR*, 2010.

[28] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *TSE*.

[29] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *FSE*, 2013.

[30] W. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang. The app sampling problem for app store mining. In *MSR 2015*.

[31] W. Martin, F. Sarro, and M. Harman. Causal impact analysis for app releases in google play. In *FSE 2016*.

[32] W. Martin, F. Sarro, and M. Harman. Causal impact analysis applied to app releases in google play and windows phone store. *RN*, 15:07, 2015.

[33] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A survey of app store analysis for software engineering. *RN*, 16:02, 2016.

[34] A. McCallum, K. Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop*.

[35] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 2010.

[36] R. Minelli and M. Lanza. Software analytics for mobile applications–insights & lessons learned. In *CSMR*, 2013.

[37] A. Mockus. Organizational volatility and its effects on software defects. In *FSE*, 2010.

[38] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 2000.

[39] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE*, 2008.

[40] H. Muccini, A. Di Francesco, and P. Esposito. Software testing of mobile applications: Challenges and future research directions. In *AST*, 2012.

[41] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE*, 2005.

[42] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE*, 2006.

[43] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *ICSE 2013*.

[44] M. Nayebi, B. Adams, and G. Ruhe. Release practices in mobile apps? users and developers perception. In *SANER 2016*.

[45] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: hit or miss? In *ESEC/FSE*, 2011.

[46] J. D. Rennie, L. Shih, J. Teevan, D. R. Karger, et al. Tackling the poor assumptions of naive bayes text classifiers. In *ICML*, 2003.

[47] B. Robinson and P. Francis. Improving industrial adoption of software engineering research: a comparison of open and closed source software. In *ESEM 2010*, ESEM '10.

[48] D. Romano and M. Pinzger. Using source code metrics to predict change-prone java interfaces. In *ICSM*, 2011.

[49] C. Rosen, B. Grawi, and E. Shihab. Commit guru: Analytics and risk prediction of software commits. In *ESEC/FSE 2015*.

[50] C. Rosen and E. Shihab. What are mobile developers asking about? a large scale study using stack overflow. *EMSE*.

[51] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *FSE*, 2012.

[52] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 2005.

[53] J. Su, H. Zhang, C. X. Ling, and S. Matwin. Discriminative parameter learning for bayesian networks. In *ICML*, 2008.

[54] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *TSE*, 2003.

[55] F. Thung, D. Lo, M. H. Osman, and M. R. Chaudron. Condensing class diagrams by analyzing design and network metrics using optimistic classification. In *ICPC*, 2014.

[56] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *ICSE*, 2012.

[57] Y. Tian, D. Lo, and C. Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *ICSM*, 2013.

[58] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *EMSE*, 2009.

[59] H. Valdivia and E. Shihab. Characterizing and predicting blocking bugs in open source projects. In *MSR*, 2014.

[60] H. Valdivia Garcia and E. Shihab. Characterizing and predicting blocking bugs in open source projects. In *MSR*, 2014.

[61] F. Wilcoxon. Some rapid approximate statistical procedures. *Annals of the New York Academy of Sciences*, 52(6):808–814, 1950.

[62] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *ESEC/FSE*, 2011.

[63] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang. Elblocker: Predicting blocking bugs with ensemble imbalance learning. *Information and Software Technology*, 61:93–106, 2015.

[64] P. Yu and C.-m. Au Yeung. App mining: finding the real value of mobile applications. In *WWW*, 2014.

[65] H. Zimmermann. *Fuzzy Set Theory and Its Applications Second, Revised Edition*. 1992.