Exploring the Integration of User Feedback in Automated Testing of Android Applications

Grano, Giovanni ; Ciurumelea, Adelina ; Panichella, Sebastiano ; Palomba, Fabio ; Gall, Harald C

Abstract: The intense competition characterizing mobile application's marketplaces forces developers to create and maintain high-quality mobile apps in order to ensure their commercial success and acquire new users. This motivated the research community to propose solutions that automate the testing process of mobile apps. However, the main problem of current testing tools is that they generate redundant and random inputs that are insufficient to properly simulate the human behavior, thus leaving feature and crash bugs undetected until they are encountered by users. To cope with this problem, we conjecture that information available in user reviews—that previous work showed as effective for maintenance and evolution problems—can be successfully exploited to identify the main issues users experience while using mobile applications, e.g., GUI problems and crashes. In this paper we provide initial insights into this direction, investigating (i) what type of user feedback can be actually exploited for testing purposes, (ii) how complementary user feedback and automated testing tools are, when detecting crash bugs or errors and (iii) whether an automated system able to monitor crash-related information reported in user feedback is sufficiently accurate. Results of our study, involving 11,296 reviews of 8 mobile applications, show that user feedback can be exploited to provide contextual details about errors or exceptions detected by automated testing tools. Moreover, they also help detecting bugs that would remain uncovered when rely on testing tools only. Finally, the accuracy of the proposed automated monitoring system demonstrates the feasibility of our vision, i.e., integrate user feedback into testing process.

DOI: https://doi.org/10.1109/SANER.2018.8330198

Exploring the Integration of User Feedback in Automated Testing of Android Applications

Giovanni Grano, Adelina Ciurumelea, Sebastiano Panichella, Fabio Palomba, Harald C. Gall
University of Zurich,
Department of Informatics, Switzerland
{grano,ciurumelea,panichella,palomba,gall}@ifi.uzh.ch

***Abstract*—The intense competition characterizing mobile application's marketplaces forces developers to create and maintain high-quality mobile apps in order to ensure their commercial success and acquire new users. This motivated the research community to propose solutions that automate the testing process of mobile apps. However, the main problem of current testing tools is that they generate redundant and random inputs that are insufficient to properly simulate the human behavior, thus leaving feature and crash bugs undetected until they are encountered by users. To cope with this problem, we conjecture that information available in user reviews—that previous work showed as effective for maintenance and evolution problems—can be successfully exploited to identify the main issues users experience while using mobile applications, e.g., GUI problems and crashes. In this paper we provide initial insights into this direction, investigating (i) what type of user feedback can be actually exploited for testing purposes, (ii) how complementary user feedback and automated testing tools are, when detecting crash bugs or errors and (iii) whether an automated system able to monitor crash-related information reported in user feedback is sufficiently accurate. Results of our study, involving 11,296 reviews of 8 mobile applications, show that user feedback can be exploited to provide contextual details about errors or exceptions detected by automated testing tools. Moreover, they also help detecting bugs that would remain uncovered when rely on testing tools only. Finally, the accuracy of the proposed automated monitoring system demonstrates the feasibility of our vision, i.e., integrate user feedback into testing process.**

***Index Terms*—Automated Software Testing, Mobile Applications, User Reviews Analysis**

I. Introduction

Mobile devices, such as smartphones and tablets, acquired more and more a central role in everyday life in recent years [31]. Consequently, we witnessed an unprecedented growth of the app industry, with around 149 billions of mobile apps downloaded by September 2016 [47] and 12 million of developers maintaining them [36]. The growing competition characterizing mobile application marketplaces, like Google Play and the Apple App Store, ensures that only high quality apps stay on the market and gain users. This forces developers to deliver high quality apps, maintaining them through adequate software testing activities [27], [31]. However, mobile applications differ from traditional software, being structured around Graphical User Interface (GUI) events and activities [1], [34]. Therefore, they expose different kinds of bugs and suffer of a higher defect density compared to traditional desktop and server applications [21]. To support developers in building high-quality applications, the research community has recently developed novel techniques and tools to automate such a testing process [21], [27], [29], [34]. They generate, according to different strategies, UI and system events, like the tap on a button or an incoming notification. Such events are then transmitted to the application under test (AUT) with the aim of detecting *unhandled runtime exceptions*. If an exception occurs, such techniques typically save two different information: (i) the correspondent *stack trace* and (ii) the sequence of events that led to the crash [34]. Unfortunately, most of these tools suffer of three important limitations. Firstly, the reports they generate (composed as mentioned, by a stack trace and a sequence of inputs) lack of contextual information and are difficult to understand and analyze [10], [23]. Secondly, they are able to detect only bugs that actually cause unhandled exceptions, thus possibly missing those not raising any. Third and most important limitation, current tools *"are not suited for generating inputs that require human intelligence (*e.g.*, constructing valid passwords, playing and winning a game, etc.)"* [27]. Indeed, the generation of such random inputs often results in a redundant sequences of events that are not able to simulate the human behavior. For this reason, such tools (i) are often not able to achieve high code coverage [12], [35] and (ii) might fail to detect bugs and crashes that are encountered by users [11], [27].

In this context, recent work demonstrate that it is possible to leverage information available in app reviews to identify the main problems encountered by users while using an app [13], [38], [41], [45], [48]. We believe that such information can be successfully exploited to overcome some of the limitations of state-of-the-art tools for automated testing of mobile apps. Specifically, we argue that the integration of user feedback into the testing process can (i) *complement* the capabilities of automated testing tools, by identifying bugs that they cannot reveal or (ii) *facilitate* the diagnosis of bugs or crashes (since users might describe the actions that led to a crash).

To better explain the motivations behind our work, we provide below two concrete examples taken from the dataset gathered for this study (detailed in Section II). In the first example, we report the content of a user review, related to the `com.danvelazco.fbwrapper` app, revealing a bug missed by widely adopted automated testing tools such as Monkey [17] and Sapienz [29]:

"Love the idea of this app but anytime I leave the page the screen

goes completely white and won't come back until force-stopped. Update: I thought the white screen was because my phone was so outdated but it still does it on my Nexus 6 ...".

In this case, the user complains for a white screen displayed after leaving the previous page in the application. However, such white screen did not throw any unhandled exception. As a consequence, *all* the current Android automated testing tools could not detect at all such a problem.
The second example shows the opposite situation: in this case, SAPIENZ detects the following exception for the `com.amaze.filemanager` app.

```
Long Msg: java.lang.NumberFormatException: Invalid int: "/"
java.lang.RuntimeException: An error occurred while executing
     doInBackground()
  at android.os.AsyncTask$3.done(AsyncTask.java:300)
  at java.util.concurrent.FutureTask.finishCompletion(FutureTask.java
     :355)
    ...
  at com.amaze.filemanager.services.asynctasks.LoadList.
     doInBackground(LoadList.java:120)
  at com.amaze.filemanager.services.asynctasks.LoadList.
     doInBackground(LoadList.java:50)
  at android.os.AsyncTask$2.call(AsyncTask.java:288)
  at java.util.concurrent.FutureTask.run(FutureTask.java:237)
  ... 3 more
```

While this piece of information correctly reveals an actual error occurring in the app, locating the exact origin of the failure might be hard for developers since it lacks of contextual information, *i.e.*, they cannot understand *what is* the event that induces the failure. Thus, we believe that having a monitoring system that provides developers with information about the bugs encountered by users might be useful for *comprehending* the causes behind a failure, *easing* the debugging phase, and *discovering* errors that existing testing tools cannot reveal.

Paper contribution. In this paper we conducted a set of methodological steps aimed at (i) characterizing the *types* of user review feedback that can be exploited for testing purposes; (ii) investigating the extent to which information coming from user reviews are *complementary* with respect to crashes discovered by testing tools. Furthermore, we define an initial *automated monitoring system* able to link user reviews to stack traces. Our aim is dual: linked feedback might describe the cause of a failure, helping in its resolution; on the contrary, developers would deeper focus on the not-linked ones, which are likely to describe bugs missed by tools. Thus, the contributions introduced by this paper are:

- the definition of a *high* and a *low* level taxonomy of user review feedback. The former allows to classify reviews relevant from a maintenance and testing perspective. The latter focuses only on the reviews claiming about crashes, discerning them from the user's perspective;
- an automated Machine Learning (ML) approach able to discriminate the user feedback exploitable for testing purposes;
- an empirical study on the complementarity between testing tools and user review information in identifying crash bugs in mobile apps. In addition, we shed some light on the root causes of such problems, specifying which ones are mostly detected rely only on one of the two analyzed sources;
- an automated approach based on Information Retrieval (IR) able to link stack traces to user feedback that refers to the same failure; the combination of such technique with the automated ML approach described above, alleviates one of the discussed problems in Android automated testing, *i.e.*, the inability of testing tools to reveal some kind of bugs. Indeed, the remaining non-linked crash-related reviews are the ones that developers might want to analyze, since they are likely revealing errors not retrieved by testing tools.

To give an intuition of the potential of the approach we propose, we report below an user review automatically linked to the stack trace we previously showed.

"Every time I press the the recent apps button the app crashes.".

There is no doubt about how such an user's perspective description, together with the occurred exception, drastically facilitate the diagnosis of the problem and therefore, its eventual fix.

Structure of the paper. Section II presents the empirical study, data collection process, our research questions and the approaches we use to answer them. Section III describes the achieved results, while Section IV discusses the main threats. Related work are presented in Section V while Section VI concludes the paper drawing the envisioned future work.

II. EMPIRICAL STUDY DESIGN

The goal of the study is to (i) assess the extent to which the information coming from the user reviews posted in app stores can be exploited to facilitate testing activities of mobile applications, (ii) analyze whether and to what extent such information is complementary with the one coming from automated testing tools and (iii) proposing an automated solution to provide developers with a monitoring system able to support the integration of user feedback in the testing process. The *perspective* is both of researchers interested in understanding how user feedback can be leveraged for testing activities and mobile developers, who might want to identify bugs in their applications relying on multiple and complementarily sources of information. Thus, this empirical study aims at answering the following research questions:

- **RQ**$_1$: *What type of user feedback can we leverage to detect bugs and support testing activities of mobile apps?*
- **RQ**$_2$: *How complementary is user feedback information with respect to the outcomes of automated testing tools?*
- **RQ**$_3$: *To what extent can we automatically link the crash-related information reported in both user feedback and testing tools?*

RQ$_1$ investigates which type of user feedback can be fruitfully integrated into the testing activities of mobile apps. **RQ**$_2$ aims at (i) investigating the complementary of user feedback information with respect to the outcomes of automated testing

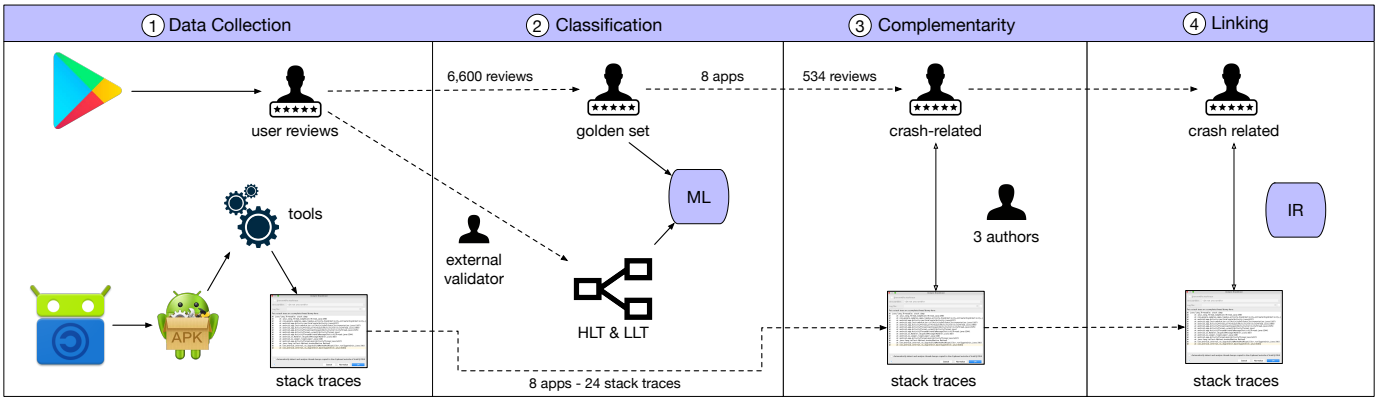

Fig. 1. Overall Approach

tools and (ii) assessing whether user feedback can reveal bugs that state-of-the-art automated testing tools cannot detect. Finally, with $\mathbf{RQ}_3$ we explore possible automated solutions able to link the failures revealed by automated tools with the ones claimed in user reviews to help developers discern the app bugs that can be detected relying on such sources of information. The research approach we adopted to answer our research questions is depicted in Figure 1: ① depicts the data collection step; the process used to answer our $\mathbf{RQ}_1$ corresponds to ②; ③ visualizes the manual analysis needed for $\mathbf{RQ}_2$, while ④ refers to the proposed linking approach.

A. Context of the Study

The context of the study is composed of the set of apps available in the F-Droid repository [15] and by the correspondent user reviews gathered from the Google Play Store. To answer our first research question we relied (as explained in Section II-C1) on such an entire dataset of apps and reviews. For the other two one, we narrowed the investigation to a subset of 8 Android applications, showed in Table I. The selection of such a subset was guided by two major considerations. At first, we selected only mobile apps for which it was possible to reveal at least 1 unhandled exception with the employed automated testing tools. Moreover, amongst them, we picked the ones having the highest number of collected reviews. To limit the threats to external validity, we also selected them considering different Play Store's categories, sizes and domains.

B. Data Collection

This section reports the steps (① in Figure 1) conducted to extract user reviews and stack traces to answer our research questions. Such a dataset is publicly available in our *replication package* [4].

User Reviews Extraction. To answer $\mathbf{RQ}_1$ and determine which kind of feedback can be exploited to support automated testing, we first build a crawler to collect the meta-data (*e.g.*, package name, version and so on) and the latest apk available for each app stored in the F-Droid repository. Afterwards, we mined from the Google Play Store the correspondent reviews, *i.e.*, the ones posted after the release of our gathered versions [18]. We are aware that such set of reviews might contain user comments that do not refer to the last version of the app. However, Pagano and Maalej empirically demonstrated that user feedback is mostly triggered by new releases [37]. Thus, at the end of this phase, we collected in total a set of about 40,000 reviews for about 600 applications that we later used to construct our *golden set* (as detailed in Section II-C1) relevant to answer our $\mathbf{RQ}_1$. It is worth to underline that, from the original F-Droid set of apps, we discarded the ones with less than 30 user reviews available.

Stack Trace Collection. To extract the stack traces for the crashes occurring in the analyzed apps, we employed two well-known tools for Android automated testing, namely MONKEY [17] and SAPIENZ [29]. The choice of using these tools was given by multiple factors. In the first place, MONKEY, coming directly integrated into the Android development environment, is the most used tool and therefore, a *standard-de-facto* for automated testing of mobile apps. Moreover, despite employing a relatively trivial random exploration strategy, it outperforms most of the recent tools in *continuous mode* (*i.e.*, fixed execution time and same hardware configuration) [29]. On the other hand, the selection of SAPIENZ was driven by recent findings reporting that it is the most effective tool in both fault revelation and code coverage [29]. These tools were ran over the applications described in Table I: as done in previous work, we ran them for 60 minutes [12], using a Samsung Galaxy Tab 8 inches with Android Kitkat 4.4. The choice of the Android version is due to SAPIENZ constraints. It is worth to remark that we refer to a crash for an application as an unhandled exceptions that results in a stack trace. At the end, we merged together the arose crashes from both tools. Thus, we manually analyzed them in order to (i) discard stack traces due to native crashes (*i.e.*, the ones not caused by the AUT) and (ii) remove possible duplicates.

C. Research Method

In this section, for each research question, we describe the research approach we employed to answer it.

1) ***$\mathbf{RQ}_1$ research method:*** The primarily goal of this research question is to identify and therefore be able to automatically classify the feedback in user reviews that can fruitfully employed to facilitate testing activities. To pursue such a goal,

TABLE I
SUBSET OF APPS SELECTED FOR THE STUDY

Application	Category	Crashes	Reviews	
			Total	Crash
com.amaze.filemanager	Tools	7	1,438	28
com.danvelazco.fbwrapper	Social	4	1,900	252
com.eleybourn.bookcatalogue	Productivity	1	677	11
com.evancharlton.mileage	Finance	2	1,064	39
com.fsck.k9	Communication	1	2,895	106
com.ringdroid	Video Pl. & Editor	4	2,363	84
cri.sanity	Communication	1	695	11
org.liberty.android.fantastischmemo	Education	4	264	3
Total		**24**	**11,296**	**534**

we built a multi-level taxonomy composed by (i) an *high-level* layer that categorize the reviews that are relevant for a maintenance and an evolution perspective, and (ii) a *low-level* layer, that further specialize the previous one, focusing on the user feedback that can be directly used to improve and facilitate testing activities.

High-level Taxonomy (HLT). The fist step required to build our taxonomy was to derive a manual labeled *golden set* (step ② in Figure 1). Therefore, using a stratified approach, we randomly selected from the entire set of reviews, collected as described in Section II-B, the 15% from each app, resulting in a set of over 6,600 reviews. To derive the HLT we proceeded as follows. At first, a Master student at University of Zurich performed an iterative content analysis [24], starting with an empty list of user feedback categories and carefully analyzing each review. Each time she found a new review feedback type, a new category was added to the list and each review was labeled with the matching categories. After the first step, the initial categorization was refined performing another interaction involving one of the authors of this paper who double-checked each review and removed potential redundant categories in the taxonomy. The resulting HLT is described in Section III-A. The set of 6,600 reviews manually validated according to the HLT represents our *golden set*.

Once defined such taxonomy, we adopted an automated approach, similar to the one presented by Ciurumelea *et al.* [13], to classify reviews according to the defined HLT. The main differences with this work is that we (i) used a larger dataset of reviews to build the taxonomy and (ii) focused on a reduced set of categories tailored for maintenance and evolution tasks. Thus, we leveraged a supervised machine learning (ML) technique by performing the following steps:

- *Preprocessing and Feature Extraction:* First, the review text is preprocessed by applying lower case reduction, English stop words removal and reducing words to their root form using stemming [5]. As features, the *tf-idf* scores of the 1-grams, 2-grams and 3-grams of terms are computed for each review. Note that an n-gram is defined as a contiguous sequence of n items from a given sequence of text [49]. We prefer the *tf-idf* score [5], as opposed to simple frequency counts, because it assigns a higher value to rare words (or group of words) and a lower value to common ones, therefore identifying the important words in a review.
- *ML Models Training:* After experimenting with different models we decided to use the Gradient Boosted Classifier algorithm, as this returned the best results. We used the implementation from the `scikit-learn` library [44], because it is open source and widely used in practice. The model achieved the best results while keeping the default values for the different hyper-parameters. We only modified the `n_estimators` argument, setting it to 500, which increased the precision and recall while slightly increasing the training time. We plan as future work to investigate whether the tuning of such hyper-parameters can further improve the accuracy of the classification.

To train and evaluate the classifiers, we used the aforementioned *golden set*, extracting the features previously described. The dataset was highly unbalanced in terms of user feedback categories, thus, training the classifiers on the entire dataset did not result in very high accuracy and recall for the minority categories. For this reason, we applied a known technique, called under-sampling, to tackle the data imbalance problem [50]. Specifically, from the initial dataset we selected all the reviews that were manually labeled as belonging to the relevant categories, and additionally we randomly selected a subset of the reviews that did not belong to either of the categories, finally obtaining a dataset of 2,565 reviews. Therefore, we used the classifier trained on such a balanced dataset and evaluated its performances using a 10-fold cross validation procedure [16]. Such results are presented in Section III-A.

Low-level Taxonomy (LLT). Differently from previous work, we aim to detect a very specific subset of user reviews that might facilitate developers' testing activities and complement the usage of automated testing tools. Being one of the focus of this work investigating the complementarity between the outcome of such tools and the user reviews, we fine-grained the HLT taxonomy with a new layer tailored to take into account only the feedback discussing crashes. That choice was driven by the fact that testing tools are not able to reveal semantic bugs, but only failures that result in unhandled exceptions. Hence, we performed a further iterative content analysis by manually categorizing a stratified sample set of 534 user reviews, selected from the projects reported in Table I, as described in Section II-C2. The result of such a process is a fine-grained taxonomy describing the various root causes that can lead to a crash, from the users perspective. We proceeded as follows. In a first iteration, three authors of the paper *independently* performed a first categorization of the sampled set of reviews, by reading their descriptions and trying to logically group them. The purpose of this task was to ensure that the three authors assigned each problem to the right category. As results, they deducted independently three different LLT. Observing the results of the initial iteration, we noticed that the low-level taxonomies were already very similar. The main difference between them concerned the adoption of different labels (or names) for the same types of

issue. Thus, in a second and final iteration, the three authors met and discussed all cases of disagreement, merging and/or renaming existing categories. That resulted in the final version of the LLT, reported in Section III-A.

*2) **RQ$_2$ research method**:* The goal of this research question is to shed some light on the possibility to leverage user feedback to *complement* the capabilities of current automated testing tools. Specifically, we aim to detect (i) crashes described by users in app reviews and that are detected as well by automated testing tools and (ii) crashes described by users that automated tools are not able to reveal. Hence, we strive to manually observe/explore possible differences and commonalities between the two sources. To achieve this goal, we manually analyzed both user reviews and stack traces generated by automated testing tools (③ in Figure 1). Being such analysis extremely time consuming, we narrowed the investigation to the 8 apps described in Table I, relying on the selection criteria described in Section II. The outcome of this manual analysis is both quantitative and qualitative. The quantitative part discusses the percentage and types of crash bugs detected (i) by users only and described in user reviews (ii) the one detected by only automated tools and (iii) the one detected by automated tools and reported in crash-related reviews. Instead, the qualitative analysis complements the quantitative side by giving insights about the reasons why specific app bugs are detected by both sources of information, while others are detectable by solely relying on one of them. To do that, we involved an external inspector having 2 years of experience in Android development. We gave her (i) the stack traces arose from the execution of the tools, (ii) the logs of the executed events that led to the crashes, (iii) the apk and the source code for the used apps, as well as the image of the same emulator used for the stack trace extraction, and (iv) the set of reviews related to the *Crashes* sub-category for our HLT. It is worth to notice that we relied on our ML classifier to discern such reviews. Table I reports, for each app, the number of stack traces and of the crash-related reviews provided. At first, using a MONKEY feature, she had to re-run the stored sequences of events for the collected crashes, to reproduce them and understand their dynamics. Whenever such analysis was not enough for a full comprehension, she had to inspect the source code trying to figure out its behavior in the proximity of the piece of code that threw the exception. At the end, she went through the set of provided reviews, linking them, whenever possible, to one of the stack traces. Despite the evident amount of work done by the external evaluation, we missed to precisely measure the hours spent for the aforementioned tasks. Once the evaluator completed the manual link process, we quantitatively answered this research question by computing the percentage of issues coming from the various sources of information, relying on the following metrics:

- I_C: % of issues reported in both reviews and crash logs;
- I_R: % of issues reported only in user reviews;
- I_S: % of issues reported only in crash logs.

To compute them, we first defined $\mathcal{T}$ as the total number of unique issues reported for a given app. Such value is formalized as follows:

$$\mathcal{T} = \mathcal{L} + \mathcal{S} + \mathcal{R}$$

where $\mathcal{L}$ represents the number of *links* detected by the evaluator between a review and a stack trace; $\mathcal{S}$ represents the number of *stack traces* that was not possible to link to any reviews and $\mathcal{R}$, on the contrary, the number of *reviews* that were not linked to any stack trace. Thus, we can formally describe the three overlap metrics introduced above as follow:

$$I_C = \frac{\mathcal{L}}{\mathcal{T}} \qquad I_R = \frac{\mathcal{R}}{\mathcal{T}} \qquad I_S = \frac{\mathcal{S}}{\mathcal{T}}$$

In order to have a fair computation of the three metrics I_C, I_R and I_S, presented above we proceeded to manually (i) detect duplicated stack traces and (ii) cluster the user reviews claiming about the same crash (avoiding to count twice or more the same bug). Moreover, we relied on the aforementioned LLT with the aim to deeper understand the nature of the user reviews that was not possible to link to any stack trace ($\mathcal{R}$). Since all the user reviews used for the **RQ$_2$** analysis were labelled according to the LLT, we observed the distribution of the not-linked reviews in the categories of such taxonomy. Results of the described quantitative and qualitative analysis are reported in Section III-B.

*3) **RQ$_3$ research method**:* The goal of this research question consists of exploring approaches able to link the failures revealed in the *stack traces* (contained in the reports) generated by automated tools to the corresponding crash-related reviews. Specifically, an automated approach of this kind is required as it will (i) enable the identification of crashes that are detected by both automated tools and user reviews and (ii) discriminating the crashes that are detected only by automated tools or only by user reviews information.

Linking the two sources of information is difficult because they are very different: user reviews contain informal text documentation which describe the overall scenario that led to a failure [32], while the *stack traces* contain technical information about the exceptions raised during the execution of a certain test case. To account for this aspect, the automated approach devised only considers the name and cause of the raised exceptions, while it removes the remaining pieces of information that creates noise in the collected stack traces (*e.g.*, the list of native methods involved in the exception). The choice of considering only some specific parts of the *stack traces* was driven by experimental results—available in our on-line appendix [4]—where we tested how the linking accuracy was influenced by the presence/absence of this information.

After cleaning the reports, the remaining text is *augmented* with the source code methods included in the *stack trace*. This step extends the information from the reports with contextual information from the source code, possibly providing additional information useful for the linking process. Also in this case, the choice was not random but driven by the experimental results available in our on-line appendix [4].

Afterwards, the approach performs a systematic Information Retrieval (IR) preprocessing [5] on both the user reviews and *augmented* stack traces aimed at (i) correcting mistakes, (ii) expanding contractions (*e.g.*, *can't* is replaced with *can not*), (iii) filtering nouns and verbs (which are the most representative textual parts of a software artifacts and a general textual description [9]), (iv) removing common words or programming language keywords (the entire list is available in the on-line appendix [4]), and (v) stemming words (*e.g.*, *aiming* is replaced with *aim*). This step returns two documents containing the bag of words representation of the two different sources of information.

Finally, to link the resulting documents we tested three different IR techniques, *i.e.*, (i) the Dice similarity coefficient [14], (ii) the Jaccard index [22], and (iii) the Vector Space Model (VSM) [5]. Specifically, the asymmetric Dice similarity coefficient [5] is defined as follow:

$$\text{Dice}\left(review_j, crash_i\right) = \frac{\left|W_{review_j} \cap W_{crash_i}\right|}{\min\left(\left|W_{review_j}\right|, \left|W_{crash_i}\right|\right)}$$

where W_{review_j} represents the set of words composing a user review j, W_{crash_i} is the set of words contained in an *augmented* stack trace i and the $\min$ function normalizes the Dice score with respect to the number of words contained in the shortest document between j and i. The asymmetric Dice similarity returns values between $[0, 1]$. In our study, pairs of documents having a Dice score higher than 0.5 were considered as linked by the approach.

The Jaccard index, instead, is defined in terms of the following equation:

$$\text{Jaccard}\left(review_j, crash_i\right) = \frac{\left|W_{review_j} \cap W_{crash_i}\right|}{\left|W_{review_j} \cup W_{crash_i}\right|}$$

where W_{review_j} and W_{crash_i} represent the set of words contained in the user review j and the *augmented* stack trace i, respectively. Also in this case, the index varies in the interval $[0, 1]$, and pairs of documents obtaining a Jaccard index higher than 0.5 were considered as candidate links.

Finally, in the VSM user reviews and *augmented* stack traces are represented as vectors of terms (*i.e.*, implemented through a term-by-document matrix [5]) that occur within the two sources. The similarity between pairs of document is given by the cosine of the angle between the corresponding vectors. The selection of these three linking strategies was based on the analysis of previous literature [3], [30], [38], [39], and allowed us to perform a wider analysis of pros and cons of their usage. In Section III-C we firstly report the performance achieved by the three different linking approaches in terms of precision, *i.e.*, number of true positive links retrieved over the total number of candidate links identified [5]. In case a particular app does not have any link between crash-related reviews and stack traces, it is not possible to compute precision (*i.e.*, division by zero). Secondly, we compute the recall, *i.e.*, number of correct links retrieved by an approach over the total number of correct links in the application [5] for all the experimented apps having at least one crash-related reviews and a stack trace related to each other (*i.e.*, if this condition does not hold the golden set for that app would be empty, thus precluding the computation of recall). To this aim, we exploited the golden set of links created in the context of RQ_2.

TABLE II
HIGH LEVEL TAXONOMY (HLT)

Category	#	Sub-Level	Description
Bugs	1,202	Crashes	claims about app crashes
		Features & UI	claims about UI or feature problems
Feature Requests	1,639	Addition	requests for new features
		Improvements	requests for improving existing features
Resources	656	Performance	discussing performance issues
		Battery	discussing battery problems
Request Info	2,149		clarification request for app features
Usability	648		discuss the ease or difficulty to use a feature
Compatibility & Update	352		issues after updates or for device compatibility

III. RESULTS AND DISCUSSIONS

In this section we report the results of the presented research questions and discuss the main related findings.

A. *RQ*$_1$ - *What type of user feedback can we leverage to detect bugs and support testing activities of mobile apps?*

The two content analysis described in Section II-C1 resulted in two different taxonomies: a *high-level* (HLT) and a *low-level* (LLT) one. The resulting HLT, along with the description for the 6 categories, their sub-levels, and the number of reviews for each category is showed in Table II. Since our main goal is to investigate user feedback useful to complement and improve the efficiency of automated testing tools, we mainly focused on the reviews belonging to the *Bugs* category. Indeed, since the current tools are only able to detect runtime unhandled exceptions (and therefore, crashes), we particularly sharped the *Crash* sub-level.

ML Classifier. Before describing the categories of user review feedback composing the LLT, we report the accuracy of the automated ML classifier described in Section II-C1. Indeed, to evaluate its performances, we adopted well-known metrics such as precision, recall, and F1 score [5]. We relied on a 10-fold cross validation on the manual labeled *golden set* as described in Section II-C1. As we can observe from the average results reported in Table III, all the evaluation metrics have values higher than 0.8 and 0.9 for the *Feature & UI Bugs* and *Crashes* sub-categories, respectively. Despite such already accurate classification results, we plan as future work to label more reviews and extend the training set in order to provide more robust and generalizable ML models.

LLT Description. Table IV shows the categories of feedback describing app crashes resulting from the manual analysis of the 534 reviews selected as explained in Section II-C1. It reports also the numbers of feedback for every category. It is worth to notice that a reviews might have more than one assigned label. As is obvious, *Not Meaningful* and *Wrong* cannot be used or are not relevant for the conducted research.

TABLE III
EVALUATION OF MACHINE LEARNING CLASSIFIERS

Category	Precision	Recall	F1 Score
Feature & UI Bugs	0.83	0.82	0.83
Crashes	0.91	0.94	0.92
Average	**0.87**	**0.88**	**0.87**

Specifically, the first one groups user reviews that discuss crashes, but do not provide any fruitful additional information to understand the cause of it (*e.g.*, *"It crashes!"*) The latter instead collects all the reviews that were misclassified from the top-level taxonomy, according to the manual analysis.
Feature category involves the reviews that discuss a crash of the application while using a specific feature of the application, as well the crashes that occurs while moving between activities. Following, two examples that fall in this category.

"I try to scan and IBSN # the app crashes".
"When I try to open links on a page using open a new tab thing crashes".

Resource Management involves every problem connected to the access of a resource, like the load, the upload or the refresh. This category counts also feedback that refers to situation where the crash of the application might occur only where the application itself is handling a considerable amount of data. The following is a review in our dataset that depicts such a situation:

"Crashes Every Time on large messages Used to be good but it can't handle large emails any more. Will they ever fix it?".

Instead, the *Update* category groups the reviews claiming about crashes popping up especially after an app update. Finally, the *Android* branch refers to crashes that might depend more on the system than from the app itself. It is worth to notice that a review could be classified/assigned to more than one category of the taxonomy. Such LLT is used to investigate in RQ_2 which kind of problems are most likely to be detected only trough user reviews (see analysis in Section III-B).

*B. **RQ_2** - How complementary is user feedback information with respect to the outcomes of automated testing tools?*

As explained in Section II-C2, to answer this research question we firstly perform a quantitative analysis to assess the overlap between the crashes detected by the tools and the ones claimed into reviews. Afterwards, we conduct a qualitative analysis to deeper investigate the reasons of such discrepancies.

Quantitative Analysis.
In order to measure the complementarity between tools and reviews, we rely on the overlap metrics described in Section II-C2. Table V reports the value of such metrics along with the counts of the crashes from each source. We can observe as the average percentage of the crashes solely described in the user reviews (I_R) is almost 62%, while the percentage of the crashes that can be observed in both sources (I_S) is 22%.

TABLE IV
LOW LEVEL TAXONOMY

Category	Description	#
Feature	a crash that occurs while trying to use a feature of the app	328
Resource Management	problems due to a resource access (loading, uploading, download, refresh)	195
Update	problems that pop up after an update	67
Android	problems relevant to the Android framework or with the interaction with native apps	15
Not Meaningful	everything that describe a crash but useless or unclear	96
Wrong	reviews misclassified and not belonging to the HLT	16

Finally, the percentage of crashes only detected by testing tools (I_C) is just 16%. Thus, our first finding is summarized as follow.

Finding 1: The number of crashes solely detected by users is higher than the one identified by automated testing tools.

Qualitative Analysis. To better understand the results described above, three of the authors performed a qualitative analysis on the links manually established by the external evaluator (as described in Section II-C2), with the aim to deeper understand the reasons that prevent for a crash claimed in a review to be revealed as well by the testing tools. We following report some of the most interesting cases encountered. As general a result, we noticed that tools miss to detect failures both when a login or an account creation is required and when a complex sequence of inputs need to be executed. For instance, in the `com.amaze.filemanager` app the testing tools were able to collect 7 distinct stack traces, all of them pretty well distinguishable from each other. Our evaluator was able to find links to reviews for 3 out of such 7 stack traces. Reading the user reviews for this app, it immediately comes out that most of the non-linked ones, *i.e.*, crashes likely not revealed by the tools, involved complex sequences of connected input actions, like *copying* or *cut/paste*. Thus, one limitation of current tools is that they are not able to detect problem caused by such *complex user behavioral patterns*.
For the app `com.danvelazsco.fbwrapper`, we were able to collect 4 distinct kind of stack traces. In this case, our validator was able to retrieve links for 3 of such 4 crashes. However, in the available set of user reviews we noticed that several descriptions of crashes occurred when using the *message functionality* of the app. Nevertheless, such problem was not detected at all by the automated testing tools. For this reason, one of the authors tested manually such a feature. It is worth to specify that this application is a Facebook wrapper. Therefore, while some components of the app could be used without be logged in, a large part of its functionalities can be properly tested only after performing a login with a valid account. Unfortunately, such prerequisite was not fulfilled by the automated testing tools. Thus, the author logged into the app with his account and tried to manually reproduce

TABLE V
COMPLEMENTARITY OF REVIEWS AND STACK TRACES

App	I_C	I_R	I_S	$\mathcal{L}$	$\mathcal{S}$	$\mathcal{R}$	$\mathcal{T}$
com.amaze.filemanager	13.6%	68.2%	18.2%	3	4	15	22
con.danvelazco.fbwrapper	23.1%	69.2%	7.7%	3	1	9	13
con.ringdroid	7.1%	71.4%	21.4%	1	3	10	14
com.eleybourn.bookcatalogue	50.0%	50.0%	0.0%	1	0	1	2
com.evancharlton.mileage	11.1%	77.8%	11.1%	1	1	7	9
com.fsck.k9	0.0%	92.3%	7.7%	0	1	12	13
cri.sanity	0.0%	50.0%	50.0%	0	1	1	2
org.liberty.android.fantastischmemo	20.0%	20.0%	60.0%	1	3	1	5
Average	**16%**	**62%**	**22%**				

the scenarios described in the reviews that led to the crash while using the messaging feature. Using the `monitor` tool provided with the Android SDK, she discovered that the app effectively crashes, without useful hints about the cause of crash in the stack traces (that is reported in [4]).

For the `com.ringdroid` app, the tools extracted 4 distinct stack traces. It is worth to notice that such stack traces do not differ too much from each other. Indeed, all of them report a `StaleDataException`. Our external validator linked just one failure to a review. While analyzing the reviews for which there are no links to crashes reported in stack traces, we discovered two interesting findings. At first, a set of reviews for this app discuss about a crash that sometimes occurs when editing or saving a new track. However, such problems was not recognizable in any of the available stack traces. Thus, one of the authors replicated some scenarios described in the reviews and was able to capture the following exception.

```
java.lang.NullPointerException
  at com.ringdroid.RingdroidEditActivity$15.run(RingdroidActivity.
     java:1268)
```

She discovered that such exception is raised when the user use the *edit* feature of a track and then try to rename it. Continuing with such analysis, we noticed that some reviews claimed crashes experienced while trying to *delete* a track. Again, there was no sign of such failure in the collected stack traces. By exercising such feature, we discovered that the app actually forced closing in such situation. However, also in this case there is not any raised unhandled exception reported by the Android monitor employed for this analysis.

At the bottom of this analysis we found that the crashes that are described only in user reviews tend to follow a certain *pattern*. In particular, in this category we often find crashes that occur after a precise sequence of input event, which are *hard to randomly replicate* by automated tools. Examples are the *cut/paste* feature we described for the `com.amaze.filemanager` app or the ones observed for `com.danvelazsco.fbwrapper` when using the messaging feature: another example is the review below, which claims about a crash occurring during a particular swipe input event:

"Quick fix for messages crash Slide in from the right go to preferences and use either desktop version or basic version..."

Similarly, situation of *overload* of an application are difficult

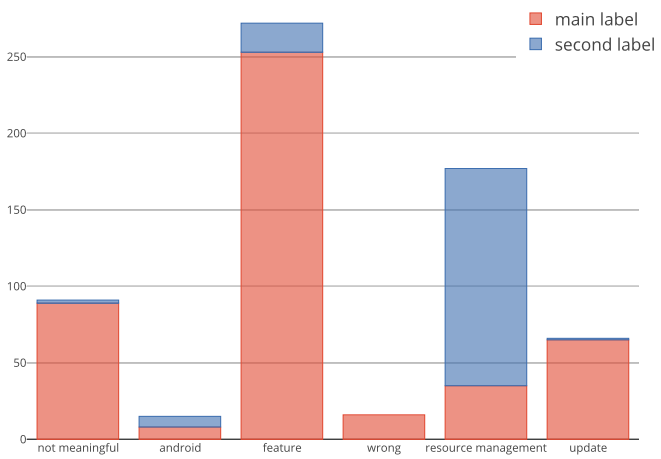

Fig. 2. Distribution of not-linked reviews in LLT

to catch for testing tools. For instance, in `com.ringdroid` an user claim that a particular crash could be due to his large library usage:

"Force closes when I search Maybe the problem is my large library but it truly is unusable..."

To broad our analysis we show in Figure 2 the absolute numbers—discerning between main and secondary labels—of the not-linked reviews for the LLT categories. We can observe that *feature* and *resource management* are the most represented ones. However, it is interesting to notice that reviews tend to be a noisy source of information: the third more occurring category indeed is the *not meaningful* one (*e.g.*, *"It crashes"*). Thus, we can conclude that:

Finding 2: Testing tools potentially miss several failures experienced by users. Such failures are hardly replicable sequences of events or external conditions. Moreover, in many cases crashes information provided in the stack trace are useless or totally not available to understand the root cause of the fault.

We believe that such results highlight the need of novel recommender systems able to distill the actionable information provided in both user reviews and mobile testing tools, providing to developers contextual information of a wider (or more complete) set of bugs present in their applications.

C. RQ_3 - *To what extent can we automatically link the crash-related information reported in both user feedback and testing tools?*

Table VI reports the performance achieved by the three different baselines experimented to link user reviews onto crash reports, *i.e.*, the ones relying on (ii) the Dice similarity coefficient [14], (ii) the Jaccard index [22], and (iii) the Vector Space Model (VSM) [5]. It is worth to note that for two of the considered apps the golden set of links was empty, thus we could not evaluate precision and recall.

Looking at the results, it is clear that the technique based on the Dice coefficient was the one having the best performance

TABLE VI
PERFORMANCE OF THE EXPERIMENTED LINKING APPROACHES (RECALL AND F1 SCORE ARE COMPUTED ONLY FOR THREE OF THE SUBJECT APPS)

App	Dice Linking			Jaccard Linking			VSM Linking		
	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
com.amaze.filemanager	67%	57%	62%	50%	43%	46%	22%	29%	25%
com.danvelazsco.fbwrapper	62%	68%	65%	68%	48%	56%	54%	40%	46%
com.ringdroid	64%	60%	62%	56%	47%	51%	55%	45%	50%
com.eleybourn.bookcatalogue	100%	66%	80%	100%	34%	50%	0%	0%	0%
com.evancharlton.mileage	100%	100%	100%	100%	100%	100%	100%	100%	100%
org.liberty.android.fantastischmemo	100%	100%	100%	100%	100%	100%	100%	100%	100%
cri.sanity	-	-	-	-	-	-	-	-	-
com.fsck.k9	-	-	-	-	-	-	-	-	-
Average	**82%**	**75%**	**78%**	**79%**	**62%**	**70%**	**55%**	**52%**	**54%**

over all the mobile apps of our study, with an average precision of 82% and a recall of 75% (F1 Score = 78%). Thus, we could confirm previous findings showing that such an index provides better insights when comparing user reviews and technical documentation [38]: likely, this is due to the fact that it is more suitable for texts composed of few words. Indeed, it weights the number of common terms between two documents with the number of terms composing the *shorter* document, thus giving more importance to the number of overlapping terms rather than the total number of terms composing the documents [14]. As a result, this characteristic seems to be highly relevant in the context of user reviews (which are usually composed by few meaningful words) because in this way they can be compared with technical documentation such as crash reports.

From a more general point of view, it is also worth noting that the average recall value tells us that other correct links cannot be identified by just relying on a Dice-based technique. While on the one hand this result was somehow expected since we are treating types of documents deeply different from each other (*i.e.*, informal and noisy user reviews vs formal and well-structured crash reports), on the other hand we can claim that more sophisticated approaches (*e.g.*, relying on clustering of user reviews [38], [48] and/or a proper adoption of other textual-based techniques such ad LDA [7], [40]) might be useful to improve the linking capabilities of our methodology and better supporting developers maintenance and testing decisions.

Turning our attention to the other experimented techniques, we could observe that the Jaccard-based one has a precision comparable to the Dice-based approach, however it has a recall value 13% lower. This result confirms that taking into consideration all the terms composing user reviews and crash reports is not the ideal way to find relationships between them.

Finally, the results achieved when using the Vector Space Model were somehow surprising. Indeed, according to Bordag [8] this technique is supposed to work better than the others since it is able to capture semantic relationships that go beyond the mere use of terms overlap. However, in the context of user reviews simpler string-matching approaches tend to be more effective. This is particularly true when considering the recall value, which just reached 54%. On the light of these results, we could conclude that:

> **Finding 3:** It is possible to provide developers with an automated tool for linking crash-related user reviews and stack traces having an average precision and recall of 82% and 75%, respectively. The best approach is the Dice similarity coefficient since it gives a higher importance to overlapping terms rather than the total number of terms composing the treated documents.

Thus, an approach based on Dice similarity coefficient can (i) enable the identification of crashes that are detected by both automated tools and user reviews, (ii) discriminating the crashes that are detected only by automated tools or only by user reviews information. This information is highly important for developers and testers of mobile applications.

IV. THREATS TO VALIDITY

Threats to Construct Validity. The main threat to internal validity in our first research question regards the assessment of the high- and low-level taxonomies which is performed on data provided by human subjects. Indeed, there is a level of subjectivity in deciding whether a user feedback belongs to a specific category of the taxonomy or not. To counteract this issue, the first taxonomy was initially defined by a Master student having two years of experience in mobile development: since she might have committed errors during the process, one of the paper's authors double-checked each review removing potential redundant categories. As for the low-level one, we conducted an iterative content analysis, where three researchers (i) firstly independently categorize user reviews and (ii) then reached a joint decision by merging the categories identified in the first iteration. Similarly, in $\mathbf{RQ}_2$ we exploited the linking provided by an external inspector having experience in Android development. Also in this case, one of the authors double-checked each provided link to verify potential errors. To automatically classify the categories in the high-level taxonomy, we devised a ML approach. Indeed, we tested different classifiers in order to rely on the most effective one. It is worth discussing that we relied on the default configurations of the experimented classifiers since the identification of the best configuration for all of them would

have been too expensive [6]. As part of our future work, we plan to analyze the role of parameters' configuration. It is worth recalling that to have a wide overview of the extent to which an automated solution might support developers in monitoring crash-related user reviews and crash reports, we experimented three textual-based linking approaches such as (i) the Dice similarity coefficient [14], (ii) the Jaccard index [22] and (iii) the Vector Space Model (VSM) [5]. Future effort will be devoted to enlarging the set of baseline linking approaches. Still about $\mathbf{RQ}_2$, it is worth to point out that multiple user reviews might refer to the same crash. We plan to tackle this issue in future work.

Threats to Conclusion Validity. In this category, it is worth observing that to interpret the results of the experimented machine learning approach as well as the linking techniques we relied on well-known metrics such as precision, recall, and F1 Score [5]. Furthermore, we supported our findings by performing fine-grained qualitative investigations with the aim of understanding the motivations behind the achieved results.

Threats to External Validity. About threats to the generalizability of our findings, our study focused on 8 apps having different size and scope. However, we cannot ensure generalizability to other apps. At the same time, it is worth recalling that the analysis required a lot of manual effort and therefore we were not able to work on a larger sample. As part of our future agenda, we plan to extend the dataset.

V. Related Work

In the following, we summarize the main research on (i) automated testing tools for Android apps, and on (ii) mining user reviews from app stores to support maintenance of mobile applications.

A. Automated Android Testing

Automated testing tools can be grouped into three major categories, depending on their exploration strategy [12]: *random* [17], [27], *systematic* [28], [33] and *model-based* [2], [11], [26]. The former employs a random generation of UI and system events which reveals many failures, but it is highly inefficient since it creates too many of such events [12]. The most widely used random tool is Monkey [17], coming directly provided by Google. Tools using a systematic explorations strategy rely on symbolic execution and evolutionary algorithms [12]. CrashScope was the first attempt to augment the crash reports with information useful for its reproduction [33]. Sapienz introduced a multi-objective search-based technique to both maximize coverage and fault revelation, while minimizing the sequence lengths [29]. We use Sapienz [29] and Monkey [17] as main baseline for the reasons we reported in Section II-B. Su *et al.* proposed Stoat, a novel approach that rely on a stochastic model of the app continuously adapted using the Gibbs sampling technique. Recent work on testing of mobile apps are related to the introduction of crowdsourced testing solutions [51]; approaches that record user-guided app executions to complement automated testing techniques [25]; or solutions that generate relevant inputs to unmodified Android apps [27]. To the best of our knowledge this paper represents the first work that propose to leverage information available in user reviews—that previous work showed as effective for maintenance and evolution problems—to identify the main issues users experience while using mobile apps, complementing state-of-art approaches for automating their testing process.

B. Mining App Stores

The concept of app store mining was first introduced by Harman *et al.* [20]. In this context, many researchers focused on the analysis of user reviews to support the maintenance and evolution of mobile applications [31]. Pagano *et al.* [37] analyzed the feedback content and their impact on the user community, while Khalid *et al.* [24] conducted a study on iOS apps discovering 12 types of user complaints posted in reviews. Several approaches have been proposed with the aim to classify useful reviews. AR-Miner [10] was the first one able to discern informative reviews. Panichella *et al.* relied on a mixture of natural language processing, text and sentiment analysis in order to automatically classify user reviews [41], [42]. Gu and Kim [19] proposed an approach that summarizes sentiments and opinions of reviews and classifies them according to 5 predefined classes (aspect evaluation, bug reports, feature requests, praise and others). Ciurumelea *et al.* [13] employed machine learning techniques for the automatic categorization of user reviews on a two level taxonomy. Following the general idea to incorporate user feedback into typical development process, Di Sorbo *et al.* [45], [46] and Scalabrino *et al.* [43], [48] proposed SURF and CLAP, two approaches aimed at recommending the most important reviews to take into account while planning a new release of a mobile application. Finally, Palomba *et al.* [38] proposed ChangeAdvisor, a tool able to suggest the source code artifacts to maintain according to user feedback.

VI. Conclusions & Future Work

In this paper we investigated the possibility to automatically integrate user feedback into the Android testing process. Results confirm the high usefulness of user reviews information for the testing process of mobile apps as it helps to (i) complement the capabilities of such testing tools, by identifying bugs that they cannot reveal and (ii) facilitating the diagnosis of bugs or crashes. Moreover, we proposed an automated approach able to link user reviews to stack traces that showed high accuracy. Those results, together with our qualitative analysis, show how support testing process with the integration of user feedback is highly promising. We believe that our work might pave the way for a *user-oriented testing*, where user feedback is systematically integrated into the testing process. As future direction of this work we envision the definition of a tool able to (i) *summarize* stack traces and user reviews linked together, supporting the bug fixing activities performed by developers, (ii) *prioritize* the generated failures taking into account the user feedback and (iii) *generate* test cases that are directly elicited from user reviews.

REFERENCES

[1] Challenges, methodologies, and issues in the usability testing of mobile applications. *International Journal of Human-Computer Interaction*, 18(3):293–308, 2005.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[4] Authors. Exploring the integration of user feedback in automated testing of android applications. https://github.com/sealuzh/saner18, 2017.

[5] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[7] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[8] S. Bordag. A comparison of co-occurrence and similarity measures as simulations of context. *Computational linguistics and intelligent text processing*, pages 52–63, 2008.

[9] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella. Improving ir-based traceability recovery via noun-based indexing of software artifacts. *Journal of Software: Evolution and Process*, 25(7):743–762, 2013.

[10] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang. Ar-miner: Mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 767–778, New York, NY, USA, 2014. ACM.

[11] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. *SIGPLAN Not.*, 48(10):623–640, Oct. 2013.

[12] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 429–440, Washington, DC, USA, 2015. IEEE Computer Society.

[13] A. Ciurumelea, A. Schaufelbuhl, S. Panichella, and H. C. Gall. Analyzing reviews and code of mobile apps for better release planning. In *SANER*, pages 91–102. IEEE Computer Society, 2017.

[14] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[15] F-Droid. https://f-droid.org/en/.

[16] S. Geisser. *Predictive inference: an introduction*. Chapman & Hall, New York, USA, 1993.

[17] Google. Android monkey. http://developer.android.com/tools/help/monkey.html.

[18] G. Grano, A. Di Sorbo, F. Mercaldo, C. A. Visaggio, G. Canfora, and S. Panichella. Android apps and user feedback: A dataset for software evolution and quality improvement. In *Proceedings of the 2Nd ACM SIGSOFT International Workshop on App Market Analytics*, WAMA 2017, pages 8–11, New York, NY, USA, 2017. ACM.

[19] X. Gu and S. Kim. "what parts of your apps are loved by users?" (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 760–770, 2015.

[20] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: Msr for app stores. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 108–111, June 2012.

[21] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA, 2011. ACM.

[22] P. Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.

[23] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, Oct 2013.

[24] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, May 2015.

[25] X. Li, Y. Jiang, Y. Liu, C. Xu, X. Ma, and J. Lu. User guided automation for testing mobile apps. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 27–34, 2014.

[26] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 111–122, Piscataway, NJ, USA, 2015. IEEE Press.

[27] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[28] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

[29] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[30] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of 25th International Conference on Software Engineering*, pages 125–135, Portland, Oregon, USA, 2003. IEEE CS Press.

[31] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.

[32] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[33] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*, pages 33–44. IEEE Computer Society, 2016.

[34] H. Muccini, A. Di Francesco, and P. Esposito. Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST '12, pages 29–35, Piscataway, NJ, USA, 2012. IEEE Press.

[35] M. Nagappan and E. Shihab. Future trends in software engineering research for mobile apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 21–32, March 2016.

[36] B. of Apps. There are 12 million mobile developers worldwide, and nearly half develop for android first. https://goo.gl/RNCSHC.

[37] D. Pagano and W. Maalej. User feedback in the appstore: An empirical study. In *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 125–134, July 2013.

[38] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE 2017, page to appear. ACM, 2018.

[39] F. Palomba, M. L. Vasquez, G. Bavota, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 291–300, Sept 2015.

[40] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 522–531, 2013.

[41] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, pages 281–290, Washington, DC, USA, 2015. IEEE Computer Society.

[42] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. Ardoc: App reviews development oriented classifier. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 2016.

[43] S. Scalabrino, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta. Listening to the crowd for the release planning of mobile apps. *IEEE Transactions on Software Engineering*, 2017.

[44] Scikitlearn. Gradient boosting classifier. http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html.

[45] A. D. Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 499–510, 2016.

[46] A. D. Sorbo, S. Panichella, C. V. Alexandru, C. A. Visaggio, and G. Canfora. SURF: summarizer of user reviews feedback. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 55–58, 2017.

[47] Statista. Number of apps available in leading app stores as of march 2017. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, Mar. 2017.

[48] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta. Release planning of mobile apps based on user reviews. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 14–24, New York, NY, USA, 2016. ACM.

[49] Wikipedia. N-grams. https://en.wikipedia.org/wiki/N-gram.

[50] S.-J. Yen and Y.-S. Lee. Cluster-based under-sampling approaches for imbalanced data distributions. *Expert Syst. Appl.*, 36(3):5718–5727, Apr. 2009.

[51] T. Zhang, J. Gao, and J. Cheng. Crowdsourced testing services for mobile apps. In *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 75–80, 2017.