Crowdsourcing User Reviews to Support the Evolution of Mobile Apps

Fabio Palomba[1], Mario Linares-Vásquez[2], Gabriele Bavota[3], Rocco Oliveto[4]
Massimiliano Di Penta[5], Denys Poshyvanyk[6], Andrea De Lucia[7]

[1] *Delft University of Technology, The Netherlands,* [2] *Universidad de los Andes, Colombia*
[3] *Università della Svizzera italiana (USI), Lugano, Switzerland,* [4] *University of Molise, Pesche (IS), Italy*
[5] *University of Sannio, Italy,* [6] *The College of William and Mary, USA,* [7] *University of Salerno, Italy*

Abstract

In recent software development and distribution scenarios, app stores are playing a major role, especially for mobile apps. On one hand, app stores allow continuous releases of app updates. On the other hand, they have become the premier point of interaction between app providers and users. After installing/updating apps, users can post reviews and provide ratings, expressing their level of satisfaction with apps, and possibly pointing out bugs or desired features. In this paper we empirically investigate—by performing a study on the evolution of 100 open source Android apps and by surveying 73 developers—to what extent app developers take user reviews into account, and whether addressing them contributes to apps' success in terms of ratings. In order to perform the study, as well as to provide a monitoring mechanism for developers and project managers, we devised an approach, named CRISTAL, for tracing informative crowd reviews onto source code changes, and for monitoring the extent to which developers accommodate crowd requests and follow-up user reactions as reflected in their ratings. The results of our study indicate that (i) on average, half of the informative reviews are addressed, and over 75% of the interviewed developers claimed to take them into account often or very often, and that (ii) developers implementing user reviews are rewarded in terms of significantly increased user ratings.

Keywords: Mobile app evolution, User reviews, Mining app stores, Empirical Study

1. Introduction

Online app stores are promoting and supporting a more dynamic way of distributing software directly to users with release periods shorter than in the case of traditional software systems. For instance, periodically planned releases that follow an established road map have been replaced by continuous releases that become available for an upgrade with a cadence of few weeks, if not days [63]. This phenomenon is particularly evident for—though not limited to—mobile apps, where releases are managed through online app stores, such as the Apple App Store [2], Google Play Market [28], or Windows Phone App Store [57]. The pressure for continuous delivery of apps is a constant in the mobile economy, in which users are demanding new and better features from the apps, and the ratings/reviews provided by the users are a strong mechanism to promote the apps in the market.

The distribution of updates—related to the introduction of new features and to bug fixes—through app online stores is accompanied by a mechanism that allows users to rate releases using scores (*i.e.,* star ratings) and text reviews. The former (*i.e.,* the score) is usually expressed as a choice of one to five stars, and the latter (*i.e.,* the review) is a free text description that does not have a predefined structure and is used to describe informally bugs and desired features. The review is also used to describe impressions, positions, comparisons, and attitudes toward the apps [9, 32, 38, 66, 71]. Therefore, online store reviews are free and fast crowd feedback mechanisms that can be used by developers as a backlog for the development process. Also, given this easy online access to app-store-review mechanisms, thousands of these informative reviews can describe various issues exhibited by the apps in certain combinations of devices, screen sizes, operating systems, and network conditions that may not necessarily be reproducible during regular development/testing activities.

Consequently, by analyzing ratings and reviews, development teams are encouraged to improve their apps, for example by fixing bugs or by adding commonly requested features. According to a Gartner report's recommendation [40], given the complexity of mobile testing *"development teams should monitor app store reviews to identify issues that are difficult to catch during testing, and to clarify issues that cause problems on the users' side"*. Moreover, useful app reviews reflect crowd-based needs and are a valuable source of comments, bug reports, feature requests, and informal user experience feedback [8, 9, 25, 38, 41, 48, 58, 60, 61, 66]. However, because of this easy access to the app stores and lack of control over the reviews' content, relying on crowdsourced reviews for planning releases may not be widely used by mobile developers because (i) a large amount of reviews need to be

analyzed, and (ii) some of these reviews may not provide tangible benefits to the developers [9].

In this paper we investigate *to what extent app development teams can leverage crowdsourcing mechanisms for planning future changes, and how these changes impact user satisfaction as measured by follow-up ratings.* Specifically, we present (i) a mining study conducted on 100 Android open source apps in which we linked user reviews to source code changes, and analyzed the impact of implementing those user reviews in terms of app success (*i.e.,* ratings); and (ii) an online survey featuring 73 responses from mobile app developers, aimed at investigating whether they rely on user reviews, what kind of requirements developers try to elicit from user reviews, and whether they observe benefits in doing that. The survey does not aim at obtaining a different, subjective measure of success. Instead, it helps understanding to what extent developers actually rely on user reviews to improve their apps.

To support our study and to provide developers and project managers with an automated mechanism to monitor whether and how user reviews have been implemented, we have developed an approach named CRISTAL (**C**rowdsourcing **R**ev**I**ews to **S**uppor**T** **A**pp evo**L**ution) able to identify traceability links between incoming app reviews and source code changes likely addressing them, and use such links to analyze the impact of crowd reviews on the development process. Although approaches for recovering traceability links between requirements and source code [1, 51] or even between development mailing lists and code [4] exist, such approaches are not directly applicable in our context because of specific characteristics of user reviews in mobile markets. First, as shown by Chen *et al.* [9], not all the reviews can be considered as useful and/or informative. Also, there is a vocabulary mismatch between user reviews and source code or issues reported in issue trackers. Unlike issue reports and emails, reviews do not refer to implementation details; moreover, because of the GUI-driven nature of mobile apps and the user interaction based on user gestures, user reviews include mobile-specific GUI terms. In order to address these challenges, CRISTAL includes a multi-step reviews-to-code-changes traceability recovery approach that firstly identifies informative comments among reviews (based on a recent approach by Chen *et al.* [9]), and then traces crowd reviews onto commit notes and issue reports by leveraging a set of heuristics specifically designed for this traceability task.

In summary, the paper makes the following contributions:

1. CRISTAL*'s novel reviews-to-code-changes traceability recovery approach.* Albeit being inspired by classic Information Retrieval (IR) based traceability recovery approaches (*e.g.,* [1, 18, 26, 51, 65]), CRISTAL combines these techniques with some specific heuristics to deal with (i) diversity and noise in crowd reviews, and (ii) inherent abstraction mismatch between reviews and developers' source code lexicon.

2. CRISTAL*'s monitoring mechanism.* After linking reviews to changes, the *premier goal* of CRISTAL is to enable developers tracking how many reviews have been addressed, and analyzing the ratings to assess users' reaction to these changes. More specifically, the ability to monitor the extent to which user reviews have been addressed over the projects' evolution history can be used as a support for release planning activities.

3. *Results of an empirical study conducted on 100 Android apps.* The study leverages CRISTAL to provide quantitative evidence on (i) how development teams follow suggestions contained in informative reviews, and (ii) how users react to those changes. The results of the study demonstrate that on average 49% of the informative reviews are implemented by developers in the new app release and this results in a substantial reward in terms of an increase in the app's rating. It should be noted that the value of responding to a user review of a mobile app has never been explored. Our analysis of app reviews and responses from 10,713 top apps in the Google Play Store shows that developers of frequently-reviewed apps never respond to reviews. However, we observe that there are positive effects to responding to reviews (users change their rating 38.7% of the time following a developer response) with a median rating increase of 20%.

4. *Results of a survey* conducted with 73 Android developers and aimed at investigating the developers' perception of user reviews, to what extent they address them and whether they observe any tangible benefits from such an activity. The achieved results show that developers (i) strongly rely on user reviews when planning the new release of their apps, (ii) mainly look in user reviews for bugs experienced by their users or for suggestions for new features, and (iii) confirm the positive effect of implementing change requests embedded in the user reviews on the app's ratings.

5. *A comprehensive replication package* [69] that includes all the materials used in our studies.

Paper structure. Section 2 describes CRISTAL, while Section 3 reports results of a study aimed at evaluating CRISTAL's accuracy. Section 4 defines and reports the design and planning of our empirical investigation, conducted by mining 100 Android apps with CRISTAL, and by surveying 73 Android developers. The study results are reported and discussed in Section 5, while threats that

could affect the validity of the reported results are discussed in Section 6. Section 7 discusses the related literature on app store user reviews mining. Finally, Section 8 concludes the paper.

2. CRISTAL: Linking User Reviews to Source Code Changes

In this section we present CRISTAL, an approach for automatically linking user reviews to source code changes, together with its evaluation in terms of the accuracy of the generated links.

2.1. Overview of the Approach

CRISTAL aims at helping developers to keep track of the informative reviews that have been considered (*i.e.,* implemented) while working on a new app release. CRISTAL follows a three-step process for extracting links between reviews for a release r_{k-1} of an app and commits/issues generated while working on a release r_k (Figure 1). Note that r_k indicates the ID of the next release of an app, on which developers are working, while r_{k-1} is the previously issued release, whose reviews can be possibly be used as input when working on r_k.

The first step aims at collecting user reviews posted for the app release r_{k-1}. These reviews are collected from the app store (Google Play in our case). After that, we prune out *non-informative reviews*, *i.e.,* reviews expressing a positive or negative judgment without explaining the reason. In other words, we remove reviews such as "*this app is terrible*" or " *great app!*". Instead, we keep *informative* reviews, *i.e.,* reviews clearly explaining the reasons for negative scores (*e.g., the app crashes when sharing photos*, or *"there is a lag in the user interface"*) or positive ones (*e.g.,"thanks for adding the new feature for geo-localizing photos!"*, or *"The battery drain issue has been solved, the app now works great!"*). To this aim, we rely on a re-implementation of the AR-Miner classifier [9].

In the second step, for each of the collected *informative reviews* ir_j, the Issue Extractor and the Commit Extractor (Figure 1) collect the issues and the commits, potentially driven by ir_j. Issues opened after the ir_j date, *i.e.,* the date in which the review was posted, and closed before the r_k release date are considered to be potentially linked to ir_j. Also, commits performed after ir_j date and before the r_k release date are considered to be potentially linked to ir_j.

Finally, in the third step each review ir_j and the issues/commits collected for it in the previous step, are provided to the Link Identifier, which identifies candidate traceability links between ir_j and issues/commits by using a customized approach based on Information Retrieval techniques [5]. The set of links retrieved for each *informative* review is stored in a database grouping together all links related to release r_k. This information is leveraged by the Monitoring Component, which creates reports for managers/developers and shows statistics on the reviews that have been implemented. In the following subsections, we provide the details behind each of these major steps.

2.1.1. Collecting Reviews

CRISTAL requires the release dates for r_{k-1} and r_k to retrieve links between reviews posted by users for the app's release r_{k-1} and the commits/issues generated while working on release r_k. These dates, together with the URLs of the app's versioning system and the issue tracker, are the only inputs required by CRISTAL. Note that CRISTAL can identify links between reviews and commits/issues even if r_k has not been issued yet. This scenario might occur when a manager/developer wants to check the implementation status of user reviews received on r_{k-1} before releasing r_k. In this case, the current date is used instead of the r_k's release date. More precisely, if the date of r_k is given as input, then we are able to find links only between a review and the commits contained in the range r_{k-1} and r_k. However, if the date of r_k is not specified, then we are able to find links occurring also on future releases of the app.

CRISTAL downloads the user reviews posted the day after r_{k-1} has been released until the day before r_k has been released. These reviews are those **likely** related to release r_{k-1}. We use the term **likely**, since nothing prevents users from leaving a review referring to a previous app release (*e.g.,* r_{k-2}) while the release r_{k-1} maybe available (*i.e.,* the user did not upgrade to the last available release yet). This problem arises because the current version of Google Play does not allow the user to associate a review with the release of an app that she is reviewing. Note that we consider both negative (*i.e.,* with low ratings) as well as positive (*i.e.,* with high ratings) user reviews. Indeed, while user complaints are generally described in negative reviews, positive reviews could also provide valuable feedback to developers, such as suggestions for new features to be implemented in future releases.

While the reviews retrieved for the release r_{k-1} may contain useful feedback for developers working on the app release r_k, as shown by Chen *et al.* [9], only some reviews contain information that can directly help developers improve their apps (35.1% on average [9]). Thus, most reviews posted by the crowd are simply *non-informative* for app developers, *i.e.,* they do not contain any useful information for improving the app. Such reviews mostly contain (i) pure user emotional expressions (*e.g., "this is a c****y app"*), (ii) very general/unclear comments (*e.g., "the app does not work on my phone"*), and (iii) questions/inquiries (*e.g., "how does the app work?"*). Thus, they are unlikely to be linkable to any commit/issue. For this reason, CRISTAL relies on AR-Miner [9] to filter out those *non-informative* reviews. AR-Miner uses the **E**xpectation **M**aximization for **N**aive **B**ayes (EMNB) [64], a semi-supervised learning algorithm, to classify reviews as *informative* and *non-informative*. In their evaluation, Chen *et al.* [9] showed that AR-Miner achieved an accu-

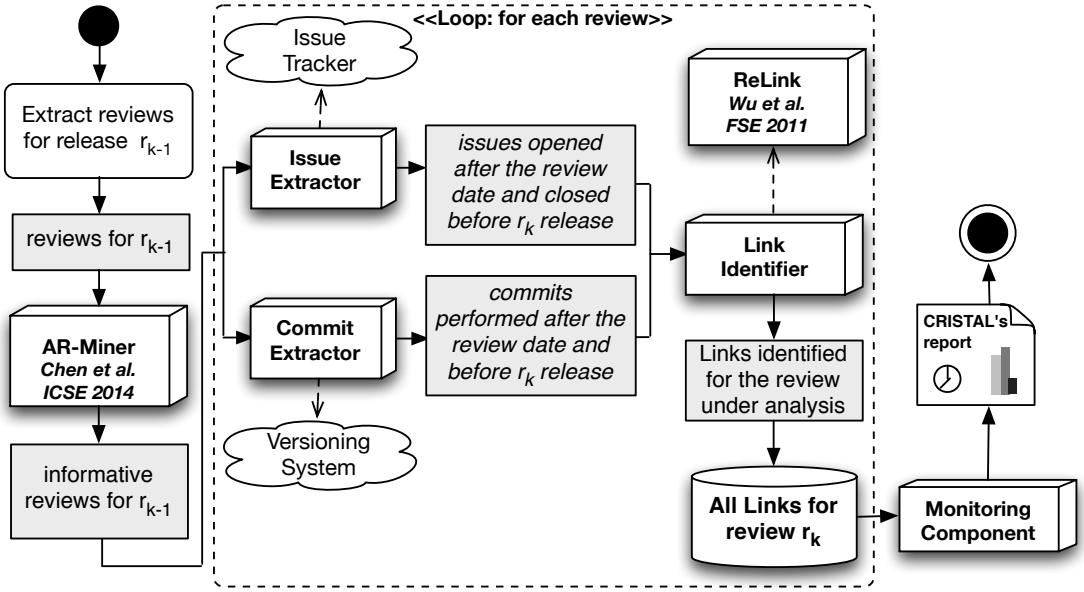

Figure 1: CRISTAL overview: solid arrows represent *information flow*, while dotted arrows represent *dependencies*.

racy between 76% and 88% in classifying *informative/non-informative* reviews. Since the original implementation of AR-Miner is currently not available, we have re-implemented its informative/non-informative review classifier by relying on the Weka library [33], and trained it on 11,157 manually classified reviews coming from 20 mobile applications of the Google Play Store. Specifically, given the title and the corpus of each of the 11,157 user reviews, the first two authors of this paper manually mark a review as *informative* or *non-informative*. Once the training set have been built, we used the EMNB machine learner [64] (as done in the original paper by Chen *et al.* [9]) in order to classify new reviews. It should be noted that these reviews from the training set were not included in the evaluation of CRISTAL (Section 3) and in our empirical study (Section 4 and Section 5).

2.1.2. Extracting Issues and Commits

The set of reviews classified as *informative* by AR-Miner is then further analyzed by CRISTAL for linking to commits/issues. For each informative review ir_j, CRISTAL extracts candidate issues and commits that can be potentially linked to it. Specifically, the Issue Extractor mines the issue tracker of the app of interest, extracting all the issues opened after the ir_j was posted, and closed before the r_k release date (or before the current date). For each issue satisfying these constraints, the Issue Extractor collects (i) the title, (ii) the description, (iii) the name of the person who opened it, (iv) the timestamps of the issue opening/closing, and (v) all comments (including timestamp and author) left on the issue. Our current implementation supports *Jira* [3] and *Bugzilla* [24] issue trackers.

The Commits Extractor mines the change log of the versioning system hosting the app of interest by selecting all the commits performed after ir_j was posted and before the r_k release date (or before the current date). For each commit satisfying such constraints, the Commits Extractor collects (i) the timestamp, (ii) the set of files involved, (iii) the author, and (iv) the commit message. Our current implementation of the Commits Extractor works on *SVN*, *CVS*, and *Git*. Finally, the set of issues/commits extracted during this step are provided, together with the referred review ir_j, to the Link Identifier for detecting traceability links (see Figure 1).

2.1.3. Detecting Links

The Link Identifier component is responsible for establishing traceability links between each informative review ir_j and the set of issues/commits selected as candidates to be linked by the Issue Extractor and the Commit Extractor. Although our main goal is to link reviews to changes (and, therefore, to commits), we also identify links to issues (that are, in turn, linked to commits) because in this way we can leverage their textual content, often richer than those of commit messages. Establishing links between reviews and issues or commits requires, in addition to using IR-based techniques, some appropriate adaptations keeping in mind requirements of the specific context (*i.e.,* mobile apps and user reviews) such as: (i) discarding words that do not help to identify apps' features, (ii) considering GUI level terms (*e.g.,* reviews have words *window*, *screen*, *activity* that refer to Android GUIs rendered by Android Activities) when performing the linking, and (iii) considering the length (*i.e.,* verbosity) difference between reviews and issues/changes. The detection process consists of several key steps explained below.

Step 1. Linking Informative Reviews and Issues. The linking between ir_j and issues consists of the following steps:

- ***Text normalization***. The text in the review and the text in the issue title and body, consisting of its description, are normalized by performing identifier splitting for *CamelCase* and underscore (we also kept the original identifiers), stop words removal, and stemming (using the Porter stemmer [73]). We built an ad-hoc stop word list composed of (i) common English words, (ii) Java keywords, and (iii) words that are very common in user reviews, and, thus, are not highly discriminating. To identify the latter words, we consider the normalized entropy [15] of a given term t in user reviews using the formula shown in Equation 1:

$$E_t = -\sum_{r \in R_t} p(t|r) \cdot \log_\mu p(t|r) \quad (1)$$

where R_t is the set of app reviews containing the term t, μ is the number of reviews on which the terms entropy is computed, and $p(t|r)$ represents the probability that the random variable (term) t is in the state (review) r. Such probability is computed as the ratio between the number of occurrences of the term t in the review r over the total number of occurrences of the term t in all the considered reviews. E_t is in the range $[0, 1]$ and the higher the value, the lower the discriminating power of the term. To estimate a suitable threshold for identifying terms having a high entropy, we computed the entropy of all the terms present in the reviews of a set of 1,000 Android apps considered in a previous study [6]. This resulted in entropy values for 13,549 different terms. Given Q_3 the third quartile of the distribution of E_t for such 13,549 terms, we included in the stop word list terms having $E_t > Q_3$ (*i.e.,* terms having a very high entropy), for a total of 3,405 terms. Examples of terms falling in our stop word list are *work* (very common in sentences such as *does not work*—$E_{work} = 0.93$), *fix*, and *please* (*e.g., please fix*—$E_{fix} = 0.81$, $E_{please} = 0.84$), *etc.* Instead, terms like *upload* ($E_{upload} = 0.24$) and *reboots* ($E_{reboots} = 0.36$) are not part of our stop word list, since showing a low entropy (high discriminating power) and likely describing features of specific apps. Including the entropy-based stop words into the stop words list helped us to improve the completeness of the identified links (*i.e.,* recall) by +4% and the precision by +2%. The resulting stop word list can be found in our replication package [69].

- ***Textual similarity computation***. We use the asymmetric Dice similarity coefficient [5] to compute a textual similarity between a review ir_j and an issue report is_i (represented as a single document containing the issue title and short description) using the formula reported in the Equation 2:

$$sim_{txt}(ir_j, is_i) = \frac{|W_{ir_j} \cap W_{is_i}|}{min(|W_{ir_j}|, |W_{is_i}|)} \quad (2)$$

where W_k is the set of words contained in the document k and the *min* function that aims at normalizing the similarity score with respect to the number of words contained in the shortest document (*i.e.,* the one containing less words) between the review and the issue. The asymmetric Dice similarity ranges in the interval $[0, 1]$. We used the asymmetric Dice coefficient instead of other similarity measures, such as the cosine similarity or the Jaccard similarity coefficient [39], because in most cases user reviews are notably shorter than issue descriptions and, as a consequence, their vocabulary is fairly limited. Specifically, the use of asymmetric Dice similarity helps matching a review against the issue covering most of its words. A similar approach has been applied in the past for matching help requests of junior developers (newcomers) with discussion corpora of senior open source developers [7].

- ***Promoting GUI-related terms***. Very often, users describe problems experienced during the apps' usage by referring to components instantiated in the apps' GUI (*e.g., when clicking on the start button nothing happens*). Thus, we conjecture that if a review ir_j and an issue report is_i have common words from the apps' GUI, it is more likely that is_i is related (*i.e.,* due) to ir_j and thus, a traceability link between these two should be established. Thus, while retrieving links for an Android app a_k we build an *a_k's GUI terms list* containing words shown in the a_k's GUI (*e.g.,* buttons' labels, string literals). Such words are extracted by parsing the `strings.xml` file, found in Android apps, which is used to encode the string literals used within the GUI components. Note that the presence of a term t in the *a_k's GUI terms list* has a priority over its presence in the stop word list, *i.e., t* is not discarded if present in both lists. Once the *a_k's GUI terms list* has been populated, GUI-based terms shared between a review ir_j and an issue report is_i are rewarded as:

$$GUI_{bonus}(ir_j, is_i) = \frac{|GUI_W(ir_j) \cap GUI_W(is_i)|}{|W_{ir_j} \cup W_{is_i}|} \quad (3)$$

where $GUI_W(k)$ are the GUI-based terms present in the document k and W_k represents the set of words present in the document k. The $GUI_{bonus}(ir_j, is_i)$ is added to the textual similarity between two

documents, obtaining the final similarity used in CRISTAL and shown in Equation 4:

$$sim(ir_j, is_i) = 0.5 \cdot sim_{txt}(ir_j, is_i) + \\ +0.5 \cdot GUI_{bonus}(ir_j, is_i) \quad (4)$$

Note that both $GUI_{bonus}(ir_j, is_i)$ and the textual similarity range in the interval $[0, 1]$. Thus, the overall similarity is also defined in $[0, 1]$. In our initial experiments, we evaluated CRISTAL without using the GUI_{bonus}, thus purely relying on the textual similarity between ir_j and is_i. During the evaluation of links found by CRISTAL, we noticed that the noise contained in the text did not allow the approach to correctly detect some links. We also observed that the terms belonging to the GUI of the application are often reported by users to manifest problems occurred during their use. Indeed, when we defined the GUI_{bonus}, we found that it helped obtaining additional improvement in terms of recall and precision up to 1% and 5%, respectively.

- ***Threshold-based selection***. Pairs of (review, issue) having a similarity higher than a threshold λ are considered to be linked by CRISTAL. The way this threshold has been calibrated is described in Section 3.1.

Step 2. Linking Informative Reviews and Commits. The process of linking each informative review ir_j to a set of commits C_j is quite similar to the one defined for the issues. However, in this case, the corpus of textual commits is composed of (i) the commit note itself, and (ii) words extracted from the names of modified files (without extension and by splitting compound names following camel case convention). Basically, we have integrated the text from commit notes with words that are contained in names of classes being changed (excluding inner classes). This additional text better describes what has been changed in the system, and can potentially match words in the review especially if the commit note is too short and if the names of the classes being changed match domain terms, which are also referred from within the reviews. We chose not to consider the whole corpus of the source code changes related to commits, because it can potentially bring more noise than useful information for our matching purposes. In fact, we experimented with four different corpora: (a) commit notes only, (b) commit notes plus words from file names, (c) commit notes plus corpus from the source code changes, and (d) commit notes plus words from file names and the result of the unix diff between the modified files pre/post commit. The option (b) turned out to be the one exhibiting highest recovery precision. In particular, the difference in favor between (b) and (a) was +11% in terms of recall and +15% in terms of precision, between (b) and (c) it was +37% for recall and +32% for precision, and between (b) and (d) it was +4% for recall and +6% for precision.

Step 3. Linking Issues and Commits. When all the links between each informative review ir_j and issues/commits have been established, CRISTAL tries to enrich the set of retrieved links by linking issues and commits. If ir_j has been linked to an issue is_i and the issue is_i is linked to a set of commits C'_i, then we can link ir_j also to all commits in C'_i. To link issues to commits we use (and complement) two existing approaches. The first one is the regular expression-based approach by Fischer *et al.* [23] and a re-implementation of the ReLink approach proposed by Wu *et al.* [82]. ReLink considers the following constraints when linking a commit to an issue: (i) matching the committer/authors with issue tracking contributor name/email; (ii) the time interval between the commit and the last comment posted by the same author/contributor on the issue tracker must be less than seven days; and (iii) Vector Space Model (VSM) [5] cosine similarity between the commit note and the last comment referred to above, which is greater than 0.7. ReLink has been shown to link issues to commits with high accuracy (89% for precision and 78% for recall) [82]. For further details on *ReLink calibration* refer to the original work by Wu *et al.* [82].

Step 4. Filtering Links. Finally, a filtering step is performed by the Link Identifier to remove spurious links, related to reviews that have been addressed already. As explained before, the current version of Google Play does not associate a review with an app release that the reviewer is using, thus allowing users to post reviews related to issues, which could have been already addressed in the past. To mitigate this problem, we also extract changes and issues before r_{k-1} release date (using the Issue Extractor and the Commit Extractor), and use the Link Identifier for tracing a review to changes already addressed in r_{k-1}. If a review is linked to past changes, all links related to it are discarded by the Link Identifier.

2.1.4. Monitoring Crowdsourced Reviews with CRISTAL

Once CRISTAL builds traceability links between reviews and commits, the Monitoring Component can be used to track whether developers implement the crowdsourced reviews. First, the links can be used during the development of r_k release to allow project managers keep track on which requests have (not) been implemented. Indeed, the Monitoring Component creates a report containing (i) the list of informative reviews (not) implemented for a given date, and (ii) the *review coverage*, providing an indication of the proportion of informative reviews that are linked to at least one commit/issue. Specifically, given the set of informative reviews IR_{k-1} posted after release $k-1$, and the subset of these reviews for which exists a traceability link towards a change ($TIR_{k-1} \subseteq IR_{k-1}$), the *review coverage* is computed as TIR_{k-1}/IR_{k-1}; a *review coverage* equals to 1 means that all the informative reviews in r_{k-1} were linked to an issue or a commit. Second, the Monitoring Component can

Table 1: Apps considered when evaluating CRISTAL accuracy.

App	KLOC	Reviews (Informative)	Commits	Issues
AFWall+ 1.2.7	20	161 (53)	181	30
AntennaPod 0.9.8.0	33	528 (112)	1,006	21
Camera 3.0	47	1,120 (299)	2,356	30
FrostWire 1.2.1	1,508	743 (230)	1,197	182
Hex 7.4	33	346 (119)	1,296	155
K-9 Mail 3.1	116	546 (174)	3,196	30
ownCloud 1.4.1	29	306 (85)	803	149
Twidere 2.9	114	541 (157)	723	23
Wifi Fixer 1.0.2.1	45	860 (253)	1,009	34
XBMC Remote 0.8.8	93	540 (167)	744	28
Overall	**2,038**	**5,691 (1,649)**	**12,307**	**682**

be leveraged after release r_k has been issued. In this case, besides providing all information described above, it also includes the gain/loss in terms of average rating with respect to r_{k-1} in the generated report. This last piece of information is the most important output of CRISTAL, because it can provide project managers with important indications on the work being done while addressing r_{k-1}'s reviews.

3. Evaluating the Linking Accuracy of CRISTAL

As a preliminary step before understanding how developers can use CRISTAL in practice, we need to evaluate its accuracy in the identification of traceability links between user reviews and commits or issues. To this aim, we perform an assessment of CRISTAL using the development history and user reviews of ten open source Android apps listed in Table 1. For each app, the table reports the analyzed release, the size in KLOCs, the number of reviews for the considered release (and the number of informative reviews as detected by AR-Miner, in parenthesis), commits, and issues. The selection of the ten apps was mainly driven by the need to consider open source Android apps published on the Google Play market with versioning system and issue tracker publicly accessible. In the second place, we picked Android apps diversifying the selection in terms of app category (*e.g.,* multimedia, communication), size, and the number of issues and commits (see Table 1).

We formulated the following research question, which has the preliminary purpose to evaluate CRISTAL 's abilities in liking user reviews onto issues/commits:

- **RQ**$_1$: *How accurate is* CRISTAL *in identifying links between informative reviews and issues/commits?*

3.1. Study Procedure

While evaluating the traceability recovery precision simply requires a (manual) validation of the links extracted by the approach, the evaluation of the recall is more challenging because the knowledge of all the links between user reviews and subsequent issues and commits is needed. Therefore, to assess CRISTAL in terms of precision and recall, we followed a three-step process: (i) for each of the considered apps, we manually created an oracle reporting the traceability links between reviews and issues/commits; (ii) we ran CRISTAL on the same set of apps to automatically identify links between reviews and issues/commits; and (iii) we compared the links automatically identified by CRISTAL with the ones identified via manual inspection to assess the recall and precision of our approach.

Table 2: Agreement in the Definition of the Oracles.

App	$E_1 \cup E_2$	$E_1 \cap E_2$	Agreement
AFWall+	15	11	73%
AntennaPod	6	4	67%
Camera	11	9	82%
FrostWire	3	3	100%
Hex	24	19	79%
K-9 Mail	9	6	67%
ownCloud	23	13	57%
Twidere	13	11	85%
Wifi Fixer	16	9	57%
XBMC Remote	57	38	67%
Overall	**177**	**123**	**69%**

To manually create the set of labeled traceability links between user reviews and commits for each app, we independently inspected reviews, issues, and commit logs/messages, in pairs (*i.e.,* two evaluators were assigned to each app), with the aim of identifying traceability links between reviews and issues/commits. In total, six of the authors were involved as evaluators and, for each app to analyze, each of them was provided with the app's source code and three spreadsheets: (i) the first reporting all user reviews for the app of interest, with the possibility of ordering them by score and review date, (ii) the second containing all the issues, characterized by title, body, comments, and date, and (iii) the third reporting for each commit performed by developers its date, commit message, and the list of involved files. Given the number of reviews/issues/commits involved (see Table 1), this process required approximately five weeks of work. This is actually the main reason why the accuracy assessment of CRISTAL was done on a relatively limited set of apps. After having completed this task, the produced oracles were compared, and all involved authors discussed the differences, *i.e.,* a link present in the oracle produced by one evaluator, but not in the oracle produced by the other.

To limit the evaluation bias, we made sure that at least one of the inspectors for each pair did not know the details of the approach. Also, the oracle was produced before running CRISTAL, hence none of the inspectors knew the potential links identified by CRISTAL. Table 2 summarizes the agreement for each app considered in the study, reporting the union of links retrieved by two evaluators ($E_1 \cup E_2$), their intersection ($E_1 \cap E_2$), and the level of

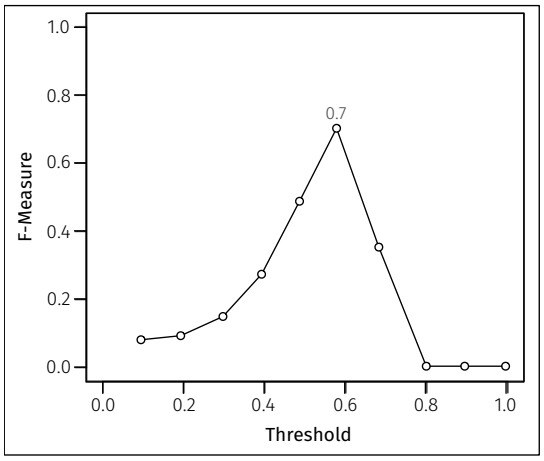

Figure 2: Results of the calibration performed for the λ threshold.

agreement computed as Jaccard similarity coefficient [39], *i.e.,* link intersection over link union. As we can see, while there is not always full agreement on the links to consider, the overall agreement of 69% is quite high and, combined with the open discussion performed by the evaluators to solve conflicts, it ensures high quality of the resulting oracle (finally composed of 141 links).

After building the *unified* oracle for each app, we used well-established metrics to evaluate the recovery accuracy of CRISTAL, namely *recall* and *precision* [5], reported in Equations 5 and 6:

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

In particular, TP represents the number of true positive links detected by CRISTAL, FP the number of links wrongly identified by the approach, and FN the number of correct links that are not identified by the approach. Also, since there is a natural trade-off between recall and precision, we assess the overall accuracy of CRISTAL by using the harmonic mean of precision and recall, known as the F-measure [5]:

$$FMeasure = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{7}$$

3.1.1. Calibration of the ranked list cutting threshold

As explained in Section 2.1.3, the presence of a link between a user review and an issue or commit is determined by cutting the ranked list of candidate links, and considering only links for which the similarity (between a review and an issue or a commit) is greater than a threshold λ. To calibrate such a threshold, we have (i) taken 10 apps different from those considered in this study including a total of 165 traceability links between commits/issues and user reviews, (ii) built an oracle by following the same protocol determined above, and (ii) computed precision, recall and F-Measure for different values of λ. Specifically, we experimented with values of λ ranging between 0.1 and 1.0 with a step of 0.1. The list of apps considered in the calibration process is available in our replication package [69]. The results of our calibration are reported in Figure 2: The best F-Measure was achieved with $\lambda = 0.6$. This is the λ value that we will exploit in the rest of this work, including the study described in Section 4.

Table 3: **RQ$_1$:** recall, precision, and F-measure achieved by CRISTAL, when considering issues only.

App	#Links (Oracle)	#TP	#FP	precision	recall	F-measure
AFWall+	5	4	2	67%	80%	73%
AntennaPod	0	0	0	-	-	-
Camera	3	3	0	100%	100%	100%
FrostWire	0	0	0	-	-	-
Hex	7	6	3	67%	86%	75%
K-9 Mail	3	2	0	100%	67%	80%
ownCloud	7	6	2	75%	86%	80%
Twidere	4	3	2	60%	75%	67%
Wifi Fixer	5	3	1	75%	60%	67%
XBMC Remote	9	4	3	57%	44%	50%
Overall	**43**	**31**	**13**	**70%**	**72%**	**71%**

Table 4: **RQ$_1$:** recall, precision, and F-measure achieved by CRISTAL, when considering commits only.

App	#Links (Oracle)	#TP	#FP	precision	recall	F-measure
AFWall+	10	7	0	100%	70%	82%
AntennaPod	3	2	1	67%	67%	67%
Camera	6	5	1	83%	83%	83%
FrostWire	2	1	0	100%	50%	67%
Hex	13	9	1	90%	69%	78%
K-9 Mail	4	3	2	60%	75%	67%
ownCloud	7	6	3	67%	86%	75%
Twidere	9	6	1	86%	67%	75%
Wifi Fixer	6	5	4	56%	83%	67%
XBMC Remote	38	28	5	85%	74%	79%
Overall	**98**	**72**	**18**	**80%**	**73%**	**77%**

3.2. Analysis of the Results

Tables 3 and 4 report (i) the size of the oracle (column #Links), (ii) the number of true and false positive links retrieved by CRISTAL (column #TP and #FP, respectively), (iii) precision, recall, and F-measure achieved by CRISTAL when tracing user reviews onto issues and commits, respectively. Overall results, considering issue reports and commit notes altogether are reported in Table 5. Note that, for the ten considered apps, we never obtained a candidate traceability link between a review and a commit performed before the review was posted (see Section 2.1.3).

On the one hand, when considering the results obtained by CRISTAL for links between user reviews and issues (Table 3), we observed that its overall F-measure is 71%, with the precision of 70% and a recall of 72%. It is important to point out that in two of the considered apps, *i.e.,* AntennaPod and FrostWire, we did not find any traceability link and, for this reason, we can not compute the accuracy metrics. In such cases, CRISTAL did not output any false positive link. On the other hand, when

Table 5: **RQ$_1$:** recall, precision, and F-measure achieved by CRISTAL, when considering both commits and issues.

App	#Links (Oracle)	#TP	#FP	precision	recall	F-measure
AFWall+	15	11	2	85%	73%	79%
AntennaPod	3	2	1	67%	67%	67%
Camera	9	8	1	89%	89%	89%
FrostWire	2	1	0	100%	50%	67%
Hex	20	15	4	79%	75%	77%
K-9 Mail	7	5	2	72%	71%	71%
ownCloud	14	12	5	71%	86%	78%
Twidere	13	9	3	75%	69%	72%
Wifi Fixer	11	8	5	62%	73%	67%
XBMC Remote	47	32	8	80%	68%	74%
Overall	**141**	**103**	**31**	**77%**	**73%**	**75%**

considering the accuracy of our approach in retrieving links between user reviews and commits (Table 4), we can observe that CRISTAL achieves a higher overall precision, *i.e.,* 80%, while its recall reaches 73% (F-measure=77%). In summary, it is interesting to notice how, in the end, the results are better when tracing user reviews onto commit notes, rather than when tracing onto issue reports. Although such a difference may or may not generalize to other apps, it has an important implication: indeed, CRISTAL can be applied also in the absence of an issue tracker, a scenario that is quite frequent in practice, since mobile developers often rely only on reviews for problem reporting.

When the links found by our approach considering both issues and commits are combined, the results continue to be relatively positive (see Table 5), showing an overall precision of 77% and a recall of 73% (F-measure=75%). Also, the precision achieved by CRISTAL is never lower than 60%. A manual analysis of false positive links identified by CRISTAL highlighted that those were mostly due to pairs of reviews and commits that, even exhibiting quite high textual similarity, did not represent cases where app developers were implementing user comments. For instance, consider the following review left by a user for the XBMC Remote app (open source remote control for XBMC home theaters):

> **Rating:** ⋆⋆⋆⋆⋆ - April 25, 2014
> *App works great.*
> I did have a few null pointer exceptions but they were only for the videos I had no metadata for.

CRISTAL links this review to a commit modifying classes VideoClient and VideoManager and having as commit note: *Handle empty playlists correctly: Do not throw* NullPointerExceptions *and* IndexOutOfBoundExceptions *when retrieving an empty playlist.* While the user was reporting a problem with videos without metadata, the commit actually aimed at fixing a bug with empty playlists. However, the high number of shared terms (*i.e., "null"*, *"pointer"*, *"exception"*, *"video"*) lead to a high similarity between the review and the commit, with the consequent identification of a false positive link. Indeed, most of the other terms present in the review (*i.e.,* all but metadata) were part of our stop word list. For nine out of ten apps the recall is above 60%, reaching peaks close to 90% for two of them. On the negative side, the lowest recall value is achieved on FrostWire, where the 50% recall value is simply due to the fact that just one out of the two correct links is retrieved by CRISTAL. We manually analyzed the false negatives, *i.e.,* links present in our oracle and missed by CRISTAL, to understand their reasons. We noticed that the missing links were mainly due to a vocabulary mismatch between the reviews and the commits/issues to which they were linked. For instance, the following review was left by a FrostWire's user:

> **Rating:** ⋆ - October 7, 2013
> *Stopped working*
> Doesn't download any song. Needs to be fixed.

FrostWire is an open source BitTorrent client available for several platforms and the user is complaining about problems experienced with downloading songs. In our oracle, the review above is linked to a commit performed to fix such a problem. However, the terms being used in the commit note, as well as the words contained in the names of the modified files, are different from those used in the review. Indeed, CRISTAL links this review to a commit performed by a developer while working on the app release 1.2.2, and dealing with the addition of a download log (*i.e.,* the download history of a user) to the FrostWire app. The linking was found because the commit, having as description "*Added download actions log*", involved several other code files, such as `StopDownloadMenuAction.java`, which share with the review the words "stop" and "download". Since the other review's words (all but *song*) are present in the stop word list adopted by CRISTAL, the Dice similarity between the review and the commit appears to be high, thus leading to a false positive link. In summary, as any other approach based on textual similarity matching, CRISTAL may fail whenever the presence of common words does not imply similar meaning of the review and the commit note. The usage and evaluation of more sophisticated similarity techniques, *e.g.,* Latent Semantic Indexing [19], is part of our future agenda.

Answer to RQ$_1$. Despite few cases discussed above, CRISTAL exhibits high accuracy in retrieving links between crowd reviews and issues/commits, with an overall precision of 77% and recall of 73%. The traceability towards commit notes has a higher performance, reaching an overall precision of 80% while keeping recall at 73%.

4. Empirical Study Design

The *goal* of this study is to investigate (i) to what extent developers use crowdsourced reviews for planning and performing changes to be implemented in the next releases, and (ii) possible gains (if any) for the app's success as reflected in improved ratings/reviews. The *perspective* is of

Table 6: Characteristics of the apps considered in the study.

Category	#Apps	#Reviews (Informative)	#Classes	#Commits	#Issues
Arcade	2	103 (25)	2,631	972	28
Books and Reference	4	226 (82)	2,982	1,021	53
Brain and Puzzle	4	778 (255)	487	2,817	61
Comics	2	254 (73)	356	1,477	15
Communication	15	1,417 (430)	21,710	51,921	260
Education	2	919 (277)	891	2,881	31
Entertainment	6	822 (248)	1,971	3,818	184
Finance	5	1,621 (492)	910	8,172	43
Lifestyle	2	1,110 (327)	398	3,741	21
Media and Video	6	452 (121)	2309	3,572	93
Music and Audio	4	1,284 (388)	988	5,566	55
News and Magazines	3	755 (220)	872	2,917	38
Personalization	5	1,440 (492)	915	5,981	78
Photography	3	965 (299)	595	6,216	49
Productivity	14	2,981 (923)	17,634	23,817	239
Social	5	1,984 (612)	1,119	16,982	70
Tools	10	1,192 (364)	8,578	10,274	167
Travel	8	921 (164)	829	1,663	113
Total	**100**	**19,224 (5,792)**	**66,175**	**153,808**	**1,598**

researchers who are interested in investigating how and why monitoring app store ratings and reviews is worthwhile to build recommendation systems able to suggest fixing issues that cause user dissatisfaction. The *context* of this study consists of:

1. Change history data and user reviews from 100 Android apps belonging to 18 different categories present on the Google Play Store, including the ten apps considered in the CRISTAL's accuracy assessment. We exploit these apps in the context of a *mining study* aimed at providing quantitative insights about the goals of the study.

2. 73 professional developers (hereinafter referred to as "participants") providing their opinions about the goals of the study. These developers have been involved in a *survey* aimed at gathering qualitative insights about the goals of the study.

The collected working data set is released as a part of our replication package [69].

4.1. Research Questions and Study Procedure

This study aims at empirically analyzing the effect of using app store reviews (for planning and implementing future changes) on the apps' success (in terms of increase of the user rating on the Google Play market). As explained in Section 2.1.4, this represents a possible application of CRISTAL. Specifically, we address the following research questions:

- **RQ$_2$:** *To what extent do developers fulfill reviews when working on a new app release?*
- **RQ$_3$:** *What is the effect of a crowd review mechanism (for planning and implementing future changes) on the app success?*

The goal of first research question is to empirically identify whether app developers take into account informative reviews when working on a new app release. We pursue this goal by mining data from the 100 studied Android apps with the support of CRISTAL, and by surveying the 73 apps developers involved as participants. On the other hand, by addressing **RQ$_3$** we aim at empirically verifying the benefit (if any) of such crowdsourcing activity, in this case measured in terms of app scores. Again, we rely on both app history mining and answers provided by the surveyed developers.

4.2. Context Selection

This section describes the context of both our mining study and of the survey.

4.2.1. Mining 100 Android Apps using CRISTAL

Table 6 provides information about the subject apps of our study by reporting, for each category, (i) the number of apps analyzed from the category (column #Apps), (ii) the total number of reviews of the apps in the category (and in parenthesis the number of informative reviews as detected by AR-Miner), (iii) the total number of classes of the apps (column #Classes), (iv) the size of the change history in terms of commits (column #Commits), and (v) the number of issues on the issue trackers of the apps in the category (column #Issues). Such set of apps has been selected taking into account different factors. In order to collect them, we developed a crawler able to download free Android apps. It has been executed over the Google Play Store for one week and, in the end, the crawler collected 2,149 apps. From this initial set, we discarded all the apps having (i) a number of commits fewer than the first quartile of the distribution of commits for all the apps (*i.e.,* 100 commits), (ii) or a number of user reviews lower than the first quartile of the distribution of reviews for all the apps (*i.e.,* 50 reviews). In addition, we only considered apps for which a history composed of multiple releases were accessible. That is, when analyzing reviews for a release r_{k-1}, there must be at least one subsequent release r_k in which reviews raised for r_{k-1} have been possibly addressed. These filters are needed since we are interested in collecting and analyzing the apps which have a good amount of both commits and user reviews. Indeed, having a small number of them may result in the application of our approach in a scenario that does not reflect a real use case. This filtering process led to the 100 apps considered in this study.

4.2.2. Surveying Android App Developers

We sent invitations to developers of 7,097 Android apps that we used in a previous work [45] and of the 100 apps considered in our study. To identify them, we mined the Google play market's webpages of the 7,197 apps to extract the email addresses of the related developers. This was possible thanks to the *Contact Developer* field present in each webpage presenting an app on the market. We automatically removed all duplicated e-mail addresses due to multiple apps developed by the same developer(s). Also,

Table 7: Breakout of the Involved Developers across App Categories.

Category	#Developers
Arcade	3
Books and Reference	4
Brain and Puzzle	1
Comics	2
Communication	8
Education	5
Entertainment	4
Finance	2
Health and fitness	3
Libraries and demo	1
Lifestyle	1
Media and Video	3
Medical	1
Music and Audio	3
News and Magazines	8
Personalization	1
Photography	1
Productivity	9
Racing	1
Shopping	1
Social	2
Tools	1
Transportation	6
Travel	1
Weather	1

we manually pruned out addresses related to customer support contact points (*e.g.,*`ask@`, `support@`, etc). In the end, we obtained a list of 2,567 developers to contact. Each developer received an email with instructions on how to participate in our study and a link to the website hosting our survey (details of how data was collected are reported in Section 4.3.2). After the survey deadline (one month) expired, we collected 73 responses in total, with respondents spread over a large variety of app categories as shown in Table 7. Five of respondents were involved in the development of four of the 100 apps considered in this study, *i.e.,* BANKDROID, CATLOG, GNUCASH, and MICDROID.

4.3. Data Collection and Analysis

In the following we report the data collection and analysis process performed in the mining study and in the survey conducted with apps' developers.

4.3.1. Mining 100 Android Apps using CRISTAL

To address $\mathbf{RQ}_2$, we used the CRISTAL's MONITORING COMPONENT (Section 2.1.4) to identify the *review coverage* (*i.e.,* the percentage of informative reviews that are linked to at least one issue/commit) in the 100 apps we considered. Indeed, if a review is covered (*i.e.,* it is linked to at least one issue/commit), it is likely that it has been taken into account by developers trying to fix the problem raised by a user in her review. Note that when computing the *review coverage* we only considered informative reviews as detected by AR-MINER, because *non-informative* reviews do not provide improvement suggestions for apps.

Before applying CRISTAL to measure reviews coverage, we need to quantify the instances when classifying user reviews as *covered* or *not covered*. For each informative review ir_j, CRISTAL classifies it as *covered* if it is able to identify a link between ir_j and at least one issue/commit, otherwise the review is classified as *not covered*. Also in this case, we need an oracle, *i.e.,* the set of reviews *covered* to compute the classification accuracy of CRISTAL. For this reason, we evaluated CRISTAL on the same set of ten apps used to answer $\mathbf{RQ}_1$ as in our previous study. The results are analyzed through the confusion matrix produced by CRISTAL classification and computing Type I and Type II errors. A Type I error occurs when CRISTAL incorrectly classifies a *covered* review as *not covered*, while a Type II error occurs when CRISTAL wrongly classifies a *not covered* review as *covered*. Note that, unlike the *review coverage* computation, when assessing the CRISTAL classification accuracy we also consider reviews that AR-MINER classified as *non-informative*. This was needed to take into account possible *informative* reviews wrongly discarded by CRISTAL that are actually linked to at least one issue/commit. Indeed, three reviews classified as *non-informative* and discarded by CRISTAL are actually present in the manually built oracle, *i.e.,* they are *covered*.

Table 8: Confusion matrix of reviews classified by CRISTAL as covered/not covered.

Oracle \ CRISTAL	covered	not covered
covered	51	23
not covered	7	5,610

The confusion matrix reported in Table 8 shows that (i) the number of reviews that are covered (*i.e.,* linked to at least one commit/issue) in the oracle (74) and that are classified as such by CRISTAL are 51, (ii) the number of reviews that are covered in the oracle and that are classified as not covered by CRISTAL (*i.e.,* Type I errors) is 23, (iii) the number of reviews that are not covered in the oracle (5,617) and that are classified as such by CRISTAL are 5,610, and (iv) the number of reviews that are not covered in the oracle and that are classified as covered by CRISTAL (*i.e.,* Type II errors) is 7. Thus, out of 5,691 reviews, 5,661 were correctly classified. However, while the percentage of Type II errors is very low (<0.01%), when applying CRISTAL to identify covered reviews we must consider that we may miss around 31% of true positive links (*i.e.,* the percentage of correctly classified covered reviews is 69%).

To address $\mathbf{RQ}_2$, we correlated the *review coverage* of 100 apps with the increment/decrement of the overall rating of the apps between the *previous release*, *i.e.,* the one to which the reviews were referring to, and the *current release*, *i.e.,* the one (not) implementing the reviews. To have a reliable overall rating for both releases, we ensure that all 100 apps had at least 100 reviews for each of the two releases we studied. After having collected all the data, we computed the Spearman rank correlation [83] between the *review coverage* of apps and the

increment/decrement of the average rating (from now on $avgRat_{change}$) assigned by users to the *current release* with respect to the *previous release*. The Spearman rank correlation is a non-parametric measure of statistical dependence between the ranking of two variables, represented in our case by the review coverage and by the $avgRat_{change}$. We interpret the correlation coefficient according to the guidelines by Cohen *et al.* [11]: no correlation when $0 \leq |\rho| < 0.1$, small correlation when $0.1 \leq |\rho| < 0.3$, medium correlation when $0.3 \leq |\rho| < 0.5$, and strong correlation when $0.5 \leq |\rho| \leq 1$. We also grouped the 100 apps based on the percentage of informative reviews they implemented (*i.e.,* the *coverage level*) to better observe any possible correlation. In particular, given $Quart_1$ and $Quart_3$ the first and the third quartile of the distribution of *coverage level* for all the apps, we grouped them into the following three sets: *high coverage level* (*coverage level* $> Quart_3$), *medium coverage level* ($Quart_3 \geq$ *coverage level* $> Quart_1$), and *low coverage level* (*coverage level* $\leq Quart_1$).

We also analyzed the boxplots of the distribution of $avgRat_{change}$ by grouping the apps into the three categories described above (*i.e., high coverage level*, *medium coverage level*, and *low coverage level*). In addition to the boxplots, we performed a pairwise comparison of $avgRat_{change}$ for the three groups of apps by using the Mann-Whitney test [12], a non-parametric test of the null hypothesis that it is equally likely that a randomly selected value from one distribution (in our case, the distribution of $avgRat_{change}$ for a specific category of apps) will be less than or greater than a randomly selected value from a second distribution. We reject the null hypothesis for $\alpha < 0.05$. Since we performed multiple tests, we adjusted our p-values using the Holm's correction procedure [35]. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n-1$, and so on. We also estimated the magnitude of the difference between the $avgRat_{change}$ for different groups of apps by using the Cliff's Delta (d) [30], a non-parametric effect size measure. We follow guidelines by Cliff [30] to interpret the effect size values: negligible for $|d| < 0.14$, small for $0.14 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$ and large for $|d| \geq 0.474$.

4.3.2. Surveying Android App Developers

We designed a survey aimed at collecting developers' opinions needed to answer our research questions. The study questions are reported in Table 9. For each question, the table specifies whether it is expected a Boolean answer (Yes or No), an open answer, or an answer on a specific scale.

The first five questions (**Q1-Q5**) aim at gathering information about the background of the developers participating in our study. We focus on their experience in mobile development (*i.e.,* number of years of experience, used mobile platforms, and number of apps developed) and on the success of their development activity (*i.e.,* number of downloads and average rating assigned by the users to their apps).

Table 9: Survey questionnaire filled in by the study participants.

Question	Answer
Questions about the developer's background	
1. How many years of experience do you have in mobile development?	Open answer
2. On which mobile platforms have you developed? (*e.g.,* Android, iOS, BlackBerry, etc.)	Open answer
3. How many apps have you developed?	Open answer
4. How many times have been downloaded your apps?	Open answer
5. What is the average rating assigned by users to your apps?	1 | 2 | 3 | 4 | 5
RQ_2-related questions	
6. How frequently do you use the information (*i.e.,* feedbacks, requests, *etc.*) in user reviews to gather/define requirements for a new release of your app?	Very often|Often|Sometimes|Rarely|Never
7. If answer to Q6 $\neq$ Never, what kind of information do you look for in user reviews?	Open answer
8. If answer to Q6 $\neq$ Never, how do you determine whether or not a review has to be taken into account?	Open answer
9. From your experience, which percentage of the user reviews contains useful information for planning the next development steps for your app?	< 25% | 25% to 50% | 50% to 75% |> 75%
RQ_3-related questions	
10. Please describe an example of change applied to your app as a consequence of considering user reviews that, in your understanding, helped to improve the app's quality.	Open answer
11. In your experience, did you observe an increase of your app ratings likely due to user requests you implemented?	YES | NO

Then, we asked developers about their opinion on the crowdsourcing of requirements from user reviews (RQ_2-related questions). We first asked developers how frequently they use information from user reviews to gather and define requirements while working on the next release of their app (**Q6**). Then, to all respondents but those answering "Never" to **Q6**, we further asked which specific information they look for in user reviews (**Q7**), and how they identify the useful reviews among all those left by users (**Q8**). Question nine (**Q9**) asks about the percentage of the reviews that developers consider useful for evolving their apps. We decided to include such a question to gather insights aimed at confirming/contradicting data reported in the literature showing that on average only $\sim$35% of reviews are informative [9].

To address RQ_3, we asked developers to provide examples of actual changes, crowdsourced from user reviews, that in their opinion helped to improve the app's overall quality (**Q10**). Finally, we asked to what extent they observed an increase of their app ratings as a consequence of addressing requests present in user reviews (**Q11**).

The survey was distributed using the *eSurveyPro*[1] Web-based tool. We asked the participants to complete the survey in a period no longer than 30 days (note that *eSurveyPro* allows one to complete the survey in multiple rounds, saving a partially completed survey). After 30

[1] `http://www.esurveyspro.com`

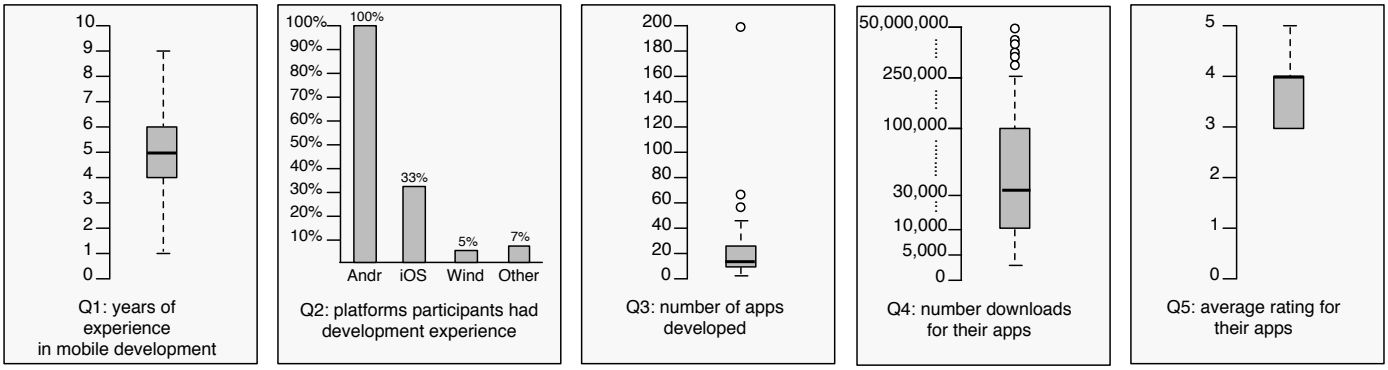

Figure 3: Developers' background.

Table 10: Number of Answers per Question.

Question	#Participants	%Participants
Q1	73	100%
Q2	73	100%
Q3	73	100%
Q4	73	100%
Q5	73	100%
Q6	73	100%
Q7	73	100%
Q8	40	55%
Q9	73	100%
Q10	43	59%
Q11	73	100%

days, we collected developers' answers in a spreadsheet in order to perform data analysis. As previously said, we received 73 questionnaires out of the 2,567 invitations made. Note that none of our open questions were compulsory to avoid high abandonment rate. We report in Table 10 the number of answers collected for each question.

We firstly analyzed the answers provided to the questions related to the developers' background (questions from 1 to 5 in Table 9) by using descriptive statistics and box plots.

The results of this analysis provided us with information about the context in which our study has been performed. Then, we answered our research questions by reporting the percentage of developers answering each of the possible nominal values for **Q6** and **Q11**. For the open questions, we first categorized them using an open coding process [54, 13]. The categorization was performed by two authors, one of which performed a first coding, and the other performed a second coding replicating the activity performed by the first one. Cases of inconsistent/diverging classification were discussed in order to converge to a common categorization and/or if needed split/merge categories.

Participants' Background. Figure 3 summarizes the background information about the 73 study participants. They claimed, on average, 4.8 years of experience in mobile software development (median=5), with a minimum of one year and a maximum of nine (see Figure 3-**Q1**). By construction, all of them worked on the Android platform, one-third on iOS, and a small percentage on Windows Mobile (5%) and on other platforms (7%) like Symbian, Maemo, and Palm Pilot (see Figure 3-**Q2**). The involved developers implemented, on average, a relatively high number of mobile apps (18, median=12), with a maximum of 200 and a minimum of one (Figure 3-**Q3**). Their apps have been downloaded between 250 and over 50 millions times, with a median of 30,000. The average rating assigned to such apps is quite high (mean=3.75, median=4) indicating a good success in the apps' stores. It is worth noting that the average rating of the analyzed apps is similar to the average rating of 5,848 free and 5,848 paid apps analyzed previously by Bavota *et al.* [6].

Overall, the experience of the 73 respondents is quite high, both in terms of years working on mobile platforms (especially considering that they represent a relatively recent technology), as well as in terms of the number of developed apps. Also, their apps have been downloaded thousands of times (Figure 3-**Q4**) and most of them also received good user ratings (median four stars, see Figure 3-**Q5**).

5. Study Results

This section answers $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$ by discussing the results achieved in both our mining study performed on 100 Android apps, and the survey involving 73 professional app developers.

5.1. RQ_2: *To what extent do developers fulfill reviews when working on a new app release?*

Figure 4 reports the percentage of informative reviews implemented by the developers of the considered 100 apps. Among these 100 apps, only two reviews were traced to a previously performed commit, *i.e.,* they were likely to be related to something already fixed, and thus discarded by the `Link Identifier`. In particular, the x-axis represents

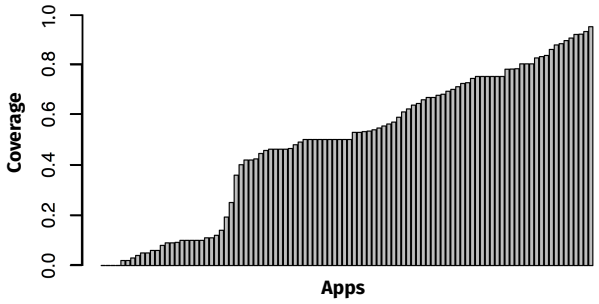

Figure 4: **RQ_2:** *review coverage* of the 100 apps. A coverage (y-axis) equals to 1.0 means that all the informative reviews were considered in the analyzed release.

the 100 apps (*i.e.,* each bar represents one app) while the *y*-axis reports their coverage level (*i.e.,* the percentage of informative reviews that they implemented). The results suggest that most of the developers carefully take into account user reviews when working on the new release of their app. On one hand, among the 100 apps, on average 49% of the informative reviews (of which 64% are negative) are implemented by developers in the new app release. Moreover, 28 apps implemented suggestions from more than 74% of parts of the informative reviews. On the other hand, we also found 27 apps implementing less than 25% of informative user reviews. Overall, we observed a first quartile of 18%, a median of 50% and a third quartile of 73%. As examples of interesting cases, we found that developers of the SMS Backup+ app considered 61 informative reviews received in release 1.5.0 by covering all of them in release 1.5.1; and Autostarts' developers implemented only five informative reviews out of 24 received for the release 1.0.9.1.

For what concerns results of the survey, the participants confirmed to strongly rely on user reviews to gather/define requirements for the new releases of their apps (**Q6**). Indeed, 49% *very often* exploit the information in user reviews, 38% *often*, 11% *sometimes*, and 1% *rarely*. None of the respondents answered *never* to **Q6**. These results represent a first indication of the importance of user reviews for mobile app developers, and enforce and support our previous findings.

Concerning **Q7**, *i.e.,* what type of information the developers look for in user reviews, we categorized the developers' open answers based on their content. For example, an answer reporting "*Mostly for errors*" clearly indicates that the developer looks for "Bugs reporting" information in the user reviews. Table 11 reports the results of such a categorization. The answers suggest that developers are interested in a very specific set of information (only five categories defined across the 73 answers). Their main focus is toward information concerning *bug reporting* (75% of respondents) and *suggestions for new features* (68%). The provided open answers highlighted very interesting practices followed by developers when looking for these types

Table 11: Q7: Information developers look for in user reviews.

Kind Of Information	#Participants	%Participants
Generic (functional) bug reporting	55	75%
Generic suggestions for new features	50	68%
Energy consumption issues	8	11%
Suggestions for GUI Improvement	6	8%
Positive feedback	1	1%

of information. Three respondents explained that bug reporting information is much more useful when the app is new and just uploaded in the app store. Subsequently, suggestions for new features become more useful. A representative answer explaining this concept is reported in the following:

> *It depends on the maturity of the app. At the beginning we look a lot for strange behaviors of the app described in user reviews. When the app becomes stable, we mainly look for suggestions to further improve it (e.g., new features, changes in the UI to make it more usable).*

This insight can be taken into account to develop recommendation systems aimed at automatically prioritizing requirements embedded in the app user reviews (as done, at least in part by AR-Miner [9]).

While the importance of addressing user reviews will also be the focus of **Q11** (and in general, of RQ_3), some developers already highlighted the positive impact on the customer satisfaction of implementing features recommended by the users while answering **Q7**:

> *Very useful are suggestions for new features, they improve the customer satisfaction. Also bug fix requests help us!*

One developer also explicitly highlighted the strong role that features suggested by the users of her apps play during release planning activities:

> *I look for bugs and suggestions for new features. New features suggested by users often influence my release planning.*

Finally, it is interesting to report one last answer related to *bugs reporting* and *suggestions for new features* and highlighting how often, it is hard for developers to gather useful feedbacks from their users:

> *I look for suggestions for new features. Bugs reporting is more difficult because often the users simply write that the app "doesn't work".*

This supports the need for recommendation systems automatically inferring requirements (including bug fix requests) as the one recently proposed by Panichella *et al.* [71].

A lower percentage of respondents (11%) indicated that they look at user reviews for recommendations on how to

Table 12: Q8: Criteria considered to identify useful reviews.

Criterion	#Participants	%Participants
Constructiveness, discard emotional reviews	12	32%
Usefulness [Generic]	7	18%
Level of detail	5	12%
Complexity of the required change	5	12%
Criticality of the change	4	10%
None [Consider all reviews]	3	8%
Score of the review	2	5%
Target audience	2	5%

improve the app's GUI (*e.g.,* "*bugs related to the GUI*", "*GUI suggestions*"). Six of them (8%) claimed their interest for user reviews indicating problems related to the energy consumption of the app, while one developer explained the central role the user reviews play for her emotional support:

> *I look mainly for positive feedback, to recharge myself emotionally.*

Question **Q8** investigates how developers determine whether or not a user review must be taken into account while working on the next release of their app. As previously observed, the insights gathered from answers to **Q8** might be used to support the automatic prioritization of user reviews and/or to automatically discard useless reviews [9]. Note that, as reported in Table 10, only 40 out of the 73 interviewed developers answered this question. As previously done for **Q7**, also in this case we categorized the open answers provided by respondents, in order to get an overview of the criteria they followed to discern useful from useless reviews. Table 12 reports the results of such a classification.

Most of the developers who provided an answer to **Q8** take into account the constructiveness of the review to judge if considering or not its content for the apps evolution. Note that two different types of answers fall into this category: Those explicitly talking about the constructiveness of the reviews, as well as those indicating long reviews as the ones to take into account. We interpret long reviews as those unlikely to contain poor emotional contents (*e.g.,* "*Hate this app*"). An example of provided answer is the following one:

> *When it does not convey information about app usability or functionality (e.g., awesome, crappy etc.), I would not bother looking at it. "The app is slow...", "cannot do...", etc. are useful comments.*

> *By the constructiveness of the review, such as the situation in which the bug occur, or concrete suggestions for new features. I cannot do anything for reviews like "Does not work!").*

The second category of answers we got was targeted as "Usefulness [Generic]". Here we include all answers that simply refer to the usefulness of the review without providing any insight on how they interpret/judge the review's usefulness: *e.g.,* "*Looking at the content searching for something interesting*", "*It depends on the content of the message in the review*".

Table 13: Percentage of useful reviews from developers' point of view.

Percentage	#Participants	%Participants
$< 25\%$	55	75%
25% to 50%	16	22%
50% to 75%	2	3%
$> 75\%$	0	0%

Five developers (12%) reported the level of detail of the review as the main criterion they use to assess its usefulness: *e.g.,* "*Clarity in exposing a thought is a good starting point*", "*how detailed is the review*", "*If the message is clear and well explains the problem a user has*".

Five respondents focused, instead, their attention on the complexity of the required change. While, unfortunately, none of them explicitly stated if they look for complex or for simple changes, from the provided answers it seems that the more complex the change recommended in the review, the lower the likelihood that it will be taken into account by the developers: *e.g.,* "*Time and cost required for implementing the change*".

Four respondents indicated the criticality of the review's content as a proxy for its importance. When talking about the criticality of a review r_i, we really refer to the number of reviews *sharing the same concerns/containing the same recommendations* of r_i, *e.g.,* "*If there are a lots of requests about the same topic*".

Three respondents claimed to consider all reviews as important (*i.e.,* no criteria used to identify the useful ones). Surprisingly, only two referred to the review's score as an indicator of the urgency of implementing the changes required in the review:

> *Generally I spend most of time on negative reviews. Those are the ones on which I focus to improve my app.*

Finally, two developers try to implement requests in reviews likely coming from their target audience: "*I estimate if the reviewer is part of the target audience*".

Table 13 summarizes the answers provided to **Q9**, asking an opinion on the percentage of user reviews that actually contain useful information. Most of the participants agreed on the small percentage of useful user reviews: 55 of them (75%) believe that less than 25% of user reviews contain useful information, 16 (22%) between 25% and 50%, two (3%) between 50% and 75%, while no one kept the $> 75\%$ option. These results (i) highlight the need for recommendation systems able to automatically detect informative reviews (as the AR-Miner tool by Chen *et al.* [9]), and (ii) support findings in the literature showing as only $\sim 35\%$ of reviews contain useful information [9] (in

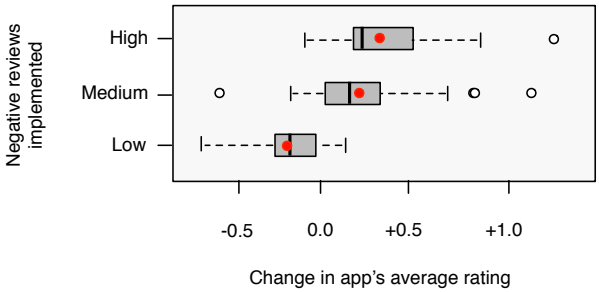

Figure 5: **RQ$_3$:** Boxplots of $avgRat_{change}$ for apps having different *coverage levels*. The red dot indicates the mean.

our mining study we found 29% of informative reviews among the analyzed 5,691).

Answer to RQ$_2$. Results obtained by mining 100 Android apps using CRISTAL show that in most cases developers carefully take into account user reviews when working on the new release of their app. Indeed, on average 49% of the informative suggestions contained in user reviews are implemented by developers in the new app release. The survey respondents confirmed such findings, claiming that they often rely on the information in user reviews when planning a new release of their app (49% very often, 38% often). They mainly look in user reviews for bug reporting (75%) and suggestions for new features (68%). Also, they identify useful reviews by searching for the constructive and detailed ones (44%), containing more than just emotional expressions. The vast majority of the survey respondents (75%) indicate that just a small percentage of user reviews is informative (< 25%).

5.2. *RQ$_3$: What is the effect of a crowd review mechanism (for planning and implementing future changes) on the app success?*

When analyzing the change of the app's rating (64 of the 100 analyzed apps increased their rating between a release and the subsequent observed one), we found a strong positive Spearman's rank correlation (ρ =0.59, p-value<0.01) between the apps' *coverage level* and the change of average score between the old and the new app release ($avgRat_{change}$). This indicates that the higher the *coverage level*, the higher the $avgRat_{change}$, *i.e.,* apps implementing more informative reviews increase more their average score in their new release. Figure 5 shows the $avgRat_{change}$ distributions for apps with *low*, *medium*, and *high coverage level*. Figure 5 confirms that apps implementing a higher percentage of reviews are rewarded by their users with a higher positive $avgRat_{change}$. Indeed, apps implementing *low* percentage of reviews obtain, on average, a -0.21 $avgRat_{change}$, *i.e.,* their average rating for the new release is lower than for the previous one. Instead, apps having a *medium* and a *high coverage level* achieve, on average, a 0.20 and a 0.33 $avgRat_{change}$, respectively.

Table 14: **RQ$_3$:** $avgRat_{change}$ for apps having different *coverage levels*: Mann-Whitney test (adj. p-value) and Cliff's Delta (d).

Test	adj. p-value	d
high level *vs* medium level	<0.001	0.82 (Large)
high level *vs* low level	<0.001	0.91 (Large)
medium level *vs* low level	0.047	0.24 (Small)

Table 14 reports the results of the Mann-Whitney test (adjusted p-value) and the Cliff's d effect size. We compared each set of apps (grouped by *coverage level*) with all other sets having a lower *coverage level* (*e.g., high level vs.* the others). Table 14 shows that apps implementing a higher number of reviews always exhibit a statistically significantly higher increment of their average score than apps having a lower percentage of reviews implemented (p-value always < 0.05). The Cliff's d is always large, except for the comparison between apps having a *medium level* and those having *low level*, where the effect size is medium.

Thus, the quantitative analysis performed to answer **RQ$_3$** provides us with empirical evidence that developers of Android apps implementing a higher percentage of informative user reviews are rewarded by users with higher ratings for their new release. Although we are aware that this is not sufficient to claim causation, we performed a qualitative analysis to (at least in part) find a rationale for the relation that we quantitatively observed.

The most direct way to find some practical evidence for our findings is analyzing comments left on Google Play by the same user for the two releases considered in our study for each of the 100 apps (*i.e., previous* and *current* releases). Specifically, we checked whether there were cases in which (i) a user complained about some issues experienced in release r_{k-1} of an app, hence grading the app with a low score, (ii) developers performed a change to solve the issue in release r_k, and (iii) the same user positively reviewed release r_k, hence acknowledging the fixes. In our study, we found 29 such cases. While this number might appear low, it must be clear that it may or may not happen that users positively (re)comment on an app after their complaints were addressed. Having said that, it is interesting to note that in all of the 29 cases the score given by the user on the previous release (r_{k-1}) increased in the new (fixed) release (r_k). The average increment was of 3.7 stars (median=4). For instance, a user of AnySoftKeyboard complained about release 74 of such an app, grading it with a one-star score: *"you cannot change the print location with the left and right arrows, a normal delete button is missing, the back space button for delete is not friendly"*. After the app was updated in release 75, the same user assigned a five-star score: *"Love the keyboard, fixed the problems"*. Another example is a user of Tinfoil for Facebook, assigning a two-star score to release 4.3, and after upgrading release 4.4 assigning a five-stars score, commenting that *"the update fixed all my gripes, great app"*. On the other hand, we found eight

cases in which (i) a user experienced an issue in release r_{k-1} of an app, grading the app with a low score, (ii) developers did not perform changes in order to fix such an issue, and (iii) the same user negatively reviewed release r_k, hence pointing out the missed issue resolution. An interesting case involves a user of FBReader, who assigned a two-star score to release 1.7.3 complaining that *"the app is crashing when I try to scroll to previous page. Please rectify it ASAP. I will improve my rating"*. In the next release of the app, the same user did not modify the rating and she commented that *"I tried to update the app to see improvements... Nevermind, I'm removing it from my phone"*. As a final highlight, it is interesting to report the increase/decrease in an average rating obtained by the two apps cited in the context of $\mathbf{RQ_3}$. SMS Backup+, implementing 100% of the 61 informative reviews for release 1.5.0 received an increase of +0.86 in the overall score for the release 1.5.1. Instead, the Autostarts app, implementing only 20% of the 24 negative reviews received on release 1.0.9.1, obtained a decrease of -0.79 on its overall score for the release 1.0.9.2.

The answers provided by the survey's participants to **Q10** and **Q11** confirm the importance of considering user reviews while evolving mobile apps. When answering **Q10**, developers described examples of useful reviews that they actually implemented in their apps. The reported answers go from very small and cosmetic changes (*e.g.,* "*I modified some icons*") to much more complex fixes/additions to the apps, including:

- **Bug fixes:**
  - *We were able to spot a bug that prevented iPhone's users to access the photo library*;
  - *I reworked a workaround for an android bug (related to multi-touch input) thanks to feedbacks from reviews. The problem only manifested on some models, and so I was not aware of the problem*;
  - *Once we had a bug that was only visible while using the app on the Samsung S4. This device was not part of the ones we used for testing. We were able to fix it thanks to the information in the review made by users of the S4*;
  - *Fixed a bug that only occurred in a certain GPS location.*

  Interestingly, the last three examples highlight the difficulties experienced by developers during the testing of their apps due to (i) the wide variety of mobile devices available on the market, and (ii) the different environments in which their apps will be used (the latter problem is typical of any real software system), and (iii) the Android API bug proneness [45, 6].

- **Improvements to the GUI:**
  - *I fixed some graphical problems occurring on old devices*;
  - *Completely reimplemented the GUI.*

- **Implementations of new features:**
  - *In one of my apps one of the most selling feature was a user's suggestion*;
  - *I integrated social features in my app (Facebook, Twitter) since required in several user reviews.*

  The first answer highlights the high potential of suggestions contained in user reviews to increase the app's success while the second one shows the influence that a high number of user requests can have on the release planning of mobile apps.

- **Improvements of non-functional requirements:**
  - *I implemented a new UI to fix energy problems*;
  - *I improved the app's usability.*

  Both answers highlight the need for tools supporting the design of GUIs with non-functional requirements (*i.e.,* quality attributes) considerations such as energy consumption [46] and usability.

Finally, 90% of the surveyed developers (*i.e.,* 66 out of 73) observed an increase of their app ratings as a consequence of user requests they implement (**Q11**). This supports our $\mathbf{RQ_3}$'s finding from the mining study: *Developers of apps implementing user reviews are rewarded in terms of ratings.*

Answer to $\mathbf{RQ_3}$. Developers of Android apps implementing user reviews are rewarded in terms of ratings. Also, our qualitative analysis supports, at least in part, our quantitative findings: the surveyed developers confirmed that the small set of informative reviews represents a very precious source of information, leading to fixing of bugs difficult to catch during testing activities, implementing new successful features, and improving non-functional requirements. The vast majority of the respondents (90%) believes that implementing requests from user reviews has a positive effect on the app's success (as assessed by the user ratings).

6. Threats to Validity

Regarding *construct validity* (relationship between theory and observation), one threat is due to how we built the oracle needed for assessing CRISTAL's traceability precision and recall. Although the evaluators are the authors of this paper, we limited the bias by (i) employing in each pair one author who did not know all the details of the approach beforehand, (ii) building the oracle before producing (and knowing) the traces, and (iii) following a clearly-defined evaluation procedure. Such a procedure is

also intended to mitigate imprecision and incompleteness in the oracle, although cannot completely avoid it.

The CRISTAL approach itself could suffer from intrinsic imprecisions of other approaches that it relies upon, such as AR-Miner [9] and ReLink [82], for which we reported performances from the original work. While there may be additional opportunities for increasing linking accuracy by relying on automatically generated commit messages [14, 44], however, this is outside the scope of this paper; hence, we leave this exploration for future work.

Threats to *internal validity* concern internal factors that could have influenced our observations, especially for the relationship between the coverage of reviews and the increase of the ratings. Clearly, a rating increase could be due to many other possible factors [6, 45, 79], such as a very important feature added in the new release, regardless of the feedback. However, this paper aims at providing a quantitative correlation (as it was also done in previous work, where the use of fault- and change- prone APIs was related to apps' lack of success [6, 45]), rather than showing a cause-effect relationship. Also, we found some clear evidence of "rewarding" by mining and discussing cases where the same user positively reviewed the new release of the app after providing a lower score on a buggy one. Finally, answers collected in the conducted survey helped in further corroborating our findings.

Our findings showed that on average 49% of informative reviews are implemented by developers. Clearly, we cannot exclude that a change implemented in an app (e.g., a new feature) that we were able to link to an app's review (i.e., the new feature was recommended in the review) was already planned by the app's developers, independently from what was suggested in the review. In other words, we cannot claim a direct relationship between the requests made in users' reviews and the changes implemented in the app, even when we are able to identify traceability links between reviews and changes.

As for *external validity* (*i.e.,* the generalizability of our findings), the accuracy and completeness of CRISTAL ($\mathbf{RQ}_1$) has been evaluated on ten apps, due to the need for manually building the oracle. Nevertheless, we found links for a total of 5,691 reviews (1,649 were classified as informative) towards 12,307 commits and 682 issues. Clearly, the traceability performance we observed could substantially vary for other apps. This is because, as described in Section 2, CRISTAL relies on the presence of consistent lexicon either between user reviews and terms used in the source code, in commit messages or, at least, between user reviews and terms used to name app's widgets. For some apps, there can be a strong mismatch between user reviews and source code/commit messages/widget terms, therefore negatively affecting the traceability accuracy.

As for the main study ($\mathbf{RQ}_2$ and $\mathbf{RQ}_3$) the evaluation is much larger (100 apps) and diversified enough in terms of apps' size and categories. While we could not directly apply diversity or other sampling metrics in our work [52, 62], since the metrics are not directly available for our Android dataset, we aimed at diversifying the set of studied apps as much as possible. Still, our results might not generalize to other open source apps and, even more, to commercial apps.

Another threat to the external validity is that the sample of surveyed developers may not be statistically representative of the Android developers community. However, as showed in Figure 3, our sample of 73 participants is diverse in terms of programming experience.

7. Related Work

In this section we describe previous work on analyzing crowdsourced requirements in mobile apps for building traceability links between informal documentation and source code. The differences between CRISTAL and the related work are summarized in Table 15. A comprehensive overview of crowd-sourcing approaches in software engineering is available elsewhere [50].

7.1. Analyzing Crowdsourced Requirements In Apps

Although CRISTAL is the first approach aimed at analyzing and monitoring the impact of crowdsourced requirements in the development of mobile apps, previous work has analyzed the topics and content of app store reviews [9, 25, 32, 38, 43, 66], the correlation between ratings, price, and downloads [34], and the correlation between reviews and ratings [66]. McIlroy *et al.* [56] reported a case study in which they observed increasing ratings for the apps where developers systematically answer to user reviews. Our empirical investigation is complementary to this work since it reports that the implementation of feature requested by the users have positive effects in terms of ratings. The same authors also defined an approach able to automatically labeling the types of issues raised by the users in the user reviews [55].

Iacob and Harrison [38] provided empirical evidence of the extent users of mobile apps rely on app store reviews to describe feature requests, and the topics that represent the requests. Among 3,279 reviews manually analyzed, 763 (23%) expressed feature requests. While CRISTAL also requires a labeled set, it uses a semi-supervised learning-based approach to classify reviews as informative and non-informative [9] instead of relying on linguistic rules.

Martin *et al.* [53] mined 3,187 releases of mobile apps in order to study the characteristics of the most impacting releases from users' perspective. They found that 40% of target releases impacted performance in the Google store, while 55% of them impacted performance in the Windows store. Moreover, they observed that more release notes that include more features descriptions than bug fixes can increase the chance for a release to be impacting, and to improve the final rating assigned by users.

Ruiz *et al.* [76] conducted an empirical study with 10,000 mobile apps in order to investigate whether the

Table 15: Summary of the related work. For each paper we report (i) the supported task (Extract Topics, Studying Rating, Identify Informative Reviews, Rank Reviews, Detect Inconsistencies, Extract Feature Requests, Linking, Monitoring, Recommendation, Reviews Categorization); (ii) the approach (Topic Models, Static Analysis, Qualitative Analysis, STatistical Analysis, Frequent Itemsets Mining, Linguistic Rules, Semi Supervised Learning, Mining Software Repositories, Lightweight Textual Analysis, Information Retrieval, Natural Language Processing, SEntiment Analysis); (iii) documentation used as the source of the analysis/link; documentation used as the target of the link (Api Elements, Source Code Elements, Source Code Changes); (iv) and repository of the documents used as source (Google Play, Apple App Store, BlackBerry app story, Stack Overlfow, GitHub, Eclipse Project, Lucene Project, Jira Issue Trackers of Open Source Projects, Mailing Lists).

Approach	Task	Approach	#Docs used as source	Target	Repository
Ciurumelea *et al.* [10]	RC	LTA+IR	7,754 app store reviews	–	GP
Maalej *et al.* [49]	RC	STA+NLP	32,210 app store reviews	–	GP
Panichella *et al.* [71]	RC	NLP+SEA	7,696 app store reviews	–	GP
Di Sorbo *et al.* [20]	RC	NLP+SEA	7,696 app store reviews	–	GP
Palomba *et al.* [67]	EFR+L	NLP	44,683 app store reviews	SCE	GP
Villaroel *et al.* [80]	RR	NLP	1,763 app store reviews	–	GP
Chen *et al.* [9]	ET+IIR+RR	STA+TM	691,097 app store reviews	–	GP
Fu *et al.* [25]	DI+QA+FT	STA+TM	13 Million app store reviews	–	GP
Harman *et al.* [34]	SR	STA	300 apps	–	BB
Iacob and Harrison [38]	EFR+ET	LR+TM	136,998 app store reviews	–	GP
Martin *et al.* [53]	EFR+ET	LR+TM	8,000 app store reviews	–	GP
Guzman and Maalej [32]	EFR+ET	LR+TM+SEA	32,210 app store reviews	–	GP+AAS
Panichella *et al.* [71]	EFR+ET	SSL+LTA+LR	32,210 app store reviews	–	GP+AAS
Khalid *et al.* [43]	ET	QA	6,390 app store reviews	–	AAS
Pagano and Maalej [66]	EFR	FIM+SA	1,126,453 app store reviews	–	AAS
Galvis and Winbladh [8]	ET	TP+SA	2,651 app store reviews	–	GP
McIlroy *et al.* [56]	M	MSR	10,713 mobile apps	–	GP
McIlroy *et al.* [55]	RC	TM	601,221 app store reviews	–	GP
Ruiz *et al.* [76]	DI	MSR	10,000 mobile apps	–	GP
Khalid *et al.* [42]	M	MSR	10,000 mobile apps	–	GP
Bacchelli *et al.* [4]	L	LTA+IR	101,149 emails	SCE	ML
Panichella *et al.* [70]	L	LTA+IR	18,046 emails + 30,486 bug reports	SCE	EP+LP
Parnin *et al.* [72]	L	LTA	307,774 discussions	AE	SO
Rigby and Robillard [75]	L	LTA	188 answer posts	AE	SO
Subramanian *et al.* [77]	L	SA	4000 code snippets	SCE	SO+GH
Thung *et al.* [78]	R	MSR	507 feature requests	AE	JITOSP
CRISTAL	IIR+L+M	SSL+LTA	19,224 app store reviews	SCC	GP

rating system used by mobile app stores is able to capture the changing user satisfaction levels, finding that such system is not dynamic enough to capture the user satisfaction. Khalid *et al.* [42] examined the relationship between the app ratings and the static analysis warnings collected using FindBugs. They found that specific categories of warnings, such as *Bad Practice*, *Internationalization*, and *Performance* are significantly more related to low-rated apps.

Previous work have also focused on categorizing reviews. Similarly to Iacob and Harrison [38], Pagano and Maalej [66] analyzed reviews in the Apple App Store and found that about 33% of the reviews were related to requirements and user experience. In addition, they also found that reviews related to recommendations, helpfulness, and feature information are accompanied by higher ratings; while reviews with lower ratings express dissuasion, dispraise, and are mostly bug reports. The results of our second study (Section 5.2) are complementary to the work by Pagano and Maalej [66], as we analyzed the impact of crowdsourcing requirements on the apps' success. Chen *et al.* [9] proposed an approach (*i.e.,*AR-Miner) for filtering and ranking informative reviews automatically. Informative reviews are identified by using a semi-supervised learning-based approach. Then the reviews are grouped into topics to measure the importance of each group of informative reviews. On average, 35% of the reviews were labeled as informative by AR-Miner. CRISTAL relies on AR-Miner for detecting informative reviews (see Figure 1). Khalid *et al.* [43] conducted a qualitative study on 6,390 user reviews of free iOS apps and qualitatively classified them into 12 kinds of complaints. Their study suggested that over 45% of the complaints are related to problems developers can address, and that they can be useful to prioritize quality assurance tasks.

Sentiment analysis is also a trend in analysis of user reviews. For instance, Guzman and Maalej [32] proposed an automatic approach for filtering, aggregating, and analyzing user reviews by using natural language processing techniques in order to identify fine-grained app features in the reviews. Such reviews are then accompanied with a general score given by the analysis of user sentiments about

the identified features. Panichella *et al.* [71] presented a taxonomy to classify app reviews into categories that are relevant for software maintenance and evolution. They used an approach that combines (i) natural language processing, (ii) textual analysis, and (iii) sentiment analysis for the automatic classification of app reviews, with a precision of 75% and a recall of 74%, into the categories identified. Also Maalej *et al.* [49] defined a set of probabilistic techniques based on review metadata, text classification, natural language processing, and sentiment analysis for the automatic classification of app reviews. They found that the combination of text classification and natural language processing techniques is particularly efficient when classifying reviews. Ciurumelea *et al.* [10] firstly proposed a low-level taxonomy describing categories of problems usually appearing in mobile apps (*e.g.,* performance or battery drains), and then they built an approach, based on Information Retrieval techniques, able to classify user reviews according to the defined taxonomy. Another work analyzing sentiments and opinions is SUR-Miner by Gu and Kim [31] that summarizes sentiments and opinions and clusters them in five categories: aspect evaluation, bug reports, feature requests, praise and others. In the same category, Di Sorbo *et al.* [20] devised an approach able to summarize user reviews in order to generate a structured agenda of software changes developers should apply on their applications. CRISTAL does not use sentiment analysis, however, future work will be devoted to prioritizing/categorizing user reviews by including sentiment analysis in the CRISTAL pipeline.

Other approaches related to user reviews/apps description mining are more concerned on extracting representative topics and keywords from the reviews/descriptions. For example, Vu *et al.* proposed MARK [81], a tool which extracts representative keywords from user reviews that can be used by developers for filtering reviews. Carreno and Winbladh [8] used the Aspect and Sentiment Unification Model to extract the main topics in app store reviews and the sentences representative of those topics. The main difference between CRISTAL and the ASUM-based approach is that CRISTAL's goal is to link reviews to changes instead of extracting main topics from app store reviews. In addition, Gorla *et al.* [29] proposed a model aimed at validating whether the implemented behavior of Android apps matches the advertised behavior in the app' description. Fu *et al.* [25] analyzed reviews at three different levels: (i) inconsistency of comments, (ii) reasons for liking or disliking an app, and (iii) user preferences and concerns. Given a sample of 50,000 reviews, they found that 0.9% were inconsistent with the ratings. Moreover, reasons for problematic apps are mainly related to functional features, performance issues, cost and compatibility. Villarroel *et al.* [80] devised an approach to classify the user reviews on the basis of the information they contain (*i.e.,* useless, suggestion for new features, bugs reporting). Palomba *et al.* [67] presented a technique able to link relevant user feedback extracted from user reviews onto source code elements. The studies by Linares-Vásquez *et al.* [45], Bavota *et al.* [6], and Tian *et al.* [79] analyzed ratings and user reviews to identify factors impacting the success of Android apps.

Finally, it is worth mentioning the existence of several commercial tools, such as *Apteligent*[2], *App Annie*[3], and *App Figures*[4] which offer analytics systems which support the decision activities of app developers. These commercial tools for "app analytics" collect real-time data from real usages in terms of devices, crashes at runtime, downloads, and app performance measures (*e.g.,* network latency, app load time), however, no commercial tool is able to (i) automatically categorize informative/non-informative reviews, (ii) monitor reviews implementation, or (iii) link reviews to changes as CRISTAL does.

7.2. Linking Informal Documentation to Source Code

Several approaches have been proposed for tracing informal documentation (*i.e.,* feature descriptions, emails, forums, *etc.*) onto source code or other artifacts [4, 17, 21, 47, 70, 72, 74, 84]. Bacchelli *et al.* [4] used lightweight textual grep-based analysis and IR techniques to link emails to source code elements. Parnin *et al.* [72] built traceability links between Stack Overflow (SO) threads (*i.e.,* questions and answers) and API classes to measure the coverage of APIs in SO discussions. Linares-Vásquez *et al.* [47] linked Stack Overflow questions to Android APIs to identify how developers react to API changes. Zhang and Hou [84] identified problematic APIs by analyzing negative sentences in forum discussions. Panichella *et al.* [70] proposed an heuristic-based approach for linking methods description in informal documentation such as emails or bug descriptions to API elements. The approach includes the following steps: (i) identifying fully-qualified class names in the text of emails/bug reports, (ii) extracting paragraphs from the text, and (iii) tracing such paragraphs to specific methods of the classes, computing the textual similarity between the paragraphs and method signatures. Gethers *et al.* [27] used Relational Topic Models (RTM) for linking source code to high level artifacts (HLA) and measuring to what extent the topics in HLAs have been covered in the source code. Differently from the approaches mentioned above, CRISTAL is based on textual analysis rather than on more sophisticated API-matching or techniques such as RTM because (i) user reviews are at a different level of abstraction than SO discussion or development emails, and (ii) a rather small corpus of reviews and commit notes may not be sufficient for effectively applying techniques like RTM. Rigby and Robillard [75] identified salient (*i.e.,* essential) API elements in informal documentation by using island grammars [59] and code contexts [16]. Subramanian *et al.* [77] extracted and analyzed incomplete ASTs

[2]https://www.apteligent.com
[3]https://www.appannie.com
[4]https://appfigures.com

from code snippets in API documentation to identify API types and methods referenced in the snippets.

Thung *et al.* [78] mined historical changes to recommend methods that are relevant to incoming feature requests. Instead of using oracles or island grammars for linking informal documentation to source code, CRISTAL uses bug reports and commit notes as a bridge between reviews and source code changes. Also, although the purpose of CRISTAL is not to recommend API elements that are relevant to incoming crowdsourced requirements, it traces user reviews to source code changes in order to monitor whether those requirements are considered and implemented by the developers.

7.3. Crowdsourcing Requirements

Crowdsourcing mechanisms have been previously used not only in the context of mobile development, but also for requirements engineering. Dumitru *et al.* [22] proposed a recommender which is able to suggest product features for a specific domain; the recommendations are based on common variants and cross-category features per domain mined from product descriptions contained in publicly available online specifications. Huffman *et al.* [37] presented a method for semi-automatically recovering domain specific pre-requirements information from mental models of diverse stakeholders of a system; therefore, the stakeholders' understanding of generic systems or domains (*e.g.,* a generic word processor) can be used for recovering traceability links or measuring the obsolescence of a system. Hu and Liu [36] proposed an approach for mining user reviews (posted in online shopping websites) and extracting features on which the customers have expressed positive and negative opinions.

7.4. Limitations of the existing work. Why is CRISTAL *needed?*

As summarized above, there is a large body of work aimed at (i) mining user reviews from app stores, and (ii) linking informal textual documentation to source code. Also, recently—though after CRISTAL was firstly published [68]— McIlroy *et al.* [56] have analyzed whether addressing user reviews increases app ratings by manually categorizing user reviews and mining changes to ratings and reviews. In this paper we analyze not only whether addressing user reviews increases app ratings, but also the thoughts and experiences of 73 mobile app developers concerning user reviews, what do they look for on reviews, what is the volume of informative reviews, among other interesting questions (See section 4.3.2). Moreover, to the best of our knowledge, there is no approach aimed at tracing links (i) between a user review and a change addressing it; and (ii) towards a new user review where the user's rating is possibly changed in view of the implemented changes. CRISTAL was designed to cover the aforementioned functionality, which is not even available in the commercial tools for app analytics.

8. Conclusion and Future Work

This paper presents an in-depth analysis into what extent app development teams can exploit crowdsourcing mechanisms for planning future changes and how these changes impact user satisfaction as measured by follow-up ratings. We devised an approach, named CRISTAL, aimed at detecting traceability links between app store reviews and code changes likely addressing them. Using such links it is possible to determine which informative reviews are addressed and what is the effect of a crowd review mechanism (for planning and implementing future changes) on the app success. Data collected by mining 100 mobile apps and their respective reviews showed that (i) on average, apps' developers implement 49% of informative user reviews while working on the new app release, and (ii) user reviews really matter for the app's success, because fulfilling a high percentage of informative reviews is usually followed by an increase in the ratings for the new release of that app. In addition, a survey performed with 73 apps developers confirmed that developers consider user reviews while evolving their apps, and in particular those with details about bugs or new features. In addition, the participants claimed that they observed a correlation between suggested changes in user reviews and an improvement in user ratings.

Both survey and mining-based study with CRISTAL confirm the importance of crowdsourcing user reviews for mobile developers, and the benefit of considering the user reviews. CRISTAL is the first step to help mobile developers to monitor user reviews and their implementation. Combining CRISTAL with approaches existing in the literature to categorize and prioritize users' reviews (*e.g.,* as in the work of Villarroel *et al.* [80]) can provide developers with a complete platform supporting the release planning of mobile apps.

While a lot has been already done by researchers to support mobile apps' developers in building successful apps, we believe more is yet to come. Two promising research directions that we plan to pursue are (i) the automatic identification of the killer features that a specific type of app (*e.g.,* a GPS navigator) must implement to succeed in the market, and (ii) the automatic creation of test cases able to reproduce bugs reported in users' reviews.

Acknowledgments

This work is supported in part by the NSF CCF-1525902, CCF-1253837 and CCF-1218129 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. 2002. Recovering traceability links between code

and documentation. *IEEE Transactions on Software Engineering 28,* 10, 970–983.

[2] Apple. 2016. Apple app store. `https://itunes.apple.com/`.

[3] Atlassian. 2016. Jira. `https://www.atlassian.com/software/jira`.

[4] Bacchelli, A., Lanza, M., and Robbes, R. 2010. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 375–384.

[5] Baeza-Yates, R. and Ribeiro-Neto, B. 1999. *Modern Information Retrieval*. Addison-Wesley.

[6] Bavota, G., Linares-Vásquez, M., Bernal-Cárdenas, C., Penta, M. D., Oliveto, R., and Poshyvanyk, D. 2015. The impact of API change- and fault-proneness on the user ratings of android apps. *IEEE Trans. Software Eng. 41,* 4, 384–407.

[7] Canfora, G., Di Penta, M., Oliveto, R., and Panichella, S. 2012. Who is going to mentor newcomers in open source projects? In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. ACM, 44.

[8] Carreno, L. V. G. and Winbladh, K. 2013. Analysis of user comments: An approach for software requirements evolution. In *35th International Conference on Software Engineering (ICSE'13)*. 582–591.

[9] Chen, N., Lin, J., Hoi, S., Xiao, X., and Zhang, B. 2014. AR-Miner: Mining informative reviews for developers from mobile app marketplace. In *36th International Conference on Software Engineering (ICSE'14)*. 767–778.

[10] Ciurumelea, A., Schaufelbühl, A., Panichella, S., and Gall, H. C. 2017. Analyzing reviews and code of mobile apps for better release planning. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 91–102.

[11] Cohen, J. 1988. *Statistical power analysis for the behavioral sciences* 2nd Ed. Lawrence Earlbaum Associates.

[12] Conover, W. J. 1998. *Practical Nonparametric Statistics* 3rd Edition Ed. Wiley.

[13] Corbin, J. and Strauss, A. 1990. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology 13,* 1, 3–21.

[14] Cortés-Coy, L., Linares-Vásquez, M., Aponte, J., and Poshyvanyk, D. 2014. On automatically generating commit messages via summarization of source code changes. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 275–284.

[15] Cover, T. and Thomas, J. 1991. *Elements of Information Theory*. Wiley-Interscience.

[16] Dagenais, B. and Robillard, M. 2012. Recovering traceability links between an API and its learning resources. In *34th International Conference on Software Engineering (ICSE'12)*. 47–57.

[17] Dasgupta, T., Grechanik, M., Moritz, E., Dit, B., and Poshyvanyk, D. 2013. Enhancing software traceability by automatically expanding corpora with relevant documentation. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. 320–329.

[18] De Lucia, A., Marcus, A., Oliveto, R., and Poshyvanyk, D. 2012. *Information retrieval methods for automated traceability recovery*. Vol. 9781447122395. Springer-Verlag London Ltd, 71–98.

[19] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science 41,* 6, 391–407.

[20] Di Sorbo, A., Panichella, S., Alexandru, C. V., Shimagaki, J., Visaggio, C. A., Canfora, G., and Gall, H. C. 2016. What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. ACM, New York, NY, USA, 499–510.

[21] Dit, B., Moritz, E., Linares-Vásquez, M., and Poshyvanyk, D. 2013. Supporting and accelerating reproducible research in software maintenance using tracelab component library. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 330–339.

[22] Dumitru, H., Gibiec, M., Hariri, N., Cleland-Huang, J., Mobasher, B., Castro-Herrera, C., and Mirakhordi, M. 2011. On-demand feature recommendations derived from mining public product descriptions. In *33rd IEEE/ACM International Conference on Software Engineering (ICSE'11)*. 181–190.

[23] Fischer, M., Pinzger, M., and Gall, H. 2003. Populating a release history database from version control and bug tracking systems. In *19th International Conference on Software Maintenance (ICSM 2003), 22-26 September 2003, Amsterdam, The Netherlands*. 23–.

[24] Foundation, M. 2016. Bugzilla. `https://www.bugzilla.org`.

[25] Fu, B., Lin, J., Li, L., Faloutsos, C., Hong, J., and Sadeh, N. 2013. Why people hate your app: Making sense of user feedback in a mobile app store. In *19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1276–1284.

[26] Gethers, M., Oliveto, R., Poshyvanyk, D., and Lucia, A. D. 2011a. On integrating orthogonal information retrieval methods to improve traceability recovery. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. 133–142.

[27] Gethers, M., Savage, T., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. 2011b. Codetopics: Which topic am i coding now? In *33rd IEEE/ACM International Conference on Software Engineering (ICSE'11),Formal Research Tool Demonstration*. 1034–1036.

[28] Google. 2016. Google play market. `https://play.google.com`.

[29] Gorla, A., Tavecchia, I., Gross, F., and Zeller, A. 2014. Checking app behavior against app descriptions. In *International Conference on Software Engineering (ICSE'14)*.

[30] Grissom, R. and Kim, J. 2005. *Effect sizes for research: A broad practical approach* 2nd Edition Ed. Lawrence Earlbaum Associates.

[31] Gu, X. and Kim, S. 2015. What parts of your apps are loved by users? In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. to appear.

[32] Guzman, E. and Maalej, W. 2014. How do users like this feature? a fine grained sentiment analysis of app reviews. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*. 153–162.

[33] Hall, M., Frank, R., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. 2009. The WEKA data mining software: an update. *SIGKDD Explorations 11,* 1, 10–18.

[34] Harman, M., Jia, Y., and Zhang, Y. 2012. App store mining and analysis: MSR for app stores. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*. IEEE, 108–111.

[35] Holm, S. 1979. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal on Statistics 6*, 65–70.

[36] Hu, M. and Liu, B. 2004. Mining and summarizing customer reviews. In *10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 168–177.

[37] Huffman-Hayes, J., Antoniol, G., and Guéhéneuc, Y. 2008. Prereqir: Recovering pre-requirements via cluster analysis. In *Working Conference on Reverse Engineering (WCRE'08)*.

[38] Iacob, C. and Harrison, R. 2013. Retrieving and analyzing mobile apps feature requests from online reviews. In *10th Working Conference on Mining Software Repositories (MSR'13)*. 41–44.

[39] Jaccard, P. 1901. Etude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin de la Société Vaudoise des Sciences Naturelles* 37.

[40] Jones, N. 2013. Seven best practices for optimizing mobile testing efforts. Tech. Rep. G00248240, Gartner. February.

[41] Khalid, H., Nagappan, M., and Hassan, A. 2015a. Examining the relationship between findbugs warnings and end user ratings: A case study on 10,000 android apps. *Software, IEEE PP,* 99, 1–1.

[42] Khalid, H., Nagappan, M., and Hassan, A. 2015b. Examining the relationship between findbugs warnings and end user ratings: A case study on 10,000 android apps. *IEEE Software PP,* 99, 1–1.

[43] Khalid, H., Shihab, E., Nagappan, M., and Hassan, A. E. 2014. What do mobile App users complain about? a study on free iOS Apps. *IEEE Software* 2-3, 103–134.

[44] Le, T.-D., Linares-Vasquez, M., Lo, D., and Poshyvanyk, D. 2015. Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*. 36–47.

[45] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., and Poshyvanyk, D. 2013. API change and fault proneness: a threat to the success of Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 477–487.

[46] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D. 2015a. Optimizing energy consumption of guis in android apps: A multi-objective approach. In *10th Joint Meeting of the European Software Engineering Conference and the 23rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*. 143–154.

[47] Linares-Vásquez, M., Bavota, G., Di Penta, M., Oliveto, R., and Poshyvanyk, D. 2014. How do api changes trigger stack overflow discussions? a study on the android sdk. In *Proceedings of the 22Nd International Conference on Program Comprehension*. ICPC 2014. ACM, New York, NY, USA, 83–94.

[48] Linares-Vásquez, M., Vendome, C., Luo, Q., and Poshyvanyk, D. 2015b. How developers detect and fix performance bottlenecks in android apps. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*. 352–361.

[49] Maalej, W., Kurtanović, Z., Nabil, H., and Stanik, C. 2016. On the automatic classification of app reviews. *Requir. Eng. 21,* 3, 311–331.

[50] Mao, K., Capra, L., Harman, M., and Jia, Y. 2015. A survey of the use of crowdsourcing in software engineering. *Research Note*.

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

[52] Martin, W., Harman, M., Jia, Y., Sarro, F., and Zhang, Y. 2015. The app sampling problem for app store mining. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR '15. IEEE Press, Piscataway, NJ, USA, 123–133.

[53] Martin, W., Sarro, F., and Harman, M. 2016. Causal impact analysis applied to app releases in google play and windows phone store. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE'16, Seattle, USA, 2016*.

[54] Maxwell, J. A. 1996. *Qualitative research design : an interactive approach*. Sage Publications Thousand Oaks, Calif.

[55] McIlroy, S., Ali, N., Khalid, H., and E. Hassan, A. 2016. Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering 21,* 3, 1067–1106.

[56] McIlroy, S., Shang, W., Ali, N., and Hassan, A. 2015. Is it worth responding to reviews? a case study of the top free apps in the google play store. In *IEEE Software*.

[57] Microsoft. 2016. Windows phone app store. `http://www.windowsphone.com/en-us/store/`.

[58] Mojica Ruiz, I., Nagappan, M., Adams, B., Berger, T., Dienst, S., and Hassan, A. 2014. Impact of ad libraries on ratings of android mobile apps. *Software, IEEE 31,* 6, 86–92.

[59] Moonen, L. 2001. Generating robust parsers using island grammars. In *8th IEEE Working Conference on Reverse Engineering (WCRE)*. 13–22.

[60] Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., and Poshyvanyk, D. 2015. Auto-completing bug reports for android applications. In *FSE'15*. 673–686.

[61] Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., and Poshyvanyk, D. 2016. Automatically discovering, reporting and reproducing android application crashes. In *ICST'16*.

[62] Nagappan, M., Zimmermann, T., and Bird, C. 2013. Diversity in software engineering research. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 466–476.

[63] Nayebi, M., Adams, B., and Ruhe, G. 2016. Release practices in mobile apps — users and developers perception. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Osaka, Japan, 552–562.

[64] Nigam, K., McCallum, A., Thurn, S., and Mitchell, T. 2000. Text classification from labeled and unlabeled documents using EM. *Machine Learning 39,* 2-3, 103–134.

[65] Oliveto, R., Gethers, M., Poshyvanyk, D., and De Lucia, A. 2010. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proceedings of the 18th International Conference on Program Comprehension*. IEEE Press, Braga, Portugal.

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

[67] Palomba, F., Salza, P., Ciurumelea, A., Panichella, S., Gall, H., Ferrucci, F., and De Lucia, A. 2017. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the IEEE/ACM International Conference on Software Engineering , ICSE17, Buenos Aires, Argentina, 2017*.

[68] Palomba, F., Vásquez, M. L., Bavota, G., Oliveto, R., Penta, M. D., Poshyvanyk, D., and Lucia, A. D. 2015. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME 2015), Bremen, Germany, 2015*. IEEE, 291–300.

[69] Palomba, G., Linares-Vásquez, M., Bavota, G., Oliveto, R., Di Penta, M., Poshyvanyk, D., and De Lucia, A. 2016. Online appendix of: Crowdsourcing user reviews to support the evolution of mobile apps. `http://www.cs.wm.edu/semeru/data/JSS-Cristal`. Tech. rep. `http://www.cs.wm.edu/semeru/data/JSS-Cristal`.

[70] Panichella, S., Aponte, J., Di Penta, M., Marcus, A., and Canfora, G. 2012. Mining source code descriptions from developer communications. In *IEEE 20th International Conference on Program Comprehension (ICPC'12)*. 63–72.

[71] Panichella, S., Di Sorbo, A., Guzman, E., Visaggio, C. A., Canfora, G., and Gall, H. 2015. How can i improve my app? classifying user reviews for software maintenance and evolution. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME 2015), Bremen, Germany, 2015*. IEEE, 281–290.

[72] Parnin, C., Treude, C., Grammel, L., and Storey, M.-A. 2012. Crowd documentation: Exploring the coverage and dynamics of API discussions on stack overflow. Tech. Rep. GIT-CS-12-05, Georgia Tech.

[73] Porter, M. F. 1980. An algorithm for suffix stripping. *Program 14,* 3, 130–137.

[74] Poshyvanyk, D. and Marcus, D. 2007. Combining formal concept analysis with information retrieval for concept location in source code. In *Proc. of 15th IEEE ICPC*. IEEE CS Press, Banff, Alberta, Canada, 37–48.

[75] Rigby, P. C. and Robillard, M. P. 2013. Discovering essential code elements in informal documentation. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE / ACM, 832–841.

[76] Ruiz, M., Nagappan, M., Adams, B., Berger, T., Dienst, S., and Hassan, A. 2015. An examination of the current rating system used in mobile app stores. In *IEEE Software*.

[77] Subramanian, S., Inozemtseva, L., and Holmes, R. 2014. Live API documentation. In *36th International Conference on Software Engineering (ICSE'14)*.

[78] Thung, F., Shaowei, W., Lo, D., and Lawall, L. 2013. Automatic recommendation of API methods from feature requests. In *28th International Conference on Automated Software Engineering (ASE'13)*. 11–15.

[79] Tian, Y., Nagappan, M., Lo, D., and Hassan, A. E. 2015. What are the characteristics of high-rated apps? a case study on free android applications. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME 2015)*. 301–310.

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

[81] Vu, P. M., Nguyen, T. T., Pham, H. V., and Nguyen, T. T. 2015. Mining user opinions in mobile app reviews: A keyword-based approach. *CoRR abs/1505.04657*.

[82] Wu, R., Zhang, H., Kim, S., and Cheung, S.-C. 2011. ReLink: recovering links between bugs and changes. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, 15–25.

[83] Zar, J. H. 1972. Significance testing of the spearman rank correlation coefficient. *Journal of the American Statistical Association 67,* 339, pp. 578–580.

[84] Zhang, Y. and Hou, D. 2013. Extracting problematic API features from forum discussions. In *21st International Conference on Program Comprehension (ICPC'13)*. 141–151.