User Reviews Matter! Tracking Crowdsourced Reviews to Support Evolution of Successful Apps

Fabio Palomba*, Mario Linares-Vásquez§, Gabriele Bavota†, Rocco Oliveto‡,
Massimiliano Di Penta¶, Denys Poshyvanyk§, Andrea De Lucia*
*University of Salerno, Fisciano (SA), Italy – §The College of William and Mary, VA, USA
†Free University of Bozen-Bolzano, Bolzano (BZ), Italy – ‡University of Molise, Pesche (IS), Italy
¶University of Sannio, Benevento, Italy

***Abstract*—Nowadays software applications, and especially mobile apps, undergo frequent release updates through app stores. 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 show—by performing a study on 100 Android apps—how applications addressing user reviews increase their success in terms of rating. Specifically, we devise 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 indicate that developers implementing user reviews are rewarded in terms of ratings. This poses the need for specialized recommendation systems aimed at analyzing informative crowd reviews and prioritizing feedback to be satisfied in order to increase the apps success.**

I. Introduction

In recent years, software development and release planning activities shifted from a traditional paradigm, in which a software system is periodically released following a road map (shipped to customers/shops or downloaded from the Internet), towards a paradigm in which continuous releases become available for upgrade with a cadence of few weeks, if not days. This phenomenon is particularly evident in —though not limited to—mobile apps, where releases are managed through online app stores, such as the Apple App Store [3], Google Play Market [15], or Windows Phone App Store [1].

In several contexts, and above all in the world of mobile apps, the distribution of updates (*i.e.,* new features and bug fixes) through online markets is accompanied by a mechanism that allows users to rate releases using scores (stars rating) and text reviews. The quantitative part of the mechanism is implemented in a form of scores, usually expressed as a choice of one to five stars. The textual part of the rating mechanism 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. Therefore, app 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 in diverse combinations of devices, screen sizes, operating systems, and network conditions that may not necessarily be reproducible during development/testing activities.

Consequently, by reading reviews and analyzing the ratings, development teams are encouraged to improve their apps, for example, by fixing bugs or by adding commonly requested features. According to a recent Gartner report's recommendation [21], 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's 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 [7], [8], [14], [19], [22], [27], [29].

In this paper we investigate *to 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*. Specifically, we devise an approach named as CRISTAL (**C**rowdsourcing **R**ev**I**ews to **S**uppor**T** **A**pps evo**L**ution) to detect 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. It is worth noting that several approaches have been proposed in the literature for recovering requirements-to-source-code traceability links [2], [26]. However, these methods are not directly applicable in our context. First, as shown by Chen *et al.* [8], not all the reviews can be considered useful and/or informative. Also, unlike issue reports and emails, reviews do not refer to implementation details. In other words, there is a vocabulary mismatch between user reviews and source code or issues reported in issue trackers. In order to address these challenges, CRISTAL includes a multi-step reviews-to-code-changes traceability link recovery approach that firstly identifies informative comments among reviews (based on a recent approach by Chen *et al.* [8]), and then traces crowd reviews onto commit notes and issue reports by exploiting a set of heuristics specifically designed for this traceability task.

We evaluated CRISTAL and the impact of a crowd review mechanism (for planning and implementing future changes) on the app success, with two empirical studies on 100 Android apps. The results of our empirical investigation revealed that a project monitoring mechanism like CRISTAL, which is based on maintaining traceability links between reviews and source

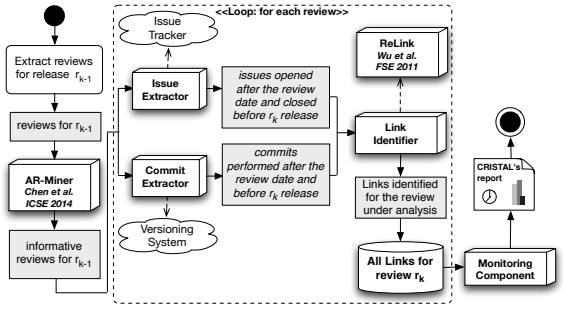


Fig. 1. CRISTAL overview. Solid arrows represent *information flow*, while dotted arrows represent *dependencies*.

code changes addressing them, is useful in the sense that monitoring and implementing user reviews are rewarded by apps' users in terms of ratings.

The paper makes the following noteworthy contributions:

1) CRISTAL*'s reviews-to-code-changes traceability recovery approach*. Albeit being inspired by classic IR-based traceability recovery approaches (*e.g.,* [2], [26]), 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. The proposed monitoring mechanism can be used by developers to monitor project's history by analyzing the relationship between reviews and changes, and thus better supporting release planning activities.
3) *The results of an empirical study conducted on 100 Android apps*. The study exploits CRISTAL to provide quantitative and qualitative evidence on (i) how development teams follow suggestions in informative reviews, and (ii) how users react to those changes.
4) *A comprehensive replication package* [30]. The package includes all the materials used in our studies, and in particular: (i) the traceability oracles manually defined for the ten Android apps; (ii) the listing and URLs of the studied 100 Android apps; (iii) the raw data reporting the links retrieved by CRISTAL on each app; and (iv) the *R* scripts used to generate reported tables and figures.

II. Overview of CRISTAL

CRISTAL follows a three-step process for extracting links between reviews for release r_{k-1} of an app and commits/issues generated while working on release r_k (Fig. 1). The first step aims at collecting user reviews posted for the app release r_{k-1}. These reviews are collected from the app store (*i.e.,* Google Play). However, considering all possible reviews is not an option, since some of them might not contain any informative feedback for the app's developers. For example, a review like "*this app is terrible*" does not provide any insight into specific problems experienced or features demanded by users. Therefore, we rely on AR-Miner by Chen *et al.* [8] to avoid this problem and to classify *informative* and *non-informative* reviews.

In the second step, for each of the collected *informative reviews* ir_j, the Issue Extractor and the Commit Extractor collect the issues and the commits, potentially driven by (*i.e.,* due to) 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, each review ir_j and the issues/commits collected for it in the previous step, are provided to the Link Identifier, which is used to identify possible links between ir_j and issues/commits by using a customized approach based on Information Retrieval (IR) techniques. 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 exploited by the Monitoring Component, which creates reports for managers/developers and shows stats on the reviews that have been implemented. In the following subsections, we provide the details behind each of these major steps.

A. Collecting Reviews

CRISTAL requires the release dates of 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. Note that we intentionally limit our focus to r_k, while some user reviews could be addressed in future releases. This is because (i) we are interested in determining how many reviews are considered as important to be addressed promptly, and (ii) looking for release beyond r_k would make the cause-effect relationship between review and change less likely.

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 due to the fact that 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.,* reviews with low ratings) as well as positive (*i.e.,* reviews with high ratings) user reviews. Indeed, while user complaints are generally described in negative reviews, positive reviews could also provide valuable feedback to developers, like 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.* [8], only some reviews contain information that can directly help developers improve their apps (35.1% on average [8]). Thus, most of reviews posted by the crowd are simply *non-informative* for app's developers. CRISTAL relies on AR-Miner [8] to filter out those *non-informative* reviews. In their evaluation, Chen *et al.* [8] showed that AR-Miner had an accuracy between 76% and 88%.

B. Extracting Issues and Commits

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.

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. Finally, the set of issues/commits extracted during this step are provided, together with the referred review ir_j, to the LINK IDENTIFIER component for detecting traceability links (see Fig. 1).

C. 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. 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 such as: (i) discarding words that do not help to identify apps' features, (ii) considering GUI level terms[1] when performing the linking, and (iii) considering the length difference between reviews and issues/changes.

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 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 [33]). 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 [11] of a given term t in user reviews:

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

where R_t is the set of apps' 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 larger set of 1,000 Android apps considered in a previous study [37]. 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 like *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 completeness of identified links (*i.e.,* recall) by +4% and precision by +2%. The resulting stop word list can be found in our replication package [30].

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):

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

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 coefficient [20], because in most cases user reviews are notably shorter than issue descriptions and, as a consequence, their vocabulary is fairly limited.

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 (*i.e.,* buttons' labels, string literals, *etc.*). 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

[1]For example, reviews have the words window, screen, activity to refer Android GUIs rendered by Android Activities.

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 in the following:

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

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:

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

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}, and found that the bonus 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. We experimented with different values of λ ranging between 0.1 and 1.0 with a step of 0.1. The best results were achieved with $\lambda = 0.6$ (detailed results of the calibration are reported in our online appendix [30]).

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.

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.* [13] and a re-implementation of the RELINK approach proposed by Wu *et al.* [38].

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.

D. 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}. Second, the MONITORING COMPONENT can be exploited 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.

III. EVALUATING CRISTAL'S LINKING ACCURACY

The *goal* of the first study is to investigate to what extent user reviews can be linked to issues/commits by using CRISTAL. In particular, we evaluated CRISTAL's accuracy by measuring precision and recall of the traceability recovery process. The *context* of this study consists of ten Android apps listed in Table I. For each app, the table reports the analyzed release, the size in KLOCs, the number of reviews for the considered release (and in parenthesis the number of informative reviews as detected by AR-MINER), commits, and issues. The choice of this set of ten apps is not completely random; we looked for (i) open source Android apps published

TABLE I
THE TEN APPS CONSIDERED IN THIS STUDY.

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**

TABLE II
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%**

on the Google Play market with versioning system and issue tracker publicly accessible, and (ii) enough diversity in terms of app category (*e.g.,* multimedia, communication), size, and number of issues and commits (see Table I).

A. Research Question and Study Procedure

We aim at answering the following research question:

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

While evaluating the traceability recovery precision simply requires a (manual) validation of the candidate links, evaluating its recall requires the knowledge of all links between user reviews and subsequent issues and commits, some of which might not be identified by CRISTAL. Therefore, to assess CRISTAL in terms of precision and recall, we have manually created an oracle as follows. For each app, the authors independently inspected reviews, issues, and commit logs/messages, in couples (*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 I), 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. Once completing 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 II 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 agreement computed as Jaccard similarity coefficient [20], *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.

After building the *unified* oracle for each app, we used well-known $recall$ and $precision$ [5] metrics to evaluate the recovery accuracy of CRISTAL. 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 F-measure [5].

TABLE III
RQ$_a$**:** RECALL, PRECISION, AND F-MEASURE ACHIEVED BY CRISTAL.

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

B. Analysis of the Results

Table III reports precision, recall, and F-measure achieved by CRISTAL when retrieving links between user reviews and issues/commits on ten apps from our study. The last row of Table III shows the results achieved when considering all 141 links present in our oracles as a single dataset. Note that, for these ten apps, it never happened for a single review to be traced onto a previously performed commit, *i.e.,* that was likely to be related to something already fixed (see Section II-C).

Results in Table III are relatively positive, showing an overall precision of 77% and a recall of 73% (75% F-measure). Also, the precision achieved by CRISTAL is never lower than 60%. 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 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. 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. 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 links present in our oracle and missed by CRISTAL, to understand the reasons behind that. 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 due to the fact that the commit, accompanied by the message "*Added download actions log*", involved among several others 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 results to be high, thus leading to a false positive link. In summary, as any approach based on textual similarity, CRISTAL may fail whenever the presence of common words does not imply similar meaning of the review and the commit note.

Answer to RQ$_a$. 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%.

IV. HOW DEVELOPERS REACT TO USER'S REVIEWS

Once we know the CRISTAL's linking accuracy, the *goal* of the second study is to apply CRISTAL in its typical usage scenario. Specifically, we aim at analyzing to what extent developers use crowdsourced reviews for planning and performing changes to be implemented in the next releases. The *purpose* is to investigate possible gains (if any) for the app's success as maybe reflected in improved ratings/reviews.

A. Research Questions and Study Procedure

In the context of this study we formulate the following RQs:

- **RQ$_c$:** *To what extent do developers fulfill reviews when working on a new app release?* The goal here is to empirically identify whether apps' developers take into account (or not) informative reviews when working on a new app release.
- **RQ$_e$:** *Which is the effect of a crowd review mechanism (for planning and implementing future changes) on the app success?* By addressing **RQ$_e$** we aim at empirically verifying the benefit (if any) of such crowdsourcing activity, in this case measured in terms of apps' scores.

To address **RQ$_c$**, we used the CRISTAL's MONITORING COMPONENT to identify the *review coverage* (*i.e.,* the percentage of informative reviews that are linked to at least one issue/commit–see Section II-D) in 100 Android apps (including the ten apps used in the context of our previous study). 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, as *non-informative* reviews do not provide improvement suggestions for apps.

Before applying CRISTAL to measure reviews coverage, we need to quantify its accuracy 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 **RQ$_a$** as in our previous study. The results are analyzed through the confusion matrix produced by the CRISTAL classification and computing Type I and Type II errors. A Type I error occurs when CRISTAL wrongly classifies a *covered* review as *not covered*, while a Type II error occurs when CRISTAL wrongly classifies a *not covered* review as *covered*.

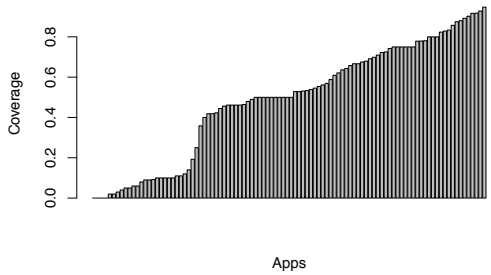
Note that, unlike the *review coverage* computation, when assessing the CRISTAL classification accuracy we also consider reviews that AR-MINER classified as *non-informative*. This was needed to take into account possible *informative* reviews wrongly discarded by CRISTAL that are actually linked to 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*.

The confusion matrix 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) are 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 (*i.e.,* Type II errors) as covered by CRISTAL are 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.

To address $\mathbf{RQ}_e$, 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. Once collected all data, we computed the Spearman rank correlation [39] 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*. We interpret the correlation coefficient according to the guidelines by Cohen *et al.* [9]: 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 Q_1 and Q_3 the first and the third quartile of the distribution of *coverage level* for all apps, we grouped them into the following three sets: *high coverage level* (*coverage level* $> Q_3$); *medium coverage level* ($Q_3 \geq$ *coverage level* $> Q_1$); *low coverage level* (*coverage level* $\leq Q_1$)

We analyzed the boxplots of the distribution of $avgRat_{change}$ by grouping the apps in the three categories described above. In addition to boxplots, we performed a pairwise comparison of $avgRat_{change}$ for the three groups of apps by using the Mann-Whitney test [10] with $\alpha = 0.05$. Since we perform multiple tests, we adjust our p-values using the Holm's correction procedure [18]. We also estimated the magnitude of the difference between the $avgRat_{change}$ for different groups of apps by using the Cliff's Delta (d) [16]. We follow guidelines by Cliff [16] to interpret the effect size values: small for $|d| < 0.33$, medium for $0.33 \leq |d| < 0.474$ and large for $|d| \geq 0.474$.

Fig. 2. $\mathbf{RQ}_e$**:** *review coverage* of the 100 apps.



B. Analysis of the Results

This section discusses the results for RQ_c and RQ_e.

1) Review coverage: Fig. 2 reports the percentage of informative reviews[2] implemented by the developers of the considered 100 apps[3]. The results suggest that most of the developers carefully take into account user reviews when working on the new release of their app. On the 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 more than 74% of 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 (Q_1) of 18%, a median (Q_2) of 50% and a third quartile (Q_3) 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.

Answer to $\mathbf{RQ}_e$. 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 reviews are implemented by developers in the new app release.

2) Benefits of crowdsourcing: When analysing the change of the app's rating (64 of the 100 analysed 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). Fig. 3 shows the $avgRat_{change}$ distributions for apps with *low*, *medium*, and *high coverage level*. Fig. 3 confirms that apps implementing a higher percentage of reviews are rewarded by their users with a higher positive

[2]It is worth noting that 55% of the informative reviews identified by AR-MINER are negative (rating ≤ 3).

[3]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`.

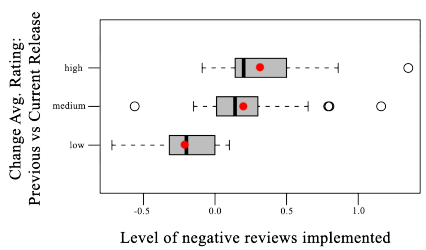


Fig. 3. **RQ$_e$:** Boxplots of $avgRat_{change}$ for apps having different *coverage levels*. The red dot indicates the mean.

TABLE IV
RQ$_e$: $avgRat_{change}$ 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)

$avgRat_{change}$. 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 IV 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 IV 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$_e$** provides us with empirical evidence that developers of Android apps implementing a higher percentage of informative user reviews are rewarded by users with higher rating 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[4] complained about release 74 of such 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-stars score: *Love the keyboard, fixed the problems*. Another example is a user of TINFOIL FOR FACEBOOK[5], 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*. As a final highlight, it is interesting to report the increase/decrease in average rating obtained by the two apps cited in the context of **RQ$_c$**. SMS BACKUP+, implementing 100% of the 61 informative reviews received for release 1.5.0, increased the overall score for release 1.5.1 in +0.86. 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.

Answer to RQ$_e$. Developers of Android apps implementing user reviews are rewarded in terms of ratings. This is confirmed by the observed positive correlation (0.59) between *review coverage* and *change in overall score* between the old and the new app releases. Also, our qualitative analysis supports, at least in part, our quantitative findings.

V. 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 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. Also, the CRISTAL approach itself could suffer from intrinsic imprecisions of other approaches that it relies upon, such as AR-MINER [8] and RELINK [38], for which we reported performances from the original 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, 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], [24]), 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

[4]An on screen keyboard with support for multiple languages.
[5]A wrapper for Facebook's site

the app after providing a lower score on a buggy one. As for *external validity* (*i.e.,* the generalizability of our findings) the accuracy and completeness of CRISTAL ($\mathbf{RQ}_a$) 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. As for the second study ($\mathbf{RQ}_c$ and $\mathbf{RQ}_e$) the evaluation is much larger (100 apps) and diversified enough in terms of apps' size and categories.

VI. 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.

A. 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 [8], [14], [19], [23], [29], the correlation between rating, price, and downloads [17], and the correlation between reviews and ratings [29]. Iacob and Harrison [19] 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. CRISTAL also requires a labeled set, but it uses a semi-supervised learning-based approach to classify reviews as informative and non-informative [8] instead of linguistic rules. Pagano and Malej [29] analyzed reviews in the Apple App Store, and similar to Iacob and Harrison [19], Pagano and Malej 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 features information have the top ratings; while reviews with worst ratings express dissuasion, dispraise, and are mostly bug reports. Results of our second study (Section IV-B2) are complementary to Pagano and Malej [29], as we analyzed the impact of crowdsourcing requirements on the apps' success. Khalid *et al.* [23] 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. Fu *et al.* [14] analyzed reviews at three different levels: (i) inconsistency of comments, (ii) reasons for liking/disliking an app, and (iii) user preferences and concerns. From a sample of 50K reviews, 0.9% were found to be inconsistent with the ratings. Regarding the reasons for problematic apps, they were related to functional features, performance issues, cost and compatibility, among others. Chen *et al.* [8] proposed an approach (i.e., AR-Miner) for filtering and ranking informative reviews automatically. On average, 35% of the reviews were labeled as informative by AR-Miner. CRISTAL relies on AR-Miner for detecting informative reviews (see Fig. 1).

B. Linking Informal Documentation to Code

Several approaches have been proposed for tracing informal documentation (*i.e.,* emails, forums, *etc.*) onto source code or other artifacts. Bacchelli *et al.* [4] used lightweight textual grep-based analysis and IR techniques to link emails to source code elements. Parnin *et al.* [32] 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.* [25] linked Stack Overflow questions to Android APIs to identify how developers react to API changes. Panichella *et al.* [31] proposed an heuristic-based approach for linking methods description in informal documentation such as emails or bug descriptions to API elements. the short corpus of reviews and commit notes is not suitable for techniques like RTM. Rigby and Robillard [34] identified salient (*i.e.,* essential) API elements in informal documentation by using island grammars [28] and code contexts [12]. Subramanian *et al.* [35] extracted and analyzed incomplete ASTs from code snippets in API documentation to identify API types and methods referenced in the snippets. Thung *et al.* [36] 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.

VII. 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 users' 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 which is the effect of a crowd review mechanism (for planning and implementing future changes) on the app success. The achieved results showed that (i) on average, apps' developers implement 49% of informative user's 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. As for the future work, we are planning on replicating the study on different app stores, not necessarily limited to mobile apps, and improving the algorithms for matching reviews onto changes. We are also planning, based on the findings from our study, to build a recommender system for prioritizing user reviews in order to increase the app's success.

REFERENCES

[1] "Windows phone app store. http://www.windowsphone.com/en-us/store/."

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[3] Apple, "Apple app store. https://itunes.apple.com/us/genre/ios/id36?mt=8."

[4] A. Bacchelli, M. Lanza, and R. Robbes, "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*. ACM, 2010, pp. 375–384.

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

[6] G. Bavota, M. Linares-Vásquez, C. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change- and fault-proneness on the user ratings of android apps," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

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

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

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

[10] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

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

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

[13] M. Fischer, M. Pinzger, and H. Gall, "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*, 2003, pp. 23–.

[14] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh, "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*, 2013, pp. 1276–1284.

[15] Google, "Google play market. https://play.google.com."

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

[17] M. Harman, Y. Jia, and Y. Zhang, "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, 2012, pp. 108–111.

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

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

[20] P. Jaccard, "Etude comparative de la distribution florale dans une portion des alpes et des jura," *Bulletin de la Société Vaudoise des Sciences Naturelles*, no. 37, 1901.

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

[22] H. Khalid, M. Nagappan, and A. Hassan, "Examining the relationship between findbugs warnings and end user ratings: A case study on 10,000 android apps," *Software, IEEE*, vol. PP, no. 99, pp. 1–1, 2015.

[23] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile App users complain about? a study on free iOS Apps," *IEEE Software*, no. 2-3, pp. 103–134, 2014.

[24] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "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, 2013, pp. 477–487.

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

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

[27] I. Mojica Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. Hassan, "Impact of ad libraries on ratings of android mobile apps," *Software, IEEE*, vol. 31, no. 6, pp. 86–92, Nov 2014.

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

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

[30] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "Online appendix of: User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," Tech. Rep., http://www.cs.wm.edu/semeru/data/ICSME15-cristal.

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

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

[33] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[34] P. C. Rigby and M. P. Robillard, "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, 2013, pp. 832–841.

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

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

[37] M. L. Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "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, 2013, pp. 477–487.

[38] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "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, 2011, pp. 15–25.

[39] J. H. Zar, "Significance testing of the spearman rank correlation coefficient," *Journal of the American Statistical Association*, vol. 67, no. 339, pp. pp. 578–580, 1972.