Recommending and Localizing Code Changes for Mobile Apps based on User Reviews

Fabio Palomba[1,2], Pasquale Salza[1], Adelina Ciurumelea[3], Sebastiano Panichella[3], Harald Gall[3], Filomena Ferrucci[1], Andrea De Lucia[1]

[1]University of Salerno, Italy — [2]Delft University of Technology, The Netherlands — [3]University of Zurich, Switzerland

***Abstract*—Researchers have proposed several approaches to extract information from user reviews useful for maintaining and evolving mobile apps. However, most of them just perform automatic classification of user reviews according to specific keywords (*e.g.*, bugs, features). Moreover, they do not provide any support for linking user feedback to the source code components to be changed, thus requiring a manual, time-consuming, and error-prone task. In this paper, we introduce CHANGEADVISOR, a novel approach that analyzes the structure, semantics, and sentiments of sentences contained in user reviews to extract useful (user) feedback from maintenance perspectives and recommend to developers changes to software artifacts. It relies on natural language processing and clustering algorithms to group user reviews around similar user needs and suggestions for change. Then, it involves textual based heuristics to determine the code artifacts that need to be maintained according to the recommended software changes. The quantitative and qualitative studies carried out on 44 683 user reviews of 10 open source mobile apps and their original developers showed a high accuracy of CHANGEADVISOR in (i) clustering similar user change requests and (ii) identifying the code components impacted by the suggested changes. Moreover, the obtained results show that CHANGEADVISOR is more accurate than a baseline approach for linking user feedback clusters to the source code in terms of both precision (+47%) and recall (+38%).**

***Index Terms*—Mobile Apps; Mining User Reviews; Natural Language Processing; Impact Analysis**

I. INTRODUCTION

Nowadays, the development and the release planning activities moved from a traditional paradigm, in which software systems are released following a clearly defined road map, towards a paradigm in which continuous releases become available for an upgrade every week [1]–[4]. It is particularly true in the case of mobile apps, where developers manage updates (*i.e.*, new features, enhancements, or bug fixes) through online app stores, such as *Google Play Store*, *Apple Store*, and *Windows Phone App Store* [3], [5].

This kind of distribution is accompanied by mechanisms which allow end users to evaluate releases by using scores, usually expressed as five stars values and user reviews. These reviews are free text that may informally contain relevant information for the development team [6] such as bugs or issues that need to be fixed [5], summaries of the user experience with certain features [7], requests for enhancements [8], ideas for new features [5], [9], and comparison with other apps.

Thus, not only do user reviews represent the simplest and fastest way end users have to express their opinions or report their suggestions, but also a powerful crowd feedback mechanisms that can be used by developers as a backlog for the development process [10], [11], aiming to improve the success/distribution of their apps [5], [12], [13].

The main problem for developers is that existing app distribution platforms provide limited support to systematically filter, classify, and aggregate user feedback to derive requirements [14]. Moreover, manually reading each user review to gather the useful ones is not feasible considering that popular apps (*e.g.*, Facebook) receive hundreds of reviews every day [5], [15]. To address the problem and reducing the manual effort, in the recent past the research community has proposed approaches to select the useful feedback from user reviews [8], [13], [16]–[18], essentially consisting of performing only an automatic classification of the review content according to specific keywords without considering sentence structures and semantics. Moreover, the information that can be gathered is restricted to user reviews [19], and there not exists any systematic way for linking user feedback to the related source code components to change, a task that requires an enormous manual effort and is highly error-prone.

To better illustrate the problem statement, let us consider the following scenario. *John* is a mobile developer of the `FrostWire` app, an open source BitTorrent client for Android, a quite popular app on Google Play Store. *John* frequently checks the numerous user reviews the app receives on a daily basis. Not only do they contain compliments, but also some informative feedback such as bug reports or feature requests. He needs to read reviews carefully and carry out a time-consuming activity of analysis and synthesis in order to identify and collect possible suggestions for improvement. Once *John* detects an informative feedback, he has to locate the source code components related to the requested changes. This is not always easy, even if he is the main developer of the application. Indeed, user reviews are usually informal text written by non-technical users, therefore poorly understandable and/or not containing enough information to match the corresponding source code components correctly. For instance, *John* finds a user review verbatim reporting: *"I can't download most off the songs."*. Besides the spelling mistakes, *John* realizes that the user is complaining about download problems, but he is not sure that this is an actual bug or maybe the user is not able to use his app correctly. Thus, he skips the review and continues to read the other ones. Afterward, he finds other negative user reviews mentioning and detailing the same problem as the one above. As an example, a user claims that he is not able

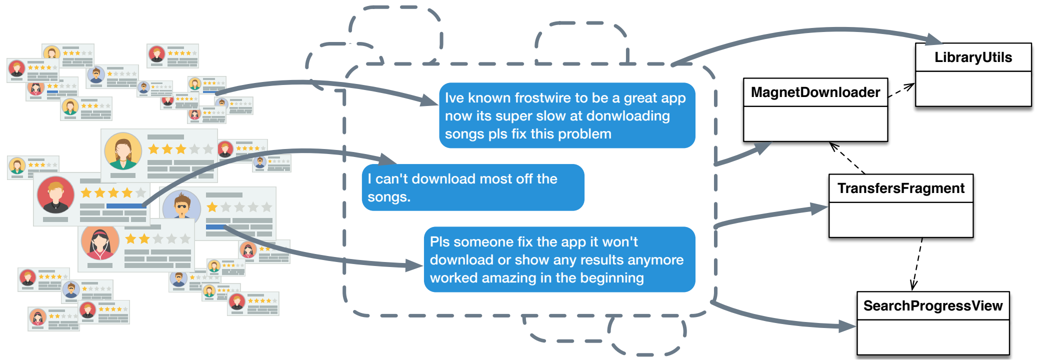
Ive known frostwire to be a great app now its super slow at donwloading songs pls fix this problem
I can't download most off the songs.
Pls someone fix the app it won't download or show any results anymore worked amazing in the beginning
LibraryUtils
MagnetDownloader
TransfersFragment
SearchProgressView

Figure 1: CHANGEADVISOR – Motivating example.

to download songs through magnet links. At this point, *John* understands that this is a real bug by reading the collection of these user reviews and identifies a possible solution.

As an alternative, *John* might have used existing tools. For instance, he might have exploited SURF [18] or CLAP [20] to summarize single reviews or prioritize groups of reviews, respectively. Nevertheless, he would not be able to (i) extract only the useful information hidden behind different user reviews, (ii) group together fine-grained information, (iii) understand the actual impact of each change request.

To overcome these limitations, in this paper we introduce CHANGEADVISOR, a novel approach that facilitates the analysis of the enormous amount of user reviews developers receive on a daily basis, by automatically (i) extracting user feedback relevant from a maintenance perspective, (ii) clustering them based on similar user needs, and (iii) identifying the set of source code components that need to be changed to accommodate the user requests.

By using our approach, the work done by *John* in the previous scenario would have been drastically reduced. Indeed, *John* can download the list of user reviews from the developer console of *Google Play Store* and give them as input to CHANGEADVISOR, together with the source code of his app. Following the steps described in Section III, our approach extracts only user feedback useful from a maintenance perspective and groups entries together into clusters representing similar user needs. Then, CHANGEADVISOR locates the specific source code components that would likely have to be changed. Figure 1 depicts the way CHANGEADVISOR supports the specific task of *John*. In particular, the user feedback is extracted and clustered in a group of sentences which describe a (i) slow download performance and (ii) the inability in showing the transfer results.

Then, CHANGEADVISOR links the grouped user feedback into four classes, *i.e.*, `TransfersFragment`, `SearchProgressView`, `MagnetDownloader`, and `LibraryUtils`. The `TransfersFragment` class is responsible for the visualization of the downloads that are being transferred, and a progress bar (implemented by the `SearchProgressView` class) shows the status of the searches initialized by the user using a magnet link. *John* discovers a connection delay bug in the `LibraryUtils` class which is used by the `MagnetDownloader` class responsible for the download of torrents related to magnet links. Given the output of CHANGEADVISOR, *John* is now able to solve the issue because he immediately identifies the problem experienced by users, and receives suggestions about which source code components need to be modified.

In this paper, we empirically evaluated the performances of CHANGEADVISOR by using 44 683 user reviews and the source code of 10 open source mobile apps. As a preliminary analysis, we quantified the CHANGEADVISOR ability to cluster user feedback toward similar change requests, finding that our approach is able to identify cohesive user feedback clusters. Then, we evaluated CHANGEADVISOR capabilities in identifying the code components impacted by the suggested changes, comparing our technique with a baseline implementation of the tool proposed by Saha *et al.* [21] in the context of bug localization. We observed that our approach is able to achieve 81 % of *precision* and 70 % of *recall*, being 47 % more precise and 38 % more complete than the baseline technique. Finally, we qualitatively evaluated CHANGEADVISOR by surveying the 10 original developers of the apps in our dataset, who confirmed the actual usefulness of the approach in practice. The prototypical implementation of CHANGEADVISOR, as well as the material and working data sets used in our study, are publicly available [22].

Structure of the Paper. Section II discusses the related literature, while Section III presents the proposed approach. In Section IV, we describe the case study conducted to evaluate the proposed approach. The qualitative case study involving the original developers is reported in Section V, whereas Section VI concludes the paper.

II. RELATED WORK

In the following, we summarize the main relevant research in the context of mining user reviews of mobile apps and linking informal textual information to the source code.

A. Mining User Reviews

Harman *et al.* [23] introduced the concept of app store mining by identifying correlations between the customer ratings and the download rank of a mobile app. Iacob and Harrison [8] empirically assessed the extent to which users of mobile apps rely on reviews to describe change requests discovering that a noticeable percentage of user reviews (23 %) describe feature requests. Moreover, Pagano and Malej [5] found that 33 % of the user reviews are related to requirements and user experience, and that developers use the feedback provided by the users to gather requirements. These papers motivated our work since they clearly indicate that users actually exploit the review mechanism in order to suggest improvements to apply to the current version of an app.

Chen *et al.* [24] devised AR-MINER, an approach to filtering and ranking informative reviews using a semi-supervised learning based approach. They demonstrated that, on average, 35 % of reviews contain informative content. Khalid *et al.* [25] reported a study with 6390 user reviews aimed at qualitatively classifying them into 12 types of complaints. The results suggest that more than 45 % of the complaints are related to problems that developers can solve. Such results highlight the importance to have tools supporting developers in evolutionary tasks, in the same way as the one proposed in this paper.

Guzman *et al.* [7] proposed an automatic approach to assigning a score to reviews indicating the user sentiment. Di Sorbo *et al.* [18] devised SURF, a tool to summarize user reviews to gather new requirements. Panichella *et al.* proposed ARDOC [13], an approach that combines natural language processing, sentiment analysis, and text analysis techniques, through a Machine Learning (ML) algorithm to detect sentences in user reviews. CHANGEADVISOR relies on ARDOC to classify user reviews as change requests (*e.g.*, fix a bug or add a new feature). Villarroel *et al.* [20] devised CLAP, an approach to classifying, clustering, and then prioritizing user reviews on the basis of the information they contain (*i.e.*, suggestions for new features or bug reporting). Unlike our work, this approach takes into account entire user reviews rather than the more fine-grained information given by user feedback, and has as a final goal of prioritizing user reviews rather than locate change requests. Finally, Gu and Kim [19] defined an approach able to summarize sentiments and opinions and classify them in aspect evaluation, bug reports, feature requests, praise, and others.

The approaches mentioned above perform an automatic classification of user reviews according to predefined topics (*e.g.*, bugs, features) [19]. Moreover, they do not provide any support for linking user feedback to the source code components to be changed.

B. Linking Informal Textual Documentation to the Source Code

Traceability between textual artifacts (*e.g.*, requirements) and the source code was widely studied in the past (see, *e.g.*, [26], [27]). Similarly, several approaches to locating features in the source code [28], and tracing informal textual documentation, such as e-mails [29], forum discussions [30], [31], and bug reports [21] to the source code have been proposed.

In this context, three works are closer to the one we proposed in this paper. Firstly, Saha *et al.* [21] proposed the use of structured Information Retrieval based on code constructs, *i.e.* class and method names, to improve bug localization. Their approach, named BLUIR, exploits the Vector Space Model [32] to link bug reports to the source code. In Section IV we report the detailed comparison between BLUIR and our approach.

Secondly, Asuncion *et al.* [33] devised TRASE, an approach that uses LDA-based topic modeling to enhance the information provided by prospective traceability. Since there is no way to know a priori the number of latent topics, the authors configured the parameter α of the LDA algorithm (*i.e.*, the number of topics) using different settings, namely $\alpha = 10$, 20, and 30. However, as previously demonstrated by Panichella *et al.* [34], the configuration used to set the clustering algorithm is an important component of topic modeling techniques, and an optimal choice of the parameters generally results in better performance. This is especially true in cases where there are no hints about the right number of clusters to create as for user reviews [34]. Moreover, the technique exploited by TRASE to retrieve links toward the source code requires additional information by developers about the part of the project that needs to be modified, which is not required by our approach.

Finally, Palomba *et al.* [12] proposed CRISTAL, a tool for tracing informative crowdsourced reviews to the source code commits and for monitoring the extent to which developers accommodate user requests and follow-up user reactions as reflected in their ratings. Unlike CRISTAL, the intent of CHANGEADVISOR is to recommend the location of code changes on the current version of an app, rather than monitor the changes already applied during the history of a project.

III. THE CHANGEADVISOR APPROACH

The goal of the proposed approach is to extract from user reviews feedback relevant from a maintenance perspective and suggest the location of such changes in the source code. To this aim, it applies the following steps:

1) user feedback identification and classification (*i.e.*, bug fixing tasks, features enhancement, and new features requests);
2) source code and user feedback preprocessing;
3) user feedback clustering, representing similar user needs (*i.e.*, code change requests);
4) determining the code artifacts related to the suggested software changes.

A. User Feedback Classification

During the first step CHANGEADVISOR extracts and classifies the informative sentences (*i.e.*, the feedback) contained in the user reviews. To achieve this goal it employs ARDOC, a *review classifier* previously defined by Panichella *et al.* [13], which is able to automatically mine feedback in user reviews. Specifically, ARDOC combines Natural Language Processing (NLP), Sentiment Analysis (SA) and Text Analysis (TA) techniques through a Machine Learning (ML) algorithm to detect the user feedback that belongs to one of the following categories: *Information Giving*, *Information Seeking*, *Feature Request* and *Problem Discovery*. We relied on the original implementation of the tool, publicly available as a Java Library [35]. We used this tool to classify and filter feedback in user reviews categorized as (i) request to fix bugs (sentences of the ARDOC category *Problem Discovery*), and (ii) feature enhancements or feature requests (*Feature Request* category). Note that we filtered out feedback in user reviews categorized as *Information Giving* and *Information Seeking* since they are not strictly related to change requests in the source code of an app. As an example, a review reporting *"It's an awesome app!"* is classified as *Information Giving* and does not provide any useful information from a maintenance perspective.

B. Input Preprocessing

In this step CHANGEADVISOR preprocesses both the user feedback extracted from the original reviews and the source code components in order to remove noise contained in the data that may hinder the accuracy of the NLP techniques exploited in further steps.

1) Source Code Preprocessing: The source code is first parsed to extract the code components it contains, such as fields, classes, and methods. The extracted code components are then normalized using a typical Information Retrieval (IR) normalization process [32]. In particular, the terms contained in the source code are transformed by applying the following steps: (i) separating composed identifiers using the camel case splitting which separates words on underscores, capital letters, and numerical digits base; (ii) reducing letters of extracted words to lower case; (iii) removing special characters, programming keywords and common English stop words; (iv) stemming words to their original roots via Porter's stemmer [36]. Finally, the normalized words are weighted using the *term frequency - inverse document frequency* (tf-idf) schema [32], which reduces the relevance of too generic words that are contained in several source code components. This step outputs a set of bag-of-words, one for each class of the analyzed app.

2) User Feedback Preprocessing: The challenge of this step is finding a methodology that is able to "parse" correctly all the words contained in a user feedback, because the language used by the end users of an application is generally informal, very noisy [37], and substantially different from the *sectorial language* used in software artifacts [38]–[40]. For this reason, we defined a specialized IR process to transform the end user language into input suitable for textual analysis using the Python libraries NLTK [41] and TEXTBLOB [42], two collections of tools for Natural Language Processing (NLP) and the spell check PYENCHANT library [43]. The involved NLP steps are:

Spelling correction: the words of user feedback are replaced if misspelled, according to the English vocabulary of PYENCHANT library.

Contractions expansion: it substitutes any possible English contractions with the related extended form (*e.g.*, "don't" becomes "do not").

Nouns and verbs filtering: the process is performed by first applying a part of speech (POS) tagging classification, which identifies the logic role of words in user feedback sentences. Then, only the nouns and verbs are selected for the following steps because they are the most representative parts of the meaning of an artifact [40].

Tokenization: user feedback is transformed into lists of words (*i.e.*, tokens), which will be the atomic part of the next steps, excluding numbers and punctuation that usually do not contain information.

Singularization: the singularization function of the TEXTBLOB library is exploited to normalize every word to its related singular form.

Stopword removal: the tokens are intersected with the WordNet English stopword list [44], which is a list of common words that are frequent in written English (*e.g.*, "the", "a", and "an") and only introduce noise to NLP activities.

Stemming: the inflected words are reduced to their stem form (*e.g.*, "pushing" is replaced with "push"). This step reduces the number of tokens and thus the complexity of the NLP work.

Repetitions removal: for each document, only one occurrence per word is preserved. We perform this step since, in this context, it is unlikely to have more occurrences of a word. Therefore, the contribution provided by having repetitions of the same word is not useful for the NLP activities.

Short tokens removal: this step excludes tokens with less than 3 characters because tokens having a low number of characters are usually conjunctions or generic terms used in informal context [45] and irrelevant for our purposes.

Short documents removal: documents with less than 3 tokens are also excluded from the output because short documents (*i.e.*, user feedback) do not have enough terms to explain a change request clearly.

As the final result, the process outputs a bag-of-words for each user feedback that will be the input of the subsequent clustering phase.

C. User Feedback Clustering

The goal of this step is to group together automatically user feedback expressing similar user needs, consisting of similar code change requests (*e.g.*, a problem in the GUI and/or performance and energy consumption problems). It is worth noting that this step is required since linking single user reviews to source code components does not provide high *precision* values, as highlighted by the preliminary analysis reported in our online appendix [22]. Moreover, this phase allows grouping together common user change requests, making them more comprehensible for the developer. Many effective clustering algorithms [46], [47] accept text objects as input. We experimented three different clustering techniques, such as the Latent Dirichlet Allocation (LDA) exploited by Asuncion *et al.* [33], the application of Genetic Algorithms to LDA (LDA-GA) devised by Panichella *et al.* [34] and the Hierarchical Dirichlet Process (HDP) algorithm proposed by Teh *et al.* [48].

The first technique is the classical implementation of the LDA algorithm [49], which requires the specification of the parameter α beforehand, namely the number of clusters (topics) to create starting from the set of sentences to group together (in our case, the user feedback extracted from the user review). The

LDA-GA technique [34] is an evolution of the LDA algorithm able to find an optimal configuration of the parameter α during their execution by relying on Genetic Algorithms. Finally, HDP [48] is an extension of the LDA algorithm where the number of topics is not known a priori. In HDP and LDA models, each document in the corpus collection is considered as a mixture of latent topics, with each topic being a multinomial distribution over a known vocabulary of words. To cluster related feedback, HDP implements a nonparametric Bayesian approach [50] which iteratively groups elements based on a probability distribution, *i.e.*, the Dirichlet process [48].

We benchmarked the three techniques concerning the quality of solutions and execution time. We decided to integrate HDP in CHANGEADVISOR because it provides a good trade-off between quality and execution time. A detailed analysis of the comparison between the techniques is reported in our online appendix [22].

D. Recommending Source Code Changes

Once the user feedback clusters are formed, grouping together similar change requests, they are subjected to a process of linking to source code components to suggest what is the set of classes to update in order to meet these requests. We linked the user feedback clusters to source code classes by measuring the asymmetric Dice similarity coefficient [32], defined as follows:

$$\text{sim}\left(cluster_j, class_i\right) = \frac{\left| W_{cluster_j} \cap W_{class_i} \right|}{\min\left(\left| W_{cluster_j} \right|, \left| W_{class_i} \right|\right)}$$

where $W_{cluster_j}$ is the set of words contained in the cluster j, W_{class_i} is the set of words contained in class i and the $\min$ function normalizes the similarity score with respect to the number of words contained in the shortest document (*i.e.*, the one containing fewer words) between the cluster and the class under analysis. The asymmetric Dice similarity ranges between $[0, 1]$. We used the asymmetric Dice coefficient instead of other similarity measures (*e.g.* the Jaccard coefficient [51]) because often the user feedback clusters are notably shorter than the source code files and therefore considering the minimum cardinality of the sets of words at the denominator of the formula allows to weight the similarity between documents better. The output is represented by a ranked list where the links having the highest similarity values are reported at the top. Pairs of $(cluster, component)$ having a Dice similarity coefficient higher than a threshold are considered by CHANGEADVISOR to be a "link". We experimented different values for this threshold, and the best results were achieved when considering the third quartile of the distribution of the Dice similarity coefficients obtained for a given application (detailed results of the calibration are in our online appendix [22]). However, the developers may want to order the list based on other criteria, such as the popularity of a change request expressed by the cluster size. In future, we plan to integrate into CHANGEADVISOR a prioritization approach that takes into account both (change) requests popularity and the similarity values with the source code.

Table I: Characteristics of the apps in the dataset.

App	KLOC	Classes	Reviews	Feedback
ACDisplay	47	267	4802	1722
Cool Reader	37	115	4484	1158
FB Reader	104	804	4495	830
Focal	19	81	2642	1285
FrostWire	291	1518	6702	1375
K-9 Mail	108	475	4542	1570
Shortyz Crosswords	22	165	3391	931
SMS Backup +	12	118	4488	1522
Solitaire	10	28	4480	1605
WordPress	110	567	4657	1512
Overall	760	4138	44 683	13 510

IV. STUDY I: THE ACCURACY OF CHANGEADVISOR

The *goal* of the study is to evaluate the effectiveness of CHANGEADVISOR in retrieving links between user feedback clusters and source code. Thus, the *quality focus* is on the accuracy of the proposed approach in suggesting source code changes based on user feedback. The *context* of the study consists of 10 Android open source apps, whose source code is hosted on the *F-Droid* [52] open source repository and also published on the *Google Play Store* [53].

For each app considered in our dataset, Table I shows: (i) the application name, (ii) the size of the app in terms of KLOC and number of classes, (iii) the number of user reviews, and (iv) the number of user feedback entries extracted for the considered user reviews. The selected apps belong to different categories, have different sizes and reviews written by users with different requirements and expectations.

A. Empirical Study Definition and Design

The study aims at answering the following research questions:

- **RQ$_1$:** *Does* CHANGEADVISOR *identify cohesive user feedback clusters representing related change requests?*
- **RQ$_2$:** *Does* CHANGEADVISOR *correctly link change requests represented by user feedback clusters to code artifacts that need to be modified and how well it works compared to a state-of-the-art technique relating informal textual documentation to source code?*

To address our research questions, we firstly ran CHANGEADVISOR against the apps in our dataset. In a normal scenario of using CHANGEADVISOR, the developer owns both the user reviews (by downloading them as CSV file from the app distribution platform) and the source code needed as input. In our case, we first had to retrieve the reviews related to the chosen apps. To this aim, we developed a web scraper that extracts the user reviews directly from the *Google Play Store*, where they are publicly available. For the source code, we downloaded the last version of the considered apps from the corresponding repositories (*e.g.*, *GitHub*) and we used a Java parser to extract single components (see Section III-B).

Before focusing on the performance of the proposed approach in suggesting source code changes (addressed by **RQ$_2$**), with **RQ$_1$** we wanted to ensure that the clusters created by CHANGEADVISOR using HDP are *cohesive* and thus meaningful. To this aim, we involved as inspectors 2 external mobile developers having 5 years of experience. The participants were asked to evaluate the *cohesiveness* of the clusters generated by CHANGEADVISOR for each app in our study. To avoid bias, the inspectors were not aware of the experimental goals and of the specific algorithm used by the approach to cluster user feedback. To express their opinion the participants used a Likert scale intensity from *very low* to *very high* values [54], *i.e.*, giving a value between 1 and 5 (where 1 means *very low*, while 5 *very-high*). This process required approximately 3 weeks of work.

In $\mathbf{RQ_2}$ our goal was to evaluate CHANGEADVISOR accuracy in linking user feedback clusters to the source code. However, as pointed out in Section II, several approaches able to link informal textual documentation to the source code have been proposed in the literature to solve other software engineering tasks (*e.g.*, bug localization). Therefore, rather than simply evaluating the performance of CHANGEADVISOR, we are also interested in understanding whether the definition of our approach is actually needed or similar performance can be achieved by using existing state-of-the-art techniques when used to link user feedback to the source code. To this aim, we used a baseline approach inspired to BLUiR, the technique proposed by Saha *et al.* [21] for linking bug reports to the source code using an Information Retrieval infrastructure. Note that bug reports may be affected by the same noise (*e.g.*, errors, different language compared to the source code programming language) of user reviews. We chose this approach as the baseline because it tries to solve a similar problem as CHANGEADVISOR. As discussed in Section III-C, linking single user reviews to source code components does not provide good results. Hence, to conduct a fair comparison, BLUiR links each of the user feedback cluster produced using HDP to the source code by computing the cosine similarity between the vectors of terms built through the application of the Vector Space Model [32]. The baseline reports a link if the similarity between a cluster and a class is ≥ 0.6.

To compare the two approaches, we needed a set reporting the actual links between user feedback contained in the clusters and the source code. Due to the absence of this data set, we had to build our own oracle manually. We asked the 2 mobile developers previously involved in the evaluation of the *cohesiveness* of user feedback clusters to analyze the change requests contained in the clusters of reviews and determine the software artifacts that need to be maintained for performing such changes. In this case as well, the participants were not aware of the details of the study. We provided the participants with both the source code of the considered apps and the clusters generated by HDP [48]. The task consisted of analyzing each user feedback cluster in order to find the set of classes that need to be modified according to the requests reported in the cluster. Each inspector performed the task independently. Once completed the task, the two different sets of links were compared and the inspectors discussed the differences they found (*e.g.*, links marked as correct by one inspector, but not by the other) in order to resolve the disagreement and reach a common decision. Finally, we considered as *golden set* all the links declared as correct by both the two inspectors after the discussion. To measure the level of agreement between the inspectors, we computed the Jaccard similarity coefficient [51], *i.e.* the number of traceability links identified by both the inspectors over the union of all the links identified by them. The overall agreement between the inspectors was of 76 %. A spreadsheet reporting the data about the agreement computation is available in our replication package [22]. This process required approximately 5 weeks of work.

Table II: Evaluation provided by the inspectors about the cohesiveness of the clusters generated by CHANGEADVISOR.

App	Clusters	Min Cohesiveness	Median	Max Cohesiveness
ACDisplay	6	3	5	5
Cool Reader	8	3	4.5	5
FB Reader	9	3	4	5
Focal	6	4	4	5
FrostWire	7	3	4	5
K-9 Mail	7	4	4.5	5
Shortyz Crosswords	6	3	4	4
SMS Backup +	12	2	3.5	5
Solitaire	8	3	4	5
WordPress	13	3	4	5
Overall	82	2	4	5

Once defined the oracle, we answered $\mathbf{RQ_2}$ by reporting *precision* and *recall* [32] achieved by CHANGEADVISOR and by the baseline.

Replication Package. CHANGEADVISOR is publicly available in the online appendix [22], together with the dataset used in the study. The prototype is a runnable DOCKER container which allows not only the replication of the experiment, but also the application of the approach to any given app data (*i.e.*, reviews and source code).

B. Analysis of the Results

In the following, we discuss the results of the study.

1) Evaluating the Cohesiveness of User Feedback Clusters: Table II reports the minimum, median, and maximum of the scores assigned by the inspector during the evaluation of the clusters generated by CHANGEADVISOR.

As we can see, our approach identified a minimum of 5 and a maximum of 13 user feedback clusters, for a total of 82 clusters on the 10 apps. All the data related to clusters and their validation are available in our online appendix [22]. From a qualitative perspective, the *cohesiveness* of clusters produced was generally evaluated by the inspectors as *high* or *very high* (*i.e.*, 4 and 5 values of the Likert scale, respectively). Indeed, the median of the distribution is 4, while the values are mainly placed between 4 and 5. This result highlights the ability of CHANGEADVISOR to group together correctly similar changes. Nevertheless, there is one specific case for the `SMS Backup +` app where the clustering phase did not provide the expected results. This app offers the possibility to backup SMS, MMS, and calls log entries by providing space in the cloud. Users often complain of energy consumption problems or the lack of compatibility with previous versions of the Android operating

Table III: Comparison between CHANGEADVISOR and the baseline approach in linking change request feedback.

App	CHANGEADVISOR *precision* (%)	CHANGEADVISOR *recall* (%)	Baseline *precision* (%)	Baseline *recall* (%)
ACDisplay	82	68	41	31
Cool Reader	79	68	48	68
FB Reader	83	73	43	17
Focal	83	63	27	50
FrostWire	81	71	29	20
K-9 Mail	84	66	38	69
Shortyz Crosswords	71	63	13	38
SMS Backup +	67	59	38	38
Solitaire	75	60	12	40
WordPress	79	74	38	40
Overall	81	70	34	32

system (*i.e.*, "Android KitKat 4.4"). Manually analyzing the clusters created by CHANGEADVISOR for this application, we discovered that the cluster which obtained the lowest score included user requests related to both these problems. Thus, the presence of co-occurrence of terms in the two different change requests generated some noise that did not allow the correct clustering of the corresponding user feedback.

In Summary. CHANGEADVISOR correctly clusters user feedback expressing similar change requests. An exception occurred because of the co-occurrence of terms expressing different change requests reported in the same user feedback.

2) Evaluating the Linking Accuracy: Table III reports the *precision* and the *recall* achieved by CHANGEADVISOR and by the baseline approach for each of the considered apps when retrieving links between user feedback clusters and source code elements. The last row (*i.e.*, *overall*) shows the results achieved when considering all the links suggested by the two approaches as a single dataset.

As we can see, the results achieved by CHANGEADVISOR are quite positive, since it is able to discover correctly, overall, 81 % of the links between user feedback clusters and classes (*i.e.*, 308 traceability links out of the total 381). At the same time, the suggestions provided by CHANGEADVISOR are quite close to the actual classes that need to be modified to implement a change request. Indeed, the *recall* ranges between 59 % and 74 % (overall is 70 %).

The lowest *precision* of CHANGEADVISOR is achieved on the `SMS Backup +` app, where the proposed approach outputs 5 false positive links. A possible cause behind this result can be related to an incorrect cluster of user requests for this app, as observed before. Besides the mentioned clustering issue, by manually investigating the links suggested by CHANGEADVISOR we discovered two other reasons for this result: (i) poor cohesion of the classes composing the application, and (ii) the poor vocabulary of the identifiers. In particular, we found several cases in which a single class of the project is responsible for multiple functionalities, leading to the co-occurrence of different topics in the same class. As an example, the class `com.zegoggles.smssync.service.state.State` is responsible for both the management of error messages and additional services (*e.g.*, user notifications and the information about the state of the execution of the app). Moreover, the same attributes are used in different methods of the class: for instance, the variable `resource` that is used to update the user interface layout and images contained in the project is referenced in all the methods. As a consequence of these observations, the similarity between user feedback clusters and the source code is blurred by the presence of different topics.

Thus, the main limitation of CHANGEADVISOR is that it is not always possible to generate reliable links from the user feedback when the vocabulary of the source code is poor, or the source code elements implement functionalities related to multiple topics. This result is expected since our approach relies on topic analysis techniques which are themselves affected by the same problem.

On the other hand, it is interesting to observe that CHANGEADVISOR results are highly accurate when the source code is well modularized and has a good vocabulary. For instance, let us consider one of the correct links found on the `Frostwire` project. This app is an open source BitTorrent client and the main problem experienced by users is related to the storage of the downloaded contents. In this example, one of the user feedback cluster identified by CHANGEADVISOR contains two reviews, both of them reporting the inability of users in finding songs they have previously downloaded. CHANGEADVISOR suggests modifying the class `AlbumSongFragment` contained in the package `com.andrew.apollo.ui.fragments.profile` and the classes `GUIUtils` and `GUIMediator` of the package `com.limegroup.gnutella.gui` in order to deal with the problems reported in the user feedback. By analyzing the source code of the suggested classes, we found that our approach correctly suggests the set of classes to change to address the user feedback. Indeed, the class `AlbumSongFragment` implements the methods `onCreateLoader` and `onLoadFinished` that are responsible for loading the multimedia content stored on the mobile phone, whereas the classes `GUIUtils` and `GUIMediator` manage the user interface related to the music library. In this case, the only class missed by CHANGEADVISOR is `MusicUtils` of `com.andrew.apollo.utils`, which contains utility methods used by the class `AlbumSongFragment` to perform the loading of the multimedia contents. This example practically shows the potential usefulness of CHANGEADVISOR from the developers perspective.

As for the baseline approach, it always has low *precision* in retrieving links (*i.e.*, 34 % overall). The poor performances are also highlighted by the value of *recall* which reaches only 32 % overall. We manually investigated some of the links provided by the baseline approach to understanding the reasons behind these results. The major issue is related to the fact that a linking done using the Vector Space Model [32] does not provide optimal performances when relating user feedback clusters and source code. User feedback clusters are almost shorter than the set of words belonging to the source code, and most of the times the words used in the two sets of terms are similar because the users tend to explain problems using the same vocabulary present in the user interface of the app [5]. For this reason, the use of the asymmetric Dice coefficient instead of the cosine similarity results in a winning choice for CHANGEADVISOR because it is able to match directly the strings belonging to the two sets of terms, rather than computing their similarity using vectors.

We also perform an additional analysis aimed at verifying the overlap between the approaches, *i.e.* the number of links correctly retrieved by one approach and missed by the other, as well as the number of links correctly found by both the techniques. A complete report is available in our online appendix [22]. This analysis confirmed the findings discussed above since we observed that 72 % of correct links between user feedback clusters and the source code are retrieved by CHANGEADVISOR only, while both the approaches found 23 % of the correct links. Therefore, only a small percentage of the links (5 %) are retrieved by BLUIR and missed by our approach, highlighting the small contribution given by the alternative approach in finding links between clusters and source code. This result is significant because it shows how existing tools locating informal textual documentation to the source code are not particularly suitable in the context of reviews of mobile applications.

We repeated the experiment by distinguishing the different types of user feedback considered by the approaches (*i.e.*, *Problem discovery* and *Feature request*) and did not find any meaningful differences. For the sake of space limitation, we report these results in the online appendix [22].

In Summary. Despite few cases discussed above, CHANGEADVISOR exhibits high *precision* (81 %) and *recall* (70 %) in linking user feedback clusters to source code components. When compared to the baseline, our approach results are 47 % more precise and 38 % more complete.

C. Threats to Validity

Threats to *construct validity* concern the relationship between the theory and the observation. For the evaluation of the experimented approaches we relied on error-prone human judgment, because of the subjectivity in deciding if a cluster of user feedbacks is cohesive or not and whether a link suggested by an approach is correct or not. To alleviate this issue we built a golden set based on the judgment of more professional inspectors. Moreover, the two inspectors firstly performed the task separately and then all disagreements were discussed and resolved.

The proposed approach itself could have been influenced from intrinsic imprecision of other approaches on which it relies, *e.g.*, ARDOC [13], HDP [48]. In future, we plan to investigate better the relationship between the outcome of CHANGEADVISOR and the use of the techniques discussed above with the goal of increasing the already high performance of our approach.

On the one hand, if the clusters have been qualitatively evaluated by two professional developers in terms of cohesiveness, we cannot speculate about their completeness. Indeed, we cannot ensure that all the user feedback of the apps in our dataset have been taken into account and correctly clustered. However, as future work, we plan to build an oracle reporting the actual clustering of user feedback of the apps in the dataset and compare such an oracle with the results provided by CHANGEADVISOR.

Finally, another threat in this category is related to our implementation of the baseline approach, which was needed to compare our approach with the state-of-the-art. However, we applied the same algorithms and the same steps than the BLUIR approach described by Saha *et al.* [21].

Threats to *internal validity* concern any confounding factors that could influence our results. A first threat is related to the choice of the versions of the apps we considered in our study. Specifically, the analyzed source code refers to the latest version of an app, while the reviews can be related to previous versions. Even though in a normal context a developer is always able to retrieve the reviews for specific versions of his or her app through the *Google Developer Console*, we were not able to distinguish them during the user reviews scraping phase. Nevertheless, this is not an issue of this study since the goal was to quantify the accuracy of the proposed approach in linking user feedback to the source code, independently from the version of the app to which the feedback refers. At the same time, user feedback provided by users could have already been implemented by developers. Although there may be additional opportunities for increasing the practical applicability of CHANGEADVISOR, by combining it with a monitoring system such as CRISTAL [12], this is outside the scope of this paper. Hence, we leave this exploration for future work. Another threat that could affect the validity of the results of **RQ$_2$** is the presence of potential missing links. While the inspectors followed a clearly defined evaluation procedure, we cannot ensure that the set of correct links identified in this phase actually reflect the total set of correct links present in a given application. Finally, the threshold adopted for the Dice similarity coefficients may have influenced our results. However, we chose its value by performing a tuning process, detailed in the online appendix [22].

Threats to *external validity* concern the generalizability of our findings. In this study, we selected 10 different apps belonging to different app categories, having different sizes and user reviews written by different audiences. To confirm the generality of our findings as future work we would have to replicate the study exploiting the data from more apps belonging to different app stores.

V. STUDY II: OPINIONS OF THE ORIGINAL DEVELOPERS

Even though our technique achieved good results in the previous study, software engineering techniques have to address the real needs of developers. For this reason, the *goal* of this second study is to investigate the opinions of original app developers about CHANGEADVISOR, with the *purpose* of analyzing its usefulness from a developer's perspective. The *context* of the study consists of the 10 apps already used in the first study.

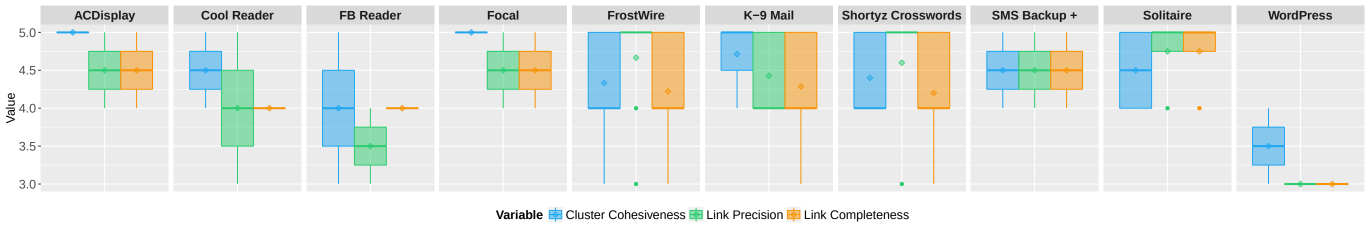

Figure 2: Survey results.

A. Empirical Study Definition and Design

In this study, we address the following research question:

- **RQ$_3$:** *Are the suggestions provided by* CHANGEADVISOR *actually useful for developers?*

To answer our research question, we sent a direct invitation to the developers of the apps taken into account. To identify them, we retrieved the e-mail addresses of the developers from the source code repositories of the considered apps, filtering the developers that have contributed with at least one commit on the main branch. We chose to involve original developers rather than relying on external developers because we wanted to collect the opinions of the developers that actually worked on the systems under analysis and, therefore, have a sound knowledge about the structure of the app. We contacted 102 developers in total, receiving 10 responses, one for each app involved in the study. Note that even though the number of respondents appears to be low (9.8 % response rate), our results are close to the suggested minimum response rate for the survey studies, which is defined around 10 % [55].

The idea behind the study design was to show to each developer a subset of the links found by CHANGEADVISOR for the app he/she developed. For this reason, we selected the top 10 links identified by our approach on each app of the study, those with the higher Dice coefficient resulting from the first study (see Section IV). This was done to avoid having a long questionnaire that might have discouraged developers to take part in our study. For each link composed of a user feedback cluster and the related classes, the participants had to answer the following questions:

1) *Could you please summarize what the users wanted to say in the shown user reviews?*
2) *How well the user reviews are grouped together according to the number of source code components that need to be modified? Please, rate your opinion from 1 = very poorly to 5 = very well.*
3) *How well the proposed set of classes matches the actual set of classes that need to be changed in order to satisfy the user requests? Please, rate your opinion from 1 = very poorly to 5 = very well.*
4) *Evaluate the completeness of the set of classes suggested above compared to the actual set of classes that need to be modified in order to satisfy the user requests. Please, rate your opinion from 1 = unacceptable to 5 = excellent.*

Besides the questions reported above, the developers also filled in a pre-questionnaire that allowed us to assess their background. They also answered a brief post-questionnaire where we asked to estimate how much time they would save using an approach like the one proposed in this paper instead of manually analyzing user reviews to identify change requests.

The survey was designed to be completed within approximately 15 minutes. To automatically collect the answers, the survey was hosted using a web-based survey software, *i.e.*, *Google Forms*. Developers were given 20 days to respond to the survey. At the end of the response period, we collected the answers of the 10 complete questionnaires in a spreadsheet to perform data analysis.

To answer **RQ$_3$**, we computed:

1) the distribution of values assigned by developers when evaluating the *cohesiveness* of user feedback clusters (question #2 of the survey);
2) the distribution of values assigned by developers when assessing the *precision* of CHANGEADVISOR (question #3 of the survey);
3) the distribution of values assigned by developers when evaluating the *completeness* of the suggestions provided by CHANGEADVISOR (question #4 of the survey).

Moreover, when answering to question #1, we collected the descriptions provided by the developers as free opinions about the clusters produced by CHANGEADVISOR.

B. Analysis of the Results

The 10 respondents declared a mobile development experience ranging from 3 to 5 years with 2 to 3 applications each and a *Google Play Store* score of 4 stars. The developers also claimed they look at user reviews "most of the time" to gather requirements for new releases of their apps, spending about 2 hours per week.

Figure 2 shows the boxplots of the distributions of the responses, divided by app, provided by the developers involved in the study. We reported the Likert scale values of the 3 aspects analyzed in **RQ$_3$**, *i.e.*, cluster *cohesiveness*, linking *precision*, and linking *completeness*.

As we can see, the level of *cohesiveness* assigned by the developers is always high (Likert value $\geq$ 4) except for the case of the `WordPress` app where, on average, the cohesiveness on this pass is equal to 3.5. However, by further analyzing his answers we observed that he was always able to summarize the requests contained in the feedback clusters indicating specific problems in the app, therefore explaining well-focused issues (*e.g.*, "*Options in the Dashboard were deactivate for a while and users are just reporting such issues*"). This observation makes us conclude that the lowest values assigned by the developer do not indicate problems in the clustering phase of CHANGEADVISOR, but rather the judgment of the developer has been just more conservative than that of other developers. The results obtained by this analysis confirmed again the ability of CHANGEADVISOR in grouping together similar user needs.

The *precision* and *completeness* of the approach were instead evaluated in general as "well" (4) and "very well" (5), respectively. While the values assigned by developers to assess the *precision* of CHANGEADVISOR somehow reflect the quantitative analysis conducted in Section IV (precision = 81 %), the most interesting thing is related to the *completeness*. Indeed, the developers perceived the links provided by our approach closer to the actual set of code components to modify with respect to what we estimated in our quantitative evaluation (*recall* = 70 %). This highlights the potential of the proposed approach as a tool useful for developers in a real-world environment. The claim is also supported by some of the answers provided by the developers when they filled in the post-questionnaire. In particular, we asked to estimate how much time they would save using CHANGEADVISOR instead of only manually analyzing user reviews. In general, they claimed that a tool like CHANGEADVISOR would be "very useful" and that they would be able to save "almost all the necessary time compared to the manual task".

One of the developers pointed out that:

> *"Trivially, I will not need to read each single review to know what requirements I should implement."*

This represents exactly the first goal of CHANGEADVISOR, *i.e.* trying to reduce a huge number of user reviews to a smaller set of useful contents through the clustering process.

On the other hand, CHANGEADVISOR links user feedback directly to source code components. Even though a developer usually is aware of the source code components responsibilities, the linking process would reveal itself to be very useful, as pointed out by two developers:

> *"I know what changes I have to make in my app when implementing a change. However, a tool like this may help in quantifying the number of classes to be modified."*

> *"It would be useful for letting me immediately know to what extent a change is complicated to apply."*

Therefore, the original developers of the apps in our dataset highlighted that the actual usefulness of CHANGEADVISOR is not only that of providing a toolkit able to support them during the daily activities of user reviews reading and understanding, but also that of providing a tool to estimate the possible cost of some change.

In Summary. The developers confirm the usefulness of CHANGEADVISOR in practice. In particular, not only do the developers think that our approach would be useful in grouping together reviews with the same meaning and linking to source components, but also in quantifying the extent of changes.

C. Threats to Validity

Threats to *construct validity* are mainly related to how we measured developers' estimation of cluster *cohesiveness*, linking *precision*, and *completeness*. We used a Likert scale [54] that allows comparing responses from multiple respondents. However, we are aware that questionnaires may only reflect a subjective perception of the problem.

Threats to *internal validity* are related to the selection of the top 10 links based on the higher Dice coefficient values resulting from the first study. We made it to avoid having a long questionnaire that might have discouraged developers to take part in our study. Moreover, a factor that may have affected the results of the survey study is the response rate of 9.8 % although it covers each app in the dataset.

As for the threats to *external validity*, they are related to how we generalized the results. A replication of the study is part of out future agenda.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced CHANGEADVISOR, a novel approach developed with the goal of supporting developers in accommodating user change requests of mobile apps and planning a new release of an application.

We believe that CHANGEADVISOR represents a significant step forward to the current state-of-art of app reviews mining since, to the best of our knowledge, it is the first approach able to (i) extract user feedback relevant from a maintenance perspective, (ii) cluster it according to similar user needs, and (iii) determine the code artifacts that need to be maintained to accommodate the change requests.

The first evaluation of CHANGEADVISOR demonstrated that the approach is able to (i) identify cohesive user feedback clusters representing similar change requests (**RQ$_1$**), (ii) identify the source code components impacted by the suggested changes with 81 % of *precision* and 70 % of *recall* (**RQ$_2$**), and (iii) it is more accurate than a state-of-the-art technique developed in the context of bug localization [21]. The study conducted with the original developers of the apps in our dataset confirmed the usefulness of our approach in practice (**RQ$_3$**).

Our future research agenda includes (i) the extension of the empirical study to include a larger number and variety of apps from different app stores, (ii) the comparison of CHANGEADVISOR with other baseline techniques in the context of software traceability (*e.g.*, the technique proposed by Bachelli *et al.* [56]), and (iii) the integration of a mechanism for prioritizing the suggested changes. Specifically, the prioritization mechanism will take into account both user requests popularity and the interdependencies between the impacted code components to help developers focus on requests important for users and avoid working on conflicting software changes.

References

[1] D. A. da Costa, S. McIntosh, U. Kulesza, and A. E. Hassan, "The impact of switching to a rapid release cycle on the integration delay of addressed issues: An empirical study of the mozilla firefox project," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 374–385. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901764

[2] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality?: An empirical case study of mozilla firefox," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, ser. MSR '12. IEEE Press, 2012, pp. 179–188. [Online]. Available: http://dl.acm.org/citation.cfm?id=2664446.2664475

[3] M. Nayebi, B. Adams, and G. Ruhe, "Release practices for mobile apps - what do users and developers think?" in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016*, 2016, pp. 552–562.

[4] B. Adams and S. McIntosh, "Modern release engineering in a nutshell – why researchers should care," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, 2016, pp. 78–90.

[5] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study." in *In Proceedings of the 21st IEEE International Requirements Engineering Conference (RE 2013).* IEEE Computer Society, 2013, pp. 125–134. [Online]. Available: http://dblp.uni-trier.de/db/conf/re/re2013.html#PaganoM13

[6] V. N. Inukollu, D. D. Keshamoni, T. Kang, and M. Inukollu, "Factors Influencing Quality of Mobile Apps:Role of Mobile App Development Life Cycle," *ArXiv e-prints*, Oct. 2014.

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

[8] 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.

[9] L. V. Galvis Carreño and K. Winbladh, "Analysis of user comments: An approach for software requirements evolution," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 582–591. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486865

[10] S. Krusche and B. Bruegge, "User feedback in mobile development," in *Proceedings of the 2Nd International Workshop on Mobile Development Lifecycle*, ser. MobileDeLi '14. New York, NY, USA: ACM, 2014, pp. 25–26. [Online]. Available: http://doi.acm.org/10.1145/2688412.2688420

[11] T. Vithani, "Modeling the mobile application development lifecycle," in *Proceedings of the International MultiConference of Engineers and Computer Scientists 2014, Vol. I*, ser. IMECS 2014, 2014, pp. 596–600.

[12] F. Palomba, M. Linares-Vasquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, Sept 2015, pp. 291–300.

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

[14] U. Abelein, H. Sharp, and B. Paech, "Does involving users in software development really influence system success?" *IEEE Software*, vol. 30, no. 6, pp. 17–23, 2013.

[15] S. A. Licorish, A. Tehir, M. F. Bosu, and S. G. MacDonell, "On satisfying the android os community: User feedback still central to developers' portfolios," in *2015 24th Australasian Software Engineering Conference (ASWEC*, 2015, pp. 78–87.

[16] E. Ha and D. Wagner, "Do android users write about electric sheep? examining consumer reviews in google play," in *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, Jan 2013, pp. 149–157.

[17] J. Oh, D. Kim, U. Lee, J.-G. Lee, and J. Song, "Facilitating developer-user interactions with mobile app review digests," in *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '13. New York, NY, USA: ACM, 2013, pp. 1809–1814. [Online]. Available: http://doi.acm.org/10.1145/2468356.2468681

[18] A. Di Sorbo, S. Panichella, C. Alexandru, J. Shimagaki, C. Visaggio, G. Canfora, and H. Gall, "What would users change in my app? summarizing app reviews for recommending software changes," in *Foundations of Software Engineering (FSE), 2016 ACM SIGSOFT International Symposium on the*, 2016, p. to appear.

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

[20] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *Proceedings of the International Conference on Software Engineering (ICSE).* IEEE, 2016.

[21] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 345–355.

[22] F. Palomba, P. Salza, S. Panichella, A. Ciurumelea, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing code changes for mobile apps based on user reviews: Online appendix," Tech. Rep., 2016, https://sites.google.com/site/changeadvisormobile/.

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

[24] 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.

[25] 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.

[26] 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.

[27] A. De Lucia, A. Marcus, R. Oliveto, and D. Poshyvanyk, *Software and Systems Traceability*. London: Springer London, 2012, ch. Information Retrieval Methods for Automated Traceability Recovery, pp. 71–98. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-2239-5_4

[28] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[29] 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*, 2010, pp. 375–384.

[30] 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.

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

[33] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806817

[34] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 522–531. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486857

[35] S. Panichella, A. Di Sorbo, E. Guzman, C. Visaggio, G. Canfora, G. Gall, H.C., and H. Gall, "Ardoc: App reviews development oriented classifier," in *Foundations of Software Engineering (FSE), 2016 ACM SIGSOFT International Symposium on the*, 2016, p. to appear.

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

[37] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp.

316–344, Dec. 2005. [Online]. Available: http://doi.acm.org/10.1145/1118890.1118892

[38] S. Gupta, S. Malik, L. L. Pollock, and K. Vijay-Shanker, "Part-of-speech tagging of program identifiers for improved text-based software engineering tools," in *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*, 2013, pp. 3–12.

[39] A. Mahmoud and N. Niu, "On the role of semantics in automated requirements tracing," *Requir. Eng.*, vol. 20, no. 3, pp. 281–300, Sep. 2015. [Online]. Available: http://dx.doi.org/10.1007/s00766-013-0199-y

[40] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving ir-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 743–762, 2013.

[41] "Nltk. http://www.nltk.org."

[42] "Textblob. https://textblob.readthedocs.org/en/dev/."

[43] "Pyenchant. http://pythonhosted.org/pyenchant/."

[44] "Wordnet english stopword list. http://www.d.umn.edu/~tpederse/Group01/WordNet/wordnet-stoplist.html."

[45] P. M. Nadkarni, L. Ohno-machado, and W. W. Chapman, "Natural language processing: an introduction," *J Am Med Inform Assoc*, p. 2011.

[46] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *5th Berkeley Symp. Mathematical Statistics and Probability*, 1967, pp. 281–297.

[47] L. Kaufman and P. J. Rousseeuw, *Introduction*. John Wiley & Sons, Inc., 2008, pp. 1–67. [Online]. Available: http://dx.doi.org/10.1002/9780470316801.ch1

[48] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei, "Hierarchical dirichlet processes," *Journal of the American Statistical Association*, 2012.

[49] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944937

[50] A. Gelman, C. Robert, N. Chopin, and J. Rousseau, "Bayesian data analysis," 1995.

[51] 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.

[52] "F-droid. https://f-droid.org."

[53] Google, "Google play store. https://play.google.com."

[54] R. Likert, "A technique for the measurement of attitudes," *Archives of Psychology*, vol. 22, no. 140, 1932.

[55] R. M. Groves, *Survey Methodology, 2nd edition*. Wiley, 2009.

[56] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *Proceedings of 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 375–385. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2012.6227177