ChangeAdvisor: A Change Requests Recommender Based on Android User Reviews

Abstract—**Android user reviews often contain precious information that can be profitably used to guide software maintenance activities. Nevertheless, because of their high frequency and lack of textual structure, analyzing the user feedback may result in a problematic and manual effort. In this paper, we describe ChangeAdvisor, a tool that (i) examines the structure, semantics, and sentiments of user reviews to distill change requests to be addressed; (ii) then, it leverages information retrieval approaches to localize the code artifacts that need to be changed. The quantitative and qualitative evaluation of ChangeAdvisor, involving 10 open source mobile apps and their original developers, showed a high accuracy of the presented tool in (i) distilling and clustering user change requests and (ii) linking the source code components that need to be changed to accomodate such requests. In evaluating ChangeAdvisor we found that the original developers of the analyzed apps confirm the practical usefulness of the provided software change recommendations.**
Index Terms—**Mobile Applications, User Reviews Analysis**

I. Introduction

In order to hit the market as quick as possible and to win market share, the software development industry has embraced a paradigm of short and iterative development cycles. Especially in the world of mobile applications, where the advent of mobile app stores has made it easier than ever to manage updates. With the app stores, came also the possibility of users to voice their opinion regarding software in the form of short texts and numerical scores. These reviews are free text and may contain information relevant for developers [8], albeit informally and unstructured. The information contained ranges from problem reports [10], to feedback regarding features [5], requests for enhancements [7] and new features [3], [10], and comparisons with competing apps. Thus app stores provide provide people a way to give their opinion in a quick and simple manner, but it is also a powerful crowd feedback mechanism, which enables developers to discover and fix potential bugs as soon as possible, as well as redirecting the project direction to fulfill market requirements.

Thus, as a natural consequence, research started to come up with ways to capitalize on this wealth of information. Harman *et al.* [6] introduced the concept of *App Store Mining* as a form of *Mining Software Repositories*.

There are difficulties in exploiting this information, however. The various app stores, usually provide only limited support for sorting and aggregating this data to derive requirements [1]. Considering the sheer amount of reviews popular apps receive, it is sometimes not feasible at all to read them all, even less to distinguish which ones are relevant for maintenance tasks. Thus researchers practitioners have come up with new approaches to distinguish relevant reviews [7], [9], [12]. Many of these approaches however only classify reviews, without considering semantics. Additionally there exists no way of correlating user feedback to the actual code components that need to be changed.

II. ChangeAdvisor

This section briefly describes the approach and technologies we employed. ChangeAdvisor is a tool built with a set of different technologies: the core part, the back-end, is written in Java while the front-end application is written in React. This section reports the main characteristics of ChangeAdvisor, as well as details about its architecture and inner-working.

A. Approach Overview

The ChangeAdvisor approach was first introduced by Palomba *et. al* [11]. The goal is to extract feedback relevant to software maintenance tasks from user reviews and suggest locations in code where improvements should be made. It consists of 4 main steps:

- user feedback classification (i.e. bug fixing, feature requests and enhancements);
- source code and user feedback preprocessing;
- user feedback clustering to create groups of reviews discussing the same changes;
- determining code elements related to the reviews clusters.

B. Design Goals

The newest iteration of ChangeAdvisor was devised to improve performance and usability. In order to support this goal, ChangeAdvisor leverages a database to cache computation results and improve search performance. It includes a user-interface in the form of a web application, which developers can use to easily import user reviews and source code. For reasons of space, screenshots of the user interface can be found on the tool's web page. In addition, it is also possible to update ChangeAdvisor's configuration, start jobs manually, and view a dashboard which includes information such as a timeseries of number of reviews vs. average of reviews, distribution of reviews based on category and labels representing the various review clusters.

Tools to import source code and user feedback are provided. Source code can be imported from repositories (at the moment only git is supported, but others will be added at a later date). Review data can be imported in two ways: (i) via a review miner; (ii) from the file system. All import tools can be configured and started through the UI.

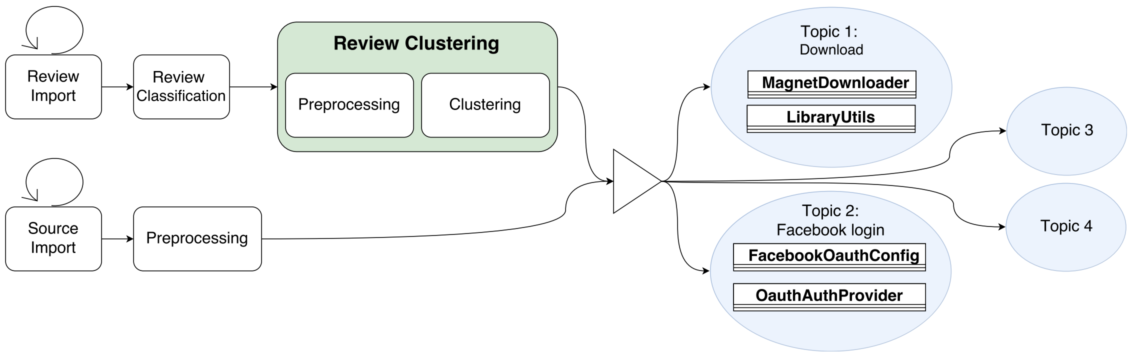

Fig. 1. Change Advisor Processing Pipeline

C. CHANGEADVISOR *Architecture*

The overall architecture of CHANGEADVISOR is depicted in Figure 1. As mentioned previously, the tool is divided in a client-side application and a back-end.

The back-end represents the core of CHANGEADVISOR. The entirety of the logic is implemented at this level. On a logical level, this system can be divided in three components: (i) the *Review Pipeline* that handles the import and preprocessing of user feedback; (ii) the *Source Code Pipeline* which imports source code and preprocessing of source code components; (iii) the *Linking*, which represents the core feature of the CHANGEADVISOR approach, i.e. the link between the source code component and the user feedback referring such elements.

The back-end was written in Java upon Spring Boot and fully leverages the Spring Framework: Spring REST for the HTTP API; Spring Data for database access; Spring Batch for long running jobs. The front-end was written in React and communicates with the back-end via the aforementioned API. Given that most processes of CHANGEADVISOR are long running, we leverage the database to persist the results of the various steps of the tool in order to never have to recompute previous results. Additionally, we leverage the database for costly search and aggregation operations.

The upcoming sections will further detail both front- as well as back-end and their composing components.

D. *The Review Subsystem*

This system can logically be seen as a sequence of four steps: (i) review import which can be configured based on a schedule to automatically mine user reviews; (ii) the review classification step where we automatically classify reviews using a feedback classifier, into labeled categories representing whether a review represents a feature request (`FEATURE REQUEST`) or a bug request (`BUG REQUEST`); (iii) the review preprocessing step in order to clean the user feedback from textual noise which might hinder the accuracy of the clustering and linking [11]; (iv) the clustering step which groups together user reviews discussing the same change request.

a) Review import: CHANGEADVISOR relies for the first step on an external tool, a Java-built scraper tool [4] that relies on PhantomJS[1] and Selenium[2] to navigate to and extract the reviews from the Google Play Store.

b) Review classification: Once the reviews are mined, we employ another external tool, a review classifier [13] which uses Natural Language Processing (NLP), Sentiment and Text Analysis (SA & TA) [13] in order to process the reviews into categories such as *FEATURE REQUEST* and *BUG REPORT*. We do this to discern the ones that are useful for linking later on.

c) Preprocessing: To prepare the reviews for clustering, we apply the typical NLP noise removal steps: contraction expansion, stopword removal, and stemming [11]. This results in a collection of bag of words which can then be used as input to a clustering algorithm.

d) Clustering: The goal of this step is to build clusters of reviews discussing the same change requests. Ideally, at the end of this step we have groups of reviews semantically similar to each other. These clusters will then be fed to the linking algorithm, as one of the inputs. We use clustering for two reasons: (i) as has been shown in the work of Palomba *et al.* [11] linking single reviews tends to return poor results, and (ii) clustering allows us to group together feedback referring ideally to the same, or at the very least, similar change request, thus avoiding duplicates and making the system more comprehensible and generally easier to follow for the developer. CHANGEADVISOR performs two separate approaches to clustering. As with the original CHANGEADVISOR paper [11] we use Hierarchical Dirichlet Process (HDP), proposed by Teh *et al.* [14]. HDP is an extension of the LDA algorithm where the number of topics is not known a priori. The `HierarchicalDirichletProcess` class implementing this approach, is a Java port of the `hdpla` python script that was used in the CHANGEADVISOR paper [11]. Additionally, we attempt to create clusters by computing the *Term-frequency inverse-document-frequency* (tf-idf) score over the N-grams of all reviews, and then picking the N-grams with the highest score and fetching all reviews containing said N-gram.

[1]http://phantomjs.org
[2]http://www.seleniumhq.org

Regardless of the clustering technique employed, at the end, we will have a set of processed reviews that we can use for linking. The return types of both clustering algorithms implement the `LinkableReview` interface, allowing the linker to use any of the two. Finally, both outputs are persisted to the database. By doing this, we do not need to compute new clusters each time we want to run the *linker*. The clusters are only recomputed when new reviews are imported into CHANGEADVISOR.

E. The Source Code Subsystem

The goal of this subsystem is to provide an easy to use interface to automatically import and process source code into a format that CHANGEADVISOR can use as input to its linking algorithm.

a) Source code import: Via the web interface a user can easily enter the url of the git repository. CHANGEADVISOR will then automatically clone the repository, parse the code and divide it in its elements. We define a *source code element* as the basic building blocks of an application, i.e. a *class* in Java. We only keep classes (normal, nested and static nested) and enums. We discards interfaces as they do not implement any logic and would thus rarely be targets for change requests. For each *source code element*, we parse its public API and comments.

b) Source code preprocessing: After parsing, we process each code element as we did with the review pipeline. We split composed identifiers, such as camelCase and snake_case, we remove special characters, english and programming stop words and stem from each code element to produce bag of tokens which we then persist in the database, ready to be used for the linking algorithm.

F. Linking

The linking between the source code elements and the user reviews represents the core of CHANGEADVISOR. After the previous steps of the two pipelines, we the preprocessed reviews and source code elements are retrieved from the database. Each two bag of words (one of each type) are compared using the Dice similarity coefficient [2], defined as follow:

$$\text{Dice}\left(cluster_i, source_j\right) = \frac{\left| W_{cluster_i} \cap W_{source_j} \right|}{\min\left(\left| W_{cluster_i} \right|, \left| W_{souce_j} \right|\right)}$$

where $W_{cluster_i}$represents the set of words contained in cluster i. W_{source_j} represents the set of words contained in the source code element j. We normalize with respect to the shortest document, because user feedback is, in most cases, considerably shorter than our source code elements. Due to the normalization step, the Dice coefficient range is [0, 1]. The tool, then, links stack traces and reviews having a Dice score higher than 0.5 (in a range between $[0, 1]$). The overall process is implemented in the `ChangeAdvisorLinker` class, which utilizes the `AsymmetricDiceIndex` class for the calculation of the index.

III. EVALUATION

IV. DEMO REMARKS

REFERENCES

[1] U. Abelein, H. Sharp, and B. Paech. Does involving users in software development really influence system success? *IEEE software*, 30(6):17–23, 2013.

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

[3] 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*, pages 582–591. IEEE Press, 2013.

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

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

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

[7] C. Iacob and R. Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 41–44. IEEE Press, 2013.

[8] 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 preprint arXiv:1410.4537*, 2014.

[9] 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*, pages 1809–1814. ACM, 2013.

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

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

[12] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *Software maintenance and evolution (ICSME), 2015 IEEE international conference on*, pages 281–290. IEEE, 2015.

[13] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. Ardoc: App reviews development oriented classifier. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1023–1027. ACM, 2016.

[14] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei. Sharing clusters among related groups: Hierarchical dirichlet processes. In *Advances in neural information processing systems*, pages 1385–1392, 2005.