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

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. Design Goals

ChangeAdvisor is designed with the main goal to link user feedback, discussing changes in a high-level language, such as feature requests and bug reports, to the source code components that will require modification in order to fulfill said change requests. In order to support this goal, ChangeAdvisor includes a user-interface in the form of a web application, which the developers can use to easily import user reviews and source code. Additionally, they can update ChangeAdvisor's configuration, start jobs manually, and view a dashboard which includes information such as a time-series of number of reviews vs. average of reviews, distribution of reviews based on the category given by `Ardoc` [3] (e.g. FEATURE REQUEST, PROBLEM DISCOVERY, INFORMATION GIVING, etc.), and labels representing the various review clusters.

Both a review miner and a tool to more easily import source code from repositories (at the moment only git is supported, but others will be added at a later date) were added, both of which can be configured and started through the UI. Since the review miner is heavily dependent on the format of the Google Play Store, updates to the store might risk breaking the miner, and thus the possibility to import reviews from the file system is also present.

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

C. The Review Pipeline

The aim of this subsystem is to facilitate in the fetching and processing of user feedback into clusters that can then be used as input for the linking algorithm.

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 process reviews into categories such as *FEATURE REQUEST* and *BUG REPORT* using a feedback classifier [3]; (iii) the review preprocessing step in order to clean the user feedback from noise; (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 that relies on

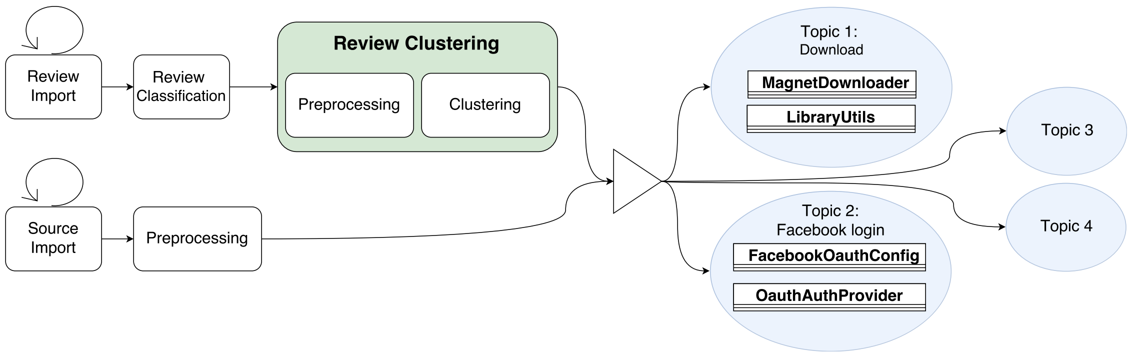

Fig. 1. Change Advisor Processing Pipeline

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 [3] which uses Natural Language Processing (NLP), Sentiment and Text Analysis (SA & TA) 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. 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 core feature of CHANGEADVISOR, 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.* [2] 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 tries two separate approaches to clustering. As with the original CHANGEADVISOR paper [2] we use Hierarchical Dirichlet Process (HDP), proposed by Teh *et al.* [4]. 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 [2]. 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. This functionality is implemented in the `ReviewAggregationService` class.

[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.

D. 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. This is achieved mainly in two classes: `DirectoryCrawler`, which recursively crawls the project's directory and `FSProjectParser`, that given a list of path to Java files creates an in-memory representation of the project, with easy to use methods to parse the corpus of the files.

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.

E. 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 [1], 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] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[2] 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.

[3] 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.

[4] 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.