2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Release Planning of Mobile Apps Based on User Reviews

Lorenzo Villarroel[1], Gabriele Bavota[1], Barbara Russo[1],
Rocco Oliveto[2], Massimiliano Di Penta[3]
[1]Free University of Bozen-Bolzano, Bolzano, Italy
[2]University of Molise, Pesche (IS), Italy — [3]University of Sannio, Benevento, Italy
lorenzo.villarroel@stud-inf.unibz.it, {gabriele.bavota, barbara.russo}@unibz.it,
rocco.oliveto@unimol.it, dipenta@unisannio.it

ABSTRACT

Developers have to to constantly improve their apps by fixing critical bugs and implementing the most desired features in order to gain shares in the continuously increasing and competitive market of mobile apps. A precious source of information to plan such activities is represented by reviews left by users on the app store. However, in order to exploit such information developers need to manually analyze such reviews. This is something not doable if, as frequently happens, the app receives hundreds of reviews per day. In this paper we introduce CLAP (**C**rowd **L**istener for rele**A**se **P**lanning), a thorough solution to (i) categorize user reviews based on the information they carry out (*e.g.,* bug reporting), (ii) cluster together related reviews (*e.g.,* all reviews reporting the same bug), and (iii) automatically prioritize the clusters of reviews to be implemented when planning the subsequent app release. We evaluated all the steps behind CLAP, showing its high accuracy in categorizing and clustering reviews and the meaningfulness of the recommended prioritizations. Also, given the availability of CLAP as a working tool, we assessed its practical applicability in industrial environments.

CCS Concepts

•**Software and its engineering** → *Software maintenance tools;*

Keywords

Release Planning, Mobile Apps, Mining Software Repositories

1. INTRODUCTION

The market of mobile apps is exhibiting a tangible growth and it is expected to reach $70 billion in annual revenue by 2017 [15]. The typical app delivery mechanism is a store in which on the one hand new releases of the app are available for download, and, on the other hand, users rate the app and post reviews. User reviews have the purpose of explaining why the users like or do not like the app, report bugs or request new features. In such a scenario, there is an enormous competition among stakeholders producing similar apps. If an app does not satisfy the users, and if relevant suggestions to improve the app are simply ignored, it is very likely that the app would loose its market share.

User reviews and ratings are therefore very important assets in the development and evolution of mobile apps. Indeed, satisfactorily addressing requests made through user reviews is likely to increase the app rating [25]. However, manually read each user review and verify if it contains useful information (*e.g.,* bug reporting or request for a new feature) is not doable for popular apps receiving hundreds of reviews per day. For such reasons, researchers have developed approaches to analyze the content of user reviews with the aim of crowd-source requirements [14, 19, 21, 26]. Among others, AR-Miner [14] is able to discern informative reviews, group and rank them in order of importance.

While approaches to identify and classify relevant and informative reviews have been proposed, it would be desirable to have a fully-automated (or semi-automated) solution that, given the user reviews for an app, recommends which ones—being them requests for new features or for bug fixes—should be addressed in the next release.

In this paper we propose CLAP, an approach to (i) automatically categorize user reviews into *suggestion for new feature*, *bug report*, and *other* (including non-informative reviews); (ii) cluster together related reviews in a single request, and (iii) recommend which review cluster developers should satisfy in the next release. Unlike AR-Miner [14], CLAP classifies reviews into bug report and feature suggestion, providing additional insights to the developer about the nature of the review. Also, while AR-Miner simply provides a ranking of the user reviews based on their importance as assessed by a pre-defined formula, CLAP learns from past history of the same app or of other apps to determine the factors that contribute to determining whether a review should be addressed or not.

We thoroughly evaluated each step of CLAP, as well as of the whole tool. First, we performed a study to assess the accuracy of CLAP in classifying reviews. The second validation stage aimed at comparing the review clusters generated by CLAP with respect to manually-produced clusters. The third validation assessed the ability of CLAP to recommend features and bug fixes for the next app releases. Last, but not least, in the fourth validation stage we provided our tool to managers of three Italian software companies to get quantitative and qualitative feedback about the applicability of CLAP in their everyday decision making process.

ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884818

2. CLAP IN A NUTSHELL

CLAP provides support to developers for the release planning of mobile apps by automatically importing and mining user reviews through a three-step process detailed in the following subsections.

2.1 Categorizing User Reviews

The first step aims at classifying user reviews into three categories: *bug report*, *suggestion for new feature*, and *other*. The rationale is that, as it will be clear in Section 2.3, developers can use different motivations to decide upon implementing bug fixes or requests for new features. Other tools such as AR-Miner [14] classify reviews into informative and non-informative. In our case, we make a more specific classification of informative reviews, whereas the non-informative ones fall into the *other* category. Also, we are aware that, besides bug reports and suggestions for new features, other relevant information could be contained in user reviews (*e.g.,* general comments on the user experience in using the mobile app). However, in this version of CLAP we choose to focus the classification on bugs to fix and features to be implemented, since we believe that they concern the macro activities in the maintenance and evolution of apps (as also confirmed by the three project managers that participated to the evaluation of CLAP).

CLAP uses the *Weka* [8] implementation of the Random Forest machine learning algorithm [11] to classify user reviews. The Random Forest algorithm builds a collection of decision trees with the aim of solving classification-type problems, where the goal is to predict values of a categorical variable from one or more continuous and/or categorical predictor variables. In our work, the categorical dependent variable is represented by the type of information reported in the review (*bug report*, *suggestion for new feature*, or *other*), and we use the rating of the user reviews and the terms/sentences they contain as predictor variables. We have chosen Random Forest after experimenting with different machine learner algorithms (details of the comparison are in the replication package [29]). We adopt a customized text preprocessing to characterize each review on the basis of its textual content. The steps of such a process are detailed in the following and the benefits brought by each step will be shown in our empirical evaluation (Section 3).

Handling Negations. Mining text in code reviews present challenges related to negation of terms. For instance, reviews containing the words "lag" and "glitches" generally indicate *bug reporting*. However, there is a strong exception to this general trend that is due to the negation of terms. Consider the following review: "*I love it, it runs smooth no lags or glitches*". It is clear that in this case, the context in which the words "lag" and "glitches" are used does not indicate any bug. However, the presence of these words in the review could lead to misclassifications from the prediction model. Thus, we adopt the Stanford parser [28] to identify negated terms in the reviews and remove them. For example, we convert "*I love it, it runs smooth no lags or glitches*" into "*I love it, it runs smooth*", a set of words better representing the message brought by the review.

Stop-words and Stemming. Terms belonging to the English stop-words list [2] are removed to reduce noise from the user reviews. Also, we apply the Porter stemmer [27] to reduce all words to their root.

Unifying Synonyms. One possibility to unify synonyms would be to use existing thesaurus such as WordNet [23]. However, in the context of user reviews, we found that general-purpose thesaurus are not adequate. Thus, we rely on a customized dictionary of synonyms that we defined by manually looking at 1,000 reviews (not used in the empirical evaluation) we collected for a previous work [10]. Examples of synsets we obtained are {*freeze*, *crash*, *bug*, *error*, *fail*, *glitch*, *problem*} (terms related to a bug/crash) or {*add*, *miss*, *lack*, *wish*} (terms related to the need for adding a new feature). Noticeably, words such as *freeze*, *crash*, *bug*, and *glitch* would not be considered synonyms by a standard thesaurus, while they are very likely to indicate the same concept in mobile apps reviews.

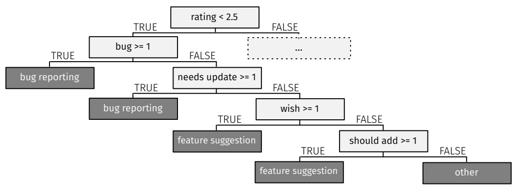


Figure 1: Example of regression tree generated by CLAP **when categorizing user reviews.**

N-grams extraction. Besides analyzing the single words contained in each review, we extract the set of *n-grams* composing it, considering $n \in [2 \ldots 4]$. For instance, the extraction of n-grams from a review stating "*The app resets itself; Needs to be fixed*" will result in the extraction of n-grams like *resets itself*, *needs to be fixed*, *etc.* Note that the three preprocessing steps described above are not performed on the n-grams (*i.e.,* they only affect the single words extracted from the review). This is done to avoid loosing important information embedded in n-grams. For example, managing negations is not needed when working with n-grams, since n-grams extracted from a review like "*I love it, it runs smooth no lags or glitches*" will include *no lags*, *no lags or glitches*, *etc.* Synonyms merging also is not applied to n-grams to avoid changing their meaning, *e.g.,"the app freezes"* should not be converted in *"the app bug"* (being *freeze* and *bug* synonyms according to our thesaurus).

After text preprocessing, we classify a user review using as predictor variables: (i) its rating, (ii) the list of n-grams derived from it, and (iii) the bag of words left as output of the previously described steps. Training data, with pre-assigned values for the dependent variables are used to build the Random Forest model. An example of generated classification tree is shown in Figure 1 (due to limited space, we just show the left-hand subtree of the root node).

2.2 Clustering Related Reviews

In order to identify groups of related reviews (*e.g.,* those reporting the same bug), we cluster reviews belonging to the same category (*e.g.,* those in *bug report*). Clustering reviews is needed for two reasons: (i) developers having hundreds of reviews in a specific category could experience information overloading, wasting almost all advantages provided by the review classification, and (ii) knowing the number of users who are experiencing a specific problem (bug) or that desire a specific feature, already represents an important information about the urgency of fixing a bug/implementing a feature. Note that we only cluster reviews classified as *bug report* or *suggestion for new feature* since those are the ones the developer should be interested in for planning the next release of her app.

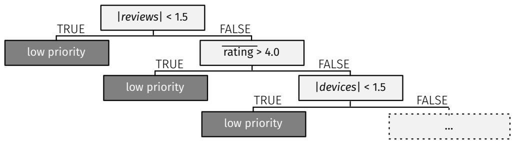


Figure 2: Example of regression tree generated by CLAP **when prioritizing clusters.**

Review clustering is performed by applying DBSCAN [17], a clustering algorithm identifying clusters as areas of high element density, assigning the elements in low-density regions to singleton clusters (*i.e.,* clusters only composed by a single element). In CLAP, the elements to clusters are the reviews in a specific category and the distance between two reviews r_i and r_j is computed as: $dist(r_i, r_j) = 1 - VSM(r_i, r_j)$, where VSM is the Vector Space Model [9] cosine similarity between r_i and r_j adopting tf-idf [9] as term-weighting schema. Before applying VSM the text in the reviews is processed as described in the categorization step (Section 2.1), with the only exception of the synonyms merging. Indeed, merging synonyms before clustering could be counterproductive since, for example, a review containing "freezes" and a review containing "crash" could indicate two different bugs. DBSCAN does not require the definition a-priori of the number of clusters to extract. However, it requires the setting of two parameters: (i) $minPts$, the minimum number of points required to form a dense region, and (ii) ϵ, the maximum distance that can exist between two points to consider them as part of the same dense region (cluster). We set $minPts = 2$, since we consider two related reviews sufficient to create a cluster, while in Section 3 we describe how to set ϵ.

2.3 Prioritizing Review Clusters

The clusters of reviews belonging to the *bug report* and *suggestion for new feature* categories are prioritized with the aim of supporting release planning activities. Also in this step CLAP makes use of the Random Forest machine learner with the aim of labelling each cluster as *high* or *low* priority, where *high* priority indicates clusters CLAP recommends to be implemented in the next app release. Thus, the dependent variable is represented by the cluster implementation priority (*high* or *low*), while we use as predictor features:

The number of reviews in the cluster ($|reviews|$). The rationale is that a bug reported (feature suggested) by several users should have a higher priority to be fixed (implemented) than a bug (feature) experienced (wanted) by a single user.

The average rating of the cluster ($\overline{rating}$). We hypothesize that a review cluster having a low average rating has a higher chance to indicate a higher priority bug (or a feature to be implemented urgently) than a cluster containing highly-rated reviews, and thus should have a higher priority. For example, people frustrated by the presence of critical bugs are more likely to lowly rating the app.

The difference between the average rating of the cluster and the average rating of the app ($\Delta rating_{app}$). This feature aims at assessing the impact of a specific cluster on the app total rating. We expect a lower difference (especially negative ones) to indicate higher priority for the cluster.

The average difference of the ratings assigned by users in the cluster who reviewed older releases of the app ($\Delta rating_u$). A Google Play user can review multiple releases of an app over time. Clearly, her rating can change over time as a consequence of her satisfaction in using the different releases. Given a cluster C containing a set of reviews R referring to the release r_i, we compute the average difference of the ratings assigned by authors of R with respect to last rating (if any) they assigned to the releases r_x, with $x < i$. If the authors of R did not review the app before r_i, she is not considered in the computation of $\Delta rating_u$. If none of the authors of R evaluated the app in the past, $\Delta rating_u = 0$.

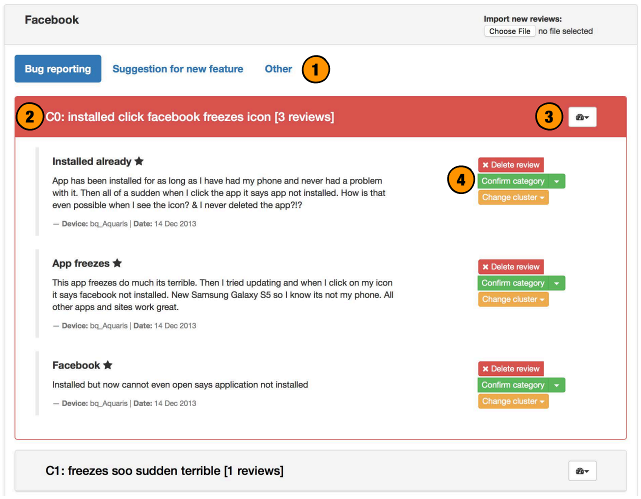
The number of different hardware devices in the cluster ($|devices|$). One of the information available to app developers when exporting their reviews from Google Play is the "Reviewer Hardware Model", reporting the name of the device used by the reviewer. We conjecture that the higher $|devices|$, the higher the priority of a cluster. For example, if a cluster of *bug report* reviews contains reviews written by users exploiting several different devices, the bug object of the cluster is likely to affect a wider set of users with respect to a bug only reported by users working with a specific device. Similarly, this holds in the case of "desired features", since the same app can expose different features on different devices (*e.g.,* on the basis of the screen size).

Also in this case, historical data with known (and labeled) value of the dependent variable is used to build the Random Forest decision tree. Note that, given the different nature of reviews reporting bugs and those suggesting new features, the prioritization is performed separately for clusters containing the two types of reviews (bugs and features). A portion of a tree generated in this step can be found in Figure 2.

2.4 CLAP Prototype

Figure 3 reports an excerpt of the user interface of CLAP (we removed side and top menu due to lack of space). In the example shown in Figure 3 the user imported in CLAP a set of user reviews from the Facebook app. As a result, the reviews have been categorized into *suggestion for new feature*, *bug report*, and *other* (see *e.g.,* element 1 in Figure 3).

In the example, the *bug report* category is expanded to unveil its review clusters. Each cluster (see *e.g.,* element 2) has a label composed of (i) a simple identifier (*e.g.,* C1), and (ii) the five most frequent terms in the reviews belonging to it. Red clusters (*e.g.,* C0 in Figure 3) are those marked by CLAP as "high priority" clusters, while grey clusters (*e.g.,* C1) represent the "low priority" ones. The tool also



C0: installed click facebook freezes icon [3 reviews]

Installed already ★

App has been installed for as long as I have had my phone and never had a problem with it. Then all of a sudden when I click the app it says app not installed. How is that even possible when I see the icon? & I never deleted the app?!?

— **Device:** bq_Aquaris | **Date:** 14 Dec 2013

App freezes ★

This app freezes do much its terrible. Then I tried updating and when I click on my icon it says facebook not installed. New Samsung Galaxy S5 so I know its not my phone. All other apps and sites work great.

— **Device:** bq_Aquaris | **Date:** 14 Dec 2013

Facebook ★

Installed but now cannot even open says application not installed

— **Device:** bq_Aquaris | **Date:** 14 Dec 2013

C1: freezes soo sudden terrible [1 reviews]

Figure 3: User interface of CLAP**.**

provides a feedback mechanism to allow the developer to indicate whether or not she is going to implement the reviews contained in a cluster (element 3 in Figure 3). Such a manual feedback can be used by the developer to expand/revise the automatic prioritization training set according to the features/bugs actually implemented.

Finally, by expanding a cluster, one can see the reviews it contains. As it can be seen in the example, the three reviews of cluster C0 report a similar problem: despite the users installed the FACEBOOK app, when clicking on the app icon they receive the "app not installed" message. Also in this case there is a feedback mechanism (element 4 in Figure 3) to change the review category or to assign it to a different cluster.

3. EMPIRICAL STUDY DESIGN

The *goal* of this study is to evaluate CLAP in terms of its (i) accuracy in categorizing user reviews in the three categories of interest (*i.e., bug report*, *suggestion for new feature*, and *others*), (ii) ability in clustering related user reviews belonging to the same category (*e.g.,* all reviews reporting the same bug), (iii) ability in proposing meaningful recommendations on how to prioritize the bugs to be fixed and new features to be implemented while planning the next release of the app, and (iv) its suitability in an industrial context. The *context* of the study consists of 1,763 reviews of 210 Android mobile apps and three Italian software companies.

The material used in this evaluation along with its working data set is publicly available in our replication package [29].

3.1 Research Questions

In the context of our study we formulated the following four research questions (RQ):

- **RQ$_1$**: *How accurate is* CLAP *in classifying user reviews in the considered categories?* This RQ assesses the accuracy of CLAP in classifying user reviews in the *bug report*, *suggestion for new feature*, and *others* categories. It aims at evaluating the step "categorizing user reviews" described in Section 2.1.

- **RQ$_2$**: *How meaningful are the clusters of reviews generated by* CLAP*?* This RQ focuses on the meaningfulness of clusters of reviews extracted by CLAP in a specific category of reviews (*e.g.,* those reporting bugs). We are interested in assessing the differences between clusters of reviews automatically produced by CLAP with respect to those manually produced by developers. RQ$_2$ evaluates the step "clustering related reviews" described in Section 2.2.

- **RQ$_3$**: *How accurate is the new features/bug fixing prioritization recommended by* CLAP*?* Our third RQ aims at evaluating the relevance of the priority assigned by CLAP to the bugs to fix and new features to be implemented in sight of the next release of the app. We assess the ability of CLAP in predicting which bugs will be fixed (features will be implemented) by developers among those reported (requested) in user reviews of release r_i when working on release r_{i+1}. This RQ evaluates the prioritization step described in Section 2.3.

- **RQ$_4$**: *Would actual developers of mobile applications consider exploiting* CLAP *for their release planning activities?* For a tool like CLAP, a successful technological transfer is the main target objective. In RQ$_4$ we investigate the industrial applicability of CLAP with the help of three software companies developing Android apps. Thus, RQ$_4$ evaluates the CLAP prototype tool as a whole, as described in Section 2.4.

Table 1: Objects used in our research questions.

RQ	#Apps	#Reviews	Origin
RQ$_1$	200	1,000	Randomly selected from [13]
RQ$_2$	5	200	Reviews from popular apps referring to the same app release
RQ$_3$	5	463	Selected on the basis of specific criteria from [13]
RQ$_4$	2	100	Reviews from two very popular apps (Facebook and Twitter)

3.2 Context Selection and Data Analysis

Table 1 summarizes the objects (*i.e.,* apps and user reviews) used in each of our research questions. To address **RQ$_1$** we manually classified a set of 1,000 users reviews randomly selected from 200 different Android apps extracted from the dataset by Chen *et al.* [13]. In particular, two of the authors independently analyzed the 1,000 reviews by assigning each of them to a category among *bugs report*, *suggestion for new feature*, and *others*. Then, they performed an open discussion to resolve any conflict and reach a consensus on the assigned category. This was needed for 69 out of the 1,000 reviews. In total, of the considered 1,000 reviews we labeled 235 as *bug report*, 179 as *suggestion for new feature*, and 596 as *others*. Then, we used this dataset to perform a 10-fold cross validation, computing the overall average accuracy of the model and reporting the obtained confusion matrix.

For **RQ$_2$** we manually collected a second set of 200 user reviews among five Android apps, *i.e.,*FACEBOOK, TWITTER, YAHOO MOBILE CLIENT, VIBER, and WHATSAPP. For this research question we have selected very popular apps since we needed to collect from each app a good number of reviews (i) related to the same app's release, and (ii) belonging to the *bug report* or to the *suggestion for new feature* category. In particular, we randomly selected from each of these apps 40 reviews, 20 *bug reports* and 20 *suggestions for new features*, referring to the same app's release[1]. Then, we asked three industrial developers having over five years of experience each to manually clustering together the set of reviews belonging to the same category (*e.g., bugs report*) in each app. We clearly explained to the developers that the goal was to obtain clusters of reviews referring to the same bugs to be fixed or feature to be implemented.

The three developers independently analyzed each of the 200 reviews to cluster them. After that, they reviewed together their individual clustering results and provided us a single "oracle" reflecting their overall point of view of the existing clusters of reviews. Once obtained the oracle, we used CLAP, and in particular the process detailed in Section 2.2, to cluster together the same sets of reviews.

As previously explained, to apply the DBSCAN clustering algorithm we need to tune its ϵ parameter. We performed such a tuning by running the DBSCAN algorithm on the YAHOO app varying ϵ between 0.1 and 0.9 at steps of 0.1 (*i.e.,* nine different configurations). Note that it does not make sense to run DBSCAN with $\epsilon = 0.0$ or $\epsilon = 1.0$ since the output would trivially be a set of singleton clusters in the former case and a single cluster with all reviews in the second case.

[1]As previously said, in CLAP we are not interested in clustering reviews belonging to the *other* category.

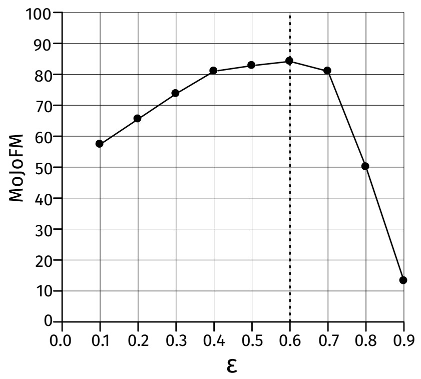


Figure 4: Tuning of the ϵ DBSCAN parameter.

To define the best configuration among the nine tested ones, we measured the similarity between the two partitions of reviews (*i.e.,* the oracle and the one produced by CLAP) by using the MoJo eFfectiveness Measure (MoJoFM) [30], a normalized variant of the MoJo distance based on the minimum number of *Move* or *Join* operations one needs to perform in order to transform a partition A into a partition B. $MoJoFM$ returns 0 if partition A is the farthest partition away from B; it returns 100 if A is exactly equal to B. The results of this tuning are shown in Figure 4. In general, values between 0.5 and 0.7 allows to achieve good performances, with the highest MoJoFM reached at 0.6. This is the default value in CLAP, and thus the one we will use in the evaluation.

In order to evaluate the CLAP's clustering step, we measured the MoJoFM distance on reviews of the remaining apps (*i.e.,* excluding YAHOO). We report the MoJoFM achieved in two different scenarios:

Intermediate feedback available. As explained in Section 2.4, CLAP allows the user to fix review misclassifications (*e.g.,* a review reporting a bug classified as a suggestion for new features). Thus, we firstly simulate a scenario in which the categorization of the reviews has been manually checked (and fixed, when needed) by the developer. This is done by separately running the clustering algorithm on the 20 *bug report* reviews and on the 20 *suggestions for new feature* reviews available for each app. This scenario allows us to assess the performances of the CLAP's clustering step "in isolation", without the risk of having the achieved MoJoFM values affected by misclassification performed in the categorization step.

Fully automated approach. To simulate a fully automated CLAP's usage scenario in which the developer does not act on the review categorization, we use CLAP to automatically categorize the 40 reviews of each app identifying those *reporting bugs* and *suggesting new features.* Then, we cluster them by using the DBSCAN algorithm and compare via MoJoFM the produced clusters of reviews with the oracle defined by the developers. Note that in this case it is possible that some of the 40 reviews are categorized by CLAP in the *other* category. In a real usage scenario, CLAP would not cluster these reviews; thus, we remove them before the clustering. However, since it is not possible to compute the MoJoFM between two different sets of clustered entities (reviews), we also removed these reviews from the oracle. For this reason, in this evaluation scenario we report (i) the MoJoFM achieved by our approach when automatically categorizing the reviews, and (ii) the number of instances **wrongly** discarded by our approach due to a misclassification.

To answer **RQ**$_3$ we exploited the Android user reviews dataset made available by Chen *et al.* [13]. This dataset reports user reviews for multiple releases of ∼21K apps, showing for each review: (i) the date in which it has been posted, (ii) the app's release it refers to, (iii) the user who posted it, (iv) the hardware device exploited by the user, (v) the rating, and (vi) the textual content of the review itself. In addition, each app in the dataset is associated to a metadata file containing its basic information, including the "updated" optional field that app's developers can use to describe the changes they made to the different app's releases (*i.e.,* a sort of release note shown in the Google Play store). We exploited such a dataset to build, for a given app, an oracle reporting which of the reviews left by its users for the release r_i have been implemented by the developers in the release r_{i+1} (*i.e., high priority* reviews) and which, instead, have been ignored/postponed (*i.e., low priority* reviews). To reach such an objective, firstly we identified the apps in the dataset having all the information/characteristics required to build the oracle:

1. A non-empty "updated" field containing at least one English word. As said before, this is an optional field where the app developers can report the changes they applied in a specific app's review. This first filtering was automated by looking for the "updated" fields matching at least one term (excluding articles) in the Mac OS X English dictionary. This left us with ∼11K apps.

2. Explicitly reporting the app's version to which the "updated" field refers. Often developers simply put in the "updated" field the changes applied to the last release of the app without specifying the "release number" (*e.g.,* release 2.1). This is an information needed to build our oracle. Indeed, starting from the release note (*i.e.,* the content of the updated field) of a specific release r_{i+1}, we have to look at the reviews left by users of the release r_i to verify which of them have been actually implemented by the developers. We adopt regular expressions (*e.g.,* `version | release | v | r` followed by at least two numbers separated by a dot) to automatically identify apps reporting the release number in the "updated" field. This left us with ∼1.4K apps.

3. Having a non-ambiguous release note (update field). Release notes only containing sentences like "fixed several bugs" or "this release brings several improvements and new features" are not detailed enough to understand which of the user reviews have been implemented by developers. For this reason, one of the authors manually looked into each of these 1.4K apps for those containing a non-ambiguous release note. This selection led to only 73 apps remaining.

4. Having available at least 30 reviews for the release r_i preceding the r_{i+1} described in the release note. The dataset by Chen *et al.* does not report reviews for all releases of an app. Thus, it is possible that the reviews for r_i are not available or are too few for observing something interesting. This further selection process led to the five apps considered in our study: `barebones 3.1.0`, `hmbtned 4.0.0`, `timeriffic 1.11`, `ebay 2.6.1`, and `viber 4.3.1`.

The five selected app releases received a total of 18,591 user reviews from the r_i releases to be labeled as "implemented" or "not implemented" in r_{i+1}. Since manually labeling all of them would not be feasible in a reasonable time, we randomly selected from each app a statistically significant sample of reviews with 95% confidence level and 5% confidence interval. This resulted in 463 reviews that were manually analyzed by two of the authors independently, and labeled as implemented/not implemented on the basis of the information contained in the related release note. Also in this case, conflicts (raised for 37 reviews) were solved with an open discussion. Once built the oracle for the five apps, we performed a 10-fold validation to assess the ability of CLAP, and in particular of the prioritization step described in Section 2.3, to correctly identify the clusters of reviews that should be prioritized in sight of the next app's release (*i.e.,* those that have been actually implemented by developers in the r_{i+1} reviews). As already done for RQ$_2$, we report the performance of CLAP considering two different scenarios:

1. A scenario in which the categorization of the reviews into *bug report*, *suggestion for new feature*, and *other*, as well as the result of the clustering step has been manually checked (and fixed, when needed) by the developer. To support such a scenario, the two authors who labeled the 463 reviews as implemented/not implemented also (i) categorized them in one of the three supported categories, and (ii) manually clustered them. This manual process led to the identification of 55 clusters of *bug report* reviews and 30 clusters of *suggestion for new features* of which 5 of those reporting bugs and 9 of those suggesting features contain implemented reviews (*i.e.,* are *high priority* clusters). In this first scenario we assess the accuracy of the CLAP's prioritization step in "isolation", by performing a ten-fold validation on the set of clusters manually defined. Given the unbalanced distribution of *high priority* and *low priority* clusters (*e.g.,* 5 *vs* 55 for those related to bug reporting), we balanced at each iteration of the ten-fold validation the training set via an under-sampling procedure, randomly selecting from the training set an equal number (*i.e.,* the number of the underrepresented cluster category) of *high* and *low priority* clusters. To avoid any bias, no changes were applied to the test set.

2. A fully automated scenario, in which we used CLAP to categorize, cluster, and prioritize the obtained clusters. Note that, differently from the manually defined clusters, it is possible that automatically generated clusters contain both "implemented" and "not implemented" reviews (as manually defined in the oracle), due to errors in the clustering step. In this case we consider a cluster as correctly classified as *high* (*low*) priority if its centroid has been marked as "implemented" ("not implemented") in the manually defined oracle. Also in this case, a ten-fold validation has been performed and under-sampling applied in case of unbalanced training sets.

Finally, to answer **RQ$_4$** we conducted semi-structured interviews with the project managers of three software companies developing Android apps[2]. Before the interviews, one of the authors showed a demo of CLAP, and let the participant interact with the tool. To avoid biases and evaluate the tool with a consistent set of reviews, all project managers worked with a version of CLAP having the reviews for TWITTER and FACEBOOK imported. Note that, using the reviews of the apps developed by the three companies was not an option, since most of the reviews they receive are not in English and the current implementation of CLAP only supports such a language. The interviews lasted for two hours with

[2]RQ$_4$'s participants were not the same involved in RQ$_2$.

Table 2: RQ$_1$: Classification Accuracy of Reviews.

Category	No text preprocessing		N-gram analysis		Handling & Negations		Stop words/ Stemming		Unifying Synonyms	
	R	P	R	P	R	P	R	P	R	P
bug report	68%	78%	70%	78%	72%	80%	73%	81%	76%	88%
sugg. new feature	63%	68%	66%	70%	66%	72%	66%	78%	67%	87%
other	84%	77%	90%	76%	91%	79%	93%	80%	96%	86%
overall accuracy	**70%**		**73%**		**77%**		**81%**		**86%**	

Table 3: RQ$_1$: Confusion Matrix.

	bug report	sugg. new feature	other	Recall	Precision
bug report	**178**	4	53	76%	88%
sugg. new feature	19	**115**	45	67%	87%
other	10	13	**572**	96%	86%

each company. Each interview was based on the think-aloud strategy. Specifically, we showed all the tool features to the managers to get qualitative feedback on both the tool and the underlying approach. In addition, we explicitly asked the following questions:

Usefulness of reviews. Do you analyze user reviews when planning a new release of your apps?

Factors considered for the prioritization phase. Are the factors considered by CLAP reasonable and sufficient for the prioritization of bugs and new features?

Review categories. Is the categorization of reviews into *bug report* and *suggestion for new feature* sufficient for release planning or there are other categories that should be taken into account?

Tool usefulness. Would you use the tool for planning new releases of your apps?

Participants answered each question using a score on a four-point Likert scale: 1=absolutely no, 2=no, 3=yes, 4=absolutely yes. The interviews were conducted by one of the authors, who annotated the provided answers as well as additional insights about the CLAP's strengths and weaknesses that emerged during the interviews.

4. STUDY RESULTS

This section reports the analysis of the results for the four research questions formulated in Section 3.1.

RQ$_1$: How accurate is CLAP **in classifying user reviews in the considered categories?** Table 2 reports the Recall (R), Precision (P) and overall accuracy achieved by CLAP when classifying user reviews. In particular, we show the accuracy of our classifier when considering/not considering the different text preprocessing steps. The second column on the left (no text preprocessing) reports the classification accuracy (70%) obtained without performing any text preprocessing (*i.e.,* by providing to the machine learner all terms present in the user reviews). By moving toward the right part of Table 2, we can observe the impact on the accuracy of CLAP when: (i) including the extracted n-grams (+3%=73%), (ii) handling negations (+4%=77%), (iii) performing stop word removal and stemming (+4%=81%), and (iv) unifying synonyms (+5%=86%). As it can be seen, the text preprocessing steps adopted in CLAP ensure a +14% of accuracy over the baseline.

Table 3 reports the confusion matrix of CLAP, along with recall and precision values for each category, detailing the overall 86% accuracy (865 correct classifications out of 1,000 reviews) achieved by CLAP. The most frequent case of failure for CLAP is represented by the misclassification of reviews belonging to *bug report* and *suggestion for new feature* categories as *other*, accounting for a total of 98 errors

Table 4: RQ$_2$: MoJoFM achieved by CLAP.

	Facebook	Twitter	Viber	Whatsapp	Average
Scenario I: Manual Categorization					
bug report	76%	75%	67%	72%	**73%**
sugg. new feature	71%	83%	94%	100%	**87%**
Scenario II: Automatic Categorization					
all	72% (2)	77% (3)	70% (3)	80% (2)	**77%**

(72% of the overall errors). A manual inspection revealed that this is mainly due to reviews, related to bugs or new features, not containing any of the keywords that, according to the learned decision tree, lead towards a *bug* or *new feature* classification.

RQ$_2$: How meaningful are the clusters of reviews generated by CLAP**?** Table 4 shows the MoJoFM between the clusters of reviews manually defined by developers and those resulting from the clustering step of CLAP in the two evaluation scenarios described in Section 3.2. In the first evaluation scenario, in which we assume that the developer has fixed possible categorization errors made by CLAP (*e.g.,* a *bug report* review classified as a *suggestion for new feature*), the average MoJoFM is 73% for bugs, and 87% for new features, suggesting a high similarity between manually- and automatically-created clusters. In one case, *i.e.,* the clustering of reviews *suggesting new features* in Whatsapp, the partition is exactly the same, indicating the meaningfulness of the clusters generated by CLAP.

In order to give a better idea of the meaning of such MoJoFM values, Figure 5 shows the two partitions of reviews manually-created by the developers involved in the study (left side) and automatically generated by CLAP (right side) for the 20 Twitter reviews *suggesting new features.* The points in light grey represent the eleven reviews considered both by developers and by CLAP as singleton clusters (*i.e.,* each of these reviews recommended the implementation of a different feature). The points in black represent, instead, the reviews clustered by developers into four non-singleton clusters, depicted with different colors in Figure 5. The first two clusters (the grey and the green ones) are exactly the same in the oracle and in the clusters generated by CLAP. The yellow cluster is similar between the two partitions. However, in the automatically generated partition it does not include the review *R5*, isolated as a singleton cluster by CLAP. Finally, the blue cluster composed of *R8* and *R9* is the only one totally missed by CLAP, that does not recognize the two reviews as semantically related. Overall, these differences resulted in a MojoFM of 83%.

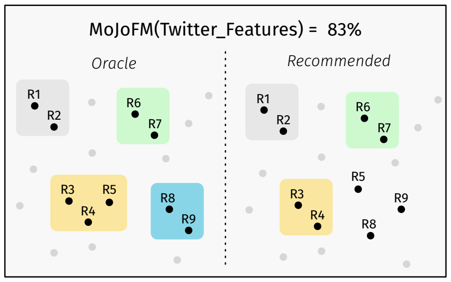


Figure 5: RQ$_2$: CLAP ***vs*** **oracle when clustering suggestions for new features on Twitter.**

Table 5: RQ$_3$: Prioritization accuracy.

	correctly classified	false positive	false negative
Scenario I: Manual Categorization and Clustering			
bug report	87%	9%	4%
sugg. new feature	80%	20%	0%
Scenario II: Automatic Categorization and Clustering			
bug report	73%	15%	12%
sugg. new feature	69%	18%	13%

The example reported in Figure 5 is very representative of the errors made by CLAP in clustering related reviews. We observed as it tends to be more conservative in clustering the reviews with respect to the manually produced oracle (*i.e.,* it generates more singleton clusters). While this could suggest a wrong calibration of the ϵ parameter, we also replicated this study with $\epsilon = 0.7$ and $\epsilon = 0.8$, since higher values of ϵ should promote the merging of related reviews. However, these settings resulted in lower values of the MoJoFM across all experimented systems, due to a too aggressive merging of reviews.

The bottom part of Table 4 reports the MoJoFM achieved in the second evaluation scenario, where the automated categorization of CLAP has been applied. Note that in this case we report the overall MoJoFM without distinguishing between bug reports and suggestions for new features, since we run the clustering on the whole dataset of 40 reviews for each app. This is needed since the automatic categorization of reviews could lead to the introduction of misclassification, and it is not possible to compare via MoJoFM partitions composed of different elements. For example, if a *bug report* review is misclassified by CLAP as a *suggestion for new features*, the set of reviews clustered in the oracle and those clustered by CLAP would be different if looking into the specific categories. For a similar reason, as explained in Section 3.2, we excluded from the comparison reviews misclassified in the *other* category. The number of such reviews are indicated in parenthesis in Table 4, and always account for less than 8% of the classified reviews (*i.e.,* no more than three out of the 40 reviews categorized in each app is wrongly put in the *other* category). As for the MoJoFM, it fluctuates between 70% (Viber) and 80% (Whatsapp), showing again the ability of CLAP to generate clusters of reviews close to those manually defined by developers.

RQ$_3$: How accurate is the new features/bug fixing prioritization recommended by CLAP**?** Table 5 reports the accuracy achieved by CLAP in classifying clusters of *bug report* and *suggestion for new feature* reviews as *high priority* (*i.e.,* the cluster of reviews has been actually implemented by the apps developers in the subsequent release) and *low priority.* False positives are clusters wrongly classified by CLAP as *high priority*, while false negatives are clusters wrongly classified as *low priority.* Results are reported for both the scenarios described in Section 3.2.

In the first scenario (top part of Table 5), simulating a situation in which the CLAP user has manually fixed possible categorization and clustering errors, CLAP correctly prioritizes 87% of clusters containing *bug report* reviews (48 out of 55), producing five (9%) false positive and two false negatives. The accuracy is slightly lower when prioritizing new features to be implemented, with 80% of correctly classified clusters (24 out of 30), six false positives (20%) and zero false negatives. For example, a *bug report* cluster correctly *highly* prioritized by CLAP is the one from the `ebay` app, in which 141 different users using a total of nine different hardware devices were pointing out a bug present in the release 2.6.0 that prevented the app user to visualize the seller's feedbacks. This cluster also had a very low average rating ($\overline{rating}$ = 2.2), much lower that the average app rating ($\Delta rating_{app}$ = -1.9); moreover, the 30 reviewers in this cluster who already evaluated past `ebay` releases assigned a much lower score to this specific release ($\Delta rating_u$ = -1.5). The `ebay` developers fixed this bug in the release 2.6.1, reporting in the release note: "*Fixed bug where seller feedback would not load*".

A false negative generated by CLAP when prioritizing clusters reporting suggestions for new features is a singleton cluster from the `barebones` app, a lightweight mobile browser. One of the users reviewing the release 3.0 assigned five stars to the app and asked for the implementation of search suggestions ("*I wish it can have search suggestions in the search bar*"). Despite the single user requiring such a feature and the high rating she assigned to the app, the `barebones` developers implemented search suggestions in the release 3.1: "*Added Google Search Suggestions*". The characteristics of this cluster led CLAP to a misclassification, since the decision trees generated in the prioritization step tend to assign a *high priority* to clusters having high values for $|reviews|$ and $|devices|$, and low values for $\overline{rating}$, $\Delta rating_{app}$, $\Delta rating_u$ (see the example in Figure 2). Note that these classification trees are the results of the training performed on the five considered apps. In a real scenario, the CLAP user can explicitly indicate which clusters she is going to implement, allowing the machine learner to adapt the classification rules on the basis of the user feedback.

When considering the clusters as produced automatically (bottom part of Table 5), the prioritization accuracy of CLAP exhibits an expected decrease. 73% of clusters related to bug reporting and 69% of those grouping suggestions for new features are correctly prioritized, with the majority of prioritization errors (15% and 18% for the two categories of clusters) due to false positives (*i.e., low priority* clusters classified as *high priority* ones). Such a decrease of performance is due to misclassifications in the review categorization step (as shown in the RQ$_1$ results, CLAP misclassifies ~15% of the reviews) and to errors introduced in the clustering step (RQ$_2$). However, we still believe that this level of accuracy represents a good starting point for helping app developers during release planning activities. Indeed, as highlighted by all the feedback mechanisms we implemented in CLAP, we did not envision our tool to be used as a black box taking user reviews as input and producing a list of prioritized clusters. We rather look at it as a support for app developers interacting with them in order to gather as much information as possible from the user reviews.

Comparison with AR-Miner. To further assess the prioritization performances of CLAP, we compared them with the prioritization performed by the state-of-the-art technique AR-Miner [14]. Note that we did not compare AR-Miner with CLAP in the previous steps (*i.e.,* the categorization and clustering of reviews), since the categories exploited by the two techniques are different, with AR-Miner limiting its categorization to *informative vs non-informative* reviews and CLAP looking for *bugs reports* and *suggestions for new features.* Instead, both techniques aim at prioritizing groups of reviews based on their "importance" for developers (*i.e.,* their relevance when working on a subsequent release). The prioritization applied by AR-Miner focuses on the reviews classified as *informative* and it is based on a weighted sum of three factors: (i) the number of reviews in the group (cluster), (ii) the average rating of the reviews in the group, and (iii) the temporal distribution of reviews (more recent reviews are considered more important). Since AR-Miner is not available[3], we reimplemented its prioritization feature, and tuned the weighting parameters as reported in [14]. Then, we applied AR-Miner on the same set of clusters prioritized by CLAP. In particular, we considered as *informative*, the reviews that were manually tagged as *bug reporting* or as *suggestion for new feature* and as cluster to prioritize those manually defined for these two categories of reviews (*i.e.,* exactly the same clusters prioritized in this evaluation by CLAP). Then, we compare the Area Under the Curve (AUC) for both techniques. We use AUC as AR-Miner produces a ranked list whereas CLAP produces a classification, hence it is not possible to directly compare precision and recall values. As expected, CLAP obtained a higher AUC when prioritizing bugs (0.86), while the AUC is lower (0.81) when prioritizing features. However, in both cases, the AUC achieved by CLAP is much higher than the one achieved by AR-Miner when prioritizing the same set of informative reviews (0.51). Note that the prioritization step of the two techniques has been compared exactly on the same set of manually created clusters of informative reviews. The only difference is that CLAP separately prioritizes *bug reports* from *suggestions for new features*, while AR-Miner prioritizes all the informative reviews as a whole (this is why we only have one value of AUC for it).

RQ$_4$: Would actual developers of mobile applications consider exploiting CLAP **for their release planning activities?** In order to answer our last research question, we qualitatively discuss the outcomes of the semi-structured interviews we conducted with project managers of three Italian companies aimed at analyzing the practical applicability of CLAP in a real development context.

Nicola Noviello, Project manager @ Next [5]. Nicola answered our first question (*i.e.,* usefulness of user reviews) with "absolutely yes", specifying that before planning a new release of an app, the developers of his company manually analyze the app reviews to identify critical bugs or feature recommendations. Nicola also confirmed that such a task is time consuming: when planning the release 2.0 of the app Unlikely quotes [7] "*A developer spent two days in analyzing more than 1,000 reviews. While the need to fix some bugs and to implement some features was easily spotted due to the fact that they were reported (required) by several users, there were also interesting features and critical bugs hidden in a large amount of non informative reviews. I strongly believe that* CLAP *would have sensibly reduced the effort we spent to identify such reviews.*" Nicola also positively answered to our questions related to the completeness of the categories of the reviews considered by CLAP and the factors it uses for prioritization ("yes" and "absolutely yes", respectively). Concerning the review categories considered by CLAP, Nicola suggested an additional category that could be considered: "*reviews referring to the app sales plan*". Nicola considers these reviews "*very important*" and he explained that "*the version 2.0 of the app Unlikely quotes was released both in a free and non-free versions, with the latter introducing some features for which the users explicitly claimed (in their reviews) that they will to pay for having such functionalities.*"

[3]We contacted the authors on July 29, 2015 and they confirmed that the tool is not publicly available.

Thus, user reviews could not only be useful to plan *what* to implement in the next release of an app, but also to define the sale strategies. Finally, Nicola was really enthusiastic about CLAP and he will be happy to use it in his company. Indeed, he considers the tool highly usable and ready to the market. He also pointed out two possible improvements for CLAP. First, it would be useful to make the tool able to store and analyze user reviews coming from different stores (*e.g.*,, Google Play and Apple App Store): "*putting together reviews posted by users running the app on different platforms could be important to discriminate between bugs strongly related to the app from those lying in the server-side. For example, if the bug is reported by both Android and iOS users, it is very likely that the bug is in the services exploited by the app, rather than in the app itself.*" Also, Nicola suggested to integrate in CLAP a mechanism that allows to read and analyze "*the reviews of competitive apps in order to identify features for my app that are not explicitly required by my users, but that have been suggested by users of competitive apps. In other words, I do not want to listen only to my users but also the users of competitive apps!*" Clearly, this would require the implementation of techniques to automatically identify similar apps. We consider this point as part of our future work agenda.

Giuseppe Socci, Project manager @ Genialapps [3]. As well as Nicola, Giuseppe answered "absolutely yes" to our first question related to the usefulness of user reviews: "*Very often reviews are informative and useful to understand which are the directions for new releases. I usually analyze the reviews manually and such a task is really time consuming. In the first year of life of our app Credit for 3 [1], I analyzed more than 11,000 reviews, dedicating six or seven hours per week to this specific task. However, keep up with the reviews helps a lot in making the app more attractive.*" Giuseppe also answered "absolutely yes" to our second question related to the completeness of the review categories: "*When I manually analyze the user reviews I classify them in exactly the same categories.*" Instead, Giuseppe answered "yes" to the question related to the completeness of the factors used to prioritize the bugs and the new features: "*While the exploited factors are reasonable, in my experience I also implemented several features and fixed some bugs that require few hours of work even if they were reported by just one person who is also already happy about the app. For instance, a user of the app Happy Birthday Show [6] rated the app with five stars, and requested to change the color of some buttons. Such a request required just a couple of hours of work. Thus, I decided to implement it. Considering the change impact of a new request or a bug fix might make the prioritization even more useful*". In addition, Giuseppe highlighted that the prioritization of the new features should take into account the kind of revision to perform, *i.e.,* minor or major revision: *If a major revision is planned, I tend to include as many feature requests as possible. Instead, if I am working on a minor revision, I really look for the most important feature requests to include (those having the highest payoff). In this case, the factors considered by* CLAP *in the prioritization are certainly valid.* Finally, Giuseppe answered positively ("absolutely yes") to our last question and he is willing to use CLAP as a support for the release planning of his future apps. The only showstopper for the application of CLAP in Genialapps is that most of the user reviews are written in languages different from English (*e.g.,* Spanish, Italian, French). We are currently adapting the tool aiming at making it multi-languages by exploiting automatic translation tools.

Luciano Cutone, Project manager @ IdeaSoftware [4]. While Luciano considers the user reviews useful for planning new releases, in his company, in general, user reviews are not analyzed. The reason is simple. IdeaSoftware usually develops app on commission. Thus, instead of considering user reviews, the developers of IdeaSoftware implement the features and fix the bugs required by their customers. Despite this, Luciano claimed that "*some of the features and bug fixes required by the customers of our apps were derived from the (in)formal analysis of user reviews.*". Luciano answered "yes" to the questions related to the completeness of the review category and the factors used to prioritize the bugs and the new features. However, he also noticed that the tool could be more usable if a criticality index is provided for each feature and bug. Specifically, "*instead of having features/bugs classified as high and low priority, I would like to see a list of features/bugs to be implemented ranked according to a criticality index ranging between 0 (low priority) and 1 (high priority). This would provide a better support for release planning especially when the number of features/bugs classified as high priority is quite high and I do not have enough resources to implement all of them.*" Finally, Luciano claimed that the tool seems to be useful "*especially when a high number of reviews needs to be analyzed*" and he is willing to use the tool in his company for analyzing the user reviews of the apps they plan to develop for the mass market (as opposed to those they currently implemented on commission for specific customers). Luciano also suggested to capture more information on how and when feature requests and bug fixes clusters have been implemented: "*For each cluster I would like to store the version of the app in which I implemented it. In this way I can maintain in* CLAP *the revision history of my apps and I could automatically generate release notes for each version*".

5. THREATS TO VALIDITY

Threats to *construct validity* are mainly related to imprecisions made when building the oracles used in the first three research questions. As explained in Section 3, the manual classifications performed for $\mathbf{RQ}_1$ and $\mathbf{RQ}_3$, as well as the golden set clusters for $\mathbf{RQ}_2$ have been performed by multiple evaluators independently, and their results discussed to converge when discrepancies occurred.

Threats to *internal validity* concern factors internal to our study that could have influenced our findings. One threat is related to the choice of the machine learning algorithm (Random Forest). As explained in Section 2 we have experimented various approaches and chosen the one exhibiting the best performance, but we cannot exclude that machine learners we did not consider (or different settings of the algorithm) could produce better accuracy. Similar considerations apply for the clustering algorithm. The ϵ parameter of the DBSCAN algorithm has been chosen using the tuning explained in Section 2.2. For the comparison with AR-Miner review prioritization, we use default weights reported in the paper [14], but it could be the case that they are not the most suited ones for our dataset. Finally, we are aware that planning the next release is a very complex process which involve different factors. Therefore, the prioritization simply based on the factors we considered in CLAP is only a recommendation that need to be complemented by factors related to the expertise and experience of software engineers.

Threats to *external validity* concern the generalization of our findings. In the context of $\mathbf{RQ}_1$ we chose to select the 1,000 reviews from a high number of apps (200) instead that from just one or two apps to obtain a more general model. Indeed, training the machine learner on reviews of a specific app_i would likely result in a model effectively working on app_i's reviews, but exhibiting low performances when applied on other apps. Still, while we tried to assess our approach on a relatively large and diversified set of apps, it is possible that results would not generalize to other apps, *e.g.,* those developed for other platforms such as iOS or Windows Phone, or the approach adaptation to reviews written in languages different from English might not exhibit the same performances we obtained.

6. RELATED WORK

Several works have focused the attention on the mining of app reviews with the goal of analyzing their topics and content [18, 21, 22, 24], the correlation between rating, price, and downloads [19], and the correlation between reviews and ratings [24]. Also, crowdsourcing mechanisms have been used outside the context of mobile development for requirements engineering, for example to suggest product features for a specific domain by mining product descriptions [16], to identify problematic APIs by mining forum discussions Zhang and Hou [31], and to summarize positive and negative aspects described in user reviews [20].

Due to lack of space, we focus our discussion on approaches aimed at automatically mining requirements from app reviews. Galvis and Winbladh [12] extract the main topics in app store reviews and the sentences representative of those topics. While such topics could certainly help app developers in capturing the mood and feelings of their users, the support provide by CLAP is wider, thanks to the automatic classification, clustering, and prioritization of reviews.

Iacob and Harrison [21] provided empirical evidence of the extent users of mobile apps rely on reviews to describe feature requests, and the topics that represent the requests. Among 3,279 reviews manually analyzed, 763 (23%) expressed feature requests. Then, linguistic rules were exploited to defined an approach, coined as MARA to automatically identify feature requests. Linguistic rules have also been recently exploited by Panichella *et al.* [26] to classify sentences in app reviews into four categories: Feature Request, Problem Discovery, Information Seeking, and Information Giving. CLAP, differently from MARA [21] and the approach by Panichella *et al.* [26], also provides clustering and prioritization functionalities to help the developers in planning the new release of their app. In our classification, we only consider categories relevant to the subsequent clustering and prioritisation.

Chen *et al.* [14] pioneered the prioritization of user reviews with AR-Miner, the closest existing approach to CLAP. AR-Miner automatically filters and ranks informative reviews. Informative reviews are identified by using a semi supervised learning-based approach exploiting textual features. Once discriminated informative from non-informative reviews, AR-Miner groups them into topics and ranks the groups of reviews by priority. The main differences between AR-Miner and CLAP are:

1. Bug/new feature reviews vs. informative/non-informative reviews. CLAP explicitly indicates to developers the category to which each review belongs (*e.g.,* "bug report" *vs* "suggestion for new feature"), while AR-Miner only discriminates between "informative" and "non-informative" reviews. Clearly, this different treatment also affects the grouping step. Indeed, while in AR-Miner a specific topic (*e.g.,* a topic referred to a specific app's feature) could indicate both suggestions on how to improve the feature as well as bugs reports, in CLAP the review clustering is performed separately between the different review categories. Also, while both techniques exploit textual features to categorize reviews, CLAP introduces a set of pre-processing steps (*e.g.,* n-grams extraction, negations management, customized synonyms list) that, as shown in Table 2, help in substantially increase the classification accuracy.

2. Recommending next release features/fixes vs. ranking reviews. CLAP exploits a machine learner to prioritize the clusters to be implemented in the next app release. This allows our approach to learn from the actual decisions made by developers over the change history of their app (see also the next point). On the opposite, AR-Miner ranks the importance of reviews based on a prioritization score, *i.e.,* a weighted sum of "prioritization factors". As shown in our evaluation, CLAP outperforms AR-Miner in predicting the items that will be implemented by developers in the next release of their app. Above all, since CLAP recommends reviews to be addressed in the next release based on the past history, it would be able to weigh different features of the prediction model differently for different apps and in general for different contexts. Finally, the bug/feature classification permits the use of different prioritization models for different kinds of change requests.

7. CONCLUSION AND FUTURE WORK

This paper described CLAP, a tool supporting the release planning activity of mobile apps by mining information from user reviews. The evaluation of CLAP highlighted its (i) high accuracy (86%) in categorizing user reviews on the basis of the contained information (*i.e., bug report*, *suggestion for new feature*, and *other*), (ii) ability to create meaningful clusters of related reviews (*e.g.,* those reporting the same bug)—77% of MoJoFM, (iii) accuracy (~72%) in recommending the features to implement and the bugs to fix in sight of the next app release, and (iv) suitability in industrial contexts, where we gathered very positive qualitative feedbacks about CLAP.

Such qualitative feedbacks will drive our future work agenda, aimed at improving CLAP with novel features and in particular: (i) the identification of similar apps in the store with the goal of mining user reviews from competitive apps; (ii) the multi-store support; and (iii) the automatic translation of reviews in English to overcome the current language limitation of CLAP.

Acknowledgements

The authors would like to thank all the people involved in the evaluation of CLAP. Bavota is supported by the Free University of Bozen-Bolzano through the RPMA project (IN2026).

8. REFERENCES

[1] Credit for 3. https://itunes.apple.com/it/app/credito-per-tre-soglie-in/id376583617?mt=8.

[2] English stopwords. https://code.google.com/p/stop-words/.

[3] Genial apps website. http://www.genialapps.eu/portale/.

[4] Ideasoftware website. http://lnx.space-service.it.

[5] Next website. http://www.nextopenspace.it/.

[6] Sing happy birthday songs. http://happybirthdayshow.net/en/.

[7] Unlikely quotes. https://itunes.apple.com/it/app/citazioni-improbabili-2.0/id555656654?mt=8.

[8] Weka. http://www.cs.waikato.ac.nz/ml/weka/.

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

[10] G. Bavota, M. L. Vásquez, C. E. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. The impact of API change- and fault-proneness on the user ratings of Android Apps. *IEEE Trans. Software Eng.*, 41(4):384–407, 2015.

[11] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

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

[13] N. Chen, S. C. Hoi, S. Li, and X. Xiao. Simapp: A framework for detecting similar mobile applications by online kernel learning. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, pages 305–314. ACM, 2015.

[14] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang. AR-miner: Mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 767–778, 2014.

[15] Digi-Captial. Mobile internet report q1 2015. http://www.digi-capital.com/reports.

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

[17] M. Ester, H. Kriegel, J. S, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pages 226–231, 1996.

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

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

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

[21] 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)*, pages 41–44, 2013.

[22] 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*, (2-3):103–134, 2014.

[23] G. A. Miller. WordNet: A lexical database for English. *Commun. ACM*, 38(11):39–41, 1995.

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

[25] F. Palomba, M. Linares-Vásquez, 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 *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME 2015, page To appear, 2015.

[26] 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 *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME 2015, page To appear, 2015.

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

[28] R. Socher, J. Bauer, C. D. Manning, and A. Y. Ng. Parsing With Compositional Vector Grammars. In *ACL*. 2013.

[29] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta. Replication package. http://www.inf.unibz.it/~gbavota/reports/app-planning.

[30] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 194–203, 2004.

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