MARAM: Tool Support for Mobile App Review Management

Claudia Iacob
School of Computing
University of Portsmouth
Portsmouth, UK
claudia.iacob@port.ac.uk

Shamal Faily
Department of Computing &
Informatics
Bournemouth University
Poole, UK
sfaily@bournemouth.ac.uk

Rachel Harrison
Department of Computing &
Communication Technologies
Oxford Brookes University
Oxford, UK
rachel.harrison@brookes.ac.uk

ABSTRACT

Mobile apps today have millions of user reviews available online. Such reviews cover a large broad of themes and are usually expressed in an informal language. They provide valuable information to developers, such as feature requests, bug reports, and detailed descriptions of one's interaction with the app. Due to the overwhelmingly large number of reviews apps usually get associated with, managing and making sense of reviews is difficult. In this paper, we address this problem by introducing MARAM, a tool designed to provide support for managing and integrating online reviews with other software management tools available, such as GitHub, JIRA and Bugzilla. The tool is designed to a) automatically extract app development relevant information from online reviews, b) support developers' queries on (subsets of) the user generated content available on app stores, namely online reviews, feature requests, and bugs, and c) support the management of online reviews and their integration with other software management tools available, namely GitHub, JIRA or Bugzilla.

CCS Concepts

•Software and its engineering → Software maintenance tools;

Keywords

Online reviews; mobile applications; querying

1. INTRODUCTION

Mobile app development today is interlinked with user generated online content associated to apps, namely online reviews. App users use mobile app reviews for a large number of purposes, from reporting bugs in a specific version of an app to asking for new features in future versions of the app [12]. This results in valuable design and development information (such as bugs or user requirements) being available in between the lines of such reviews. Based on [13], a quarter of the feedback provided by users through reviews is represented by feature requests, direct requests coming from users for changes in the app's functionality or additions of features or customizations. In addition to that, 10% of the feedback reported by users is used to provide details related to bugs, brief descriptions of malfunctioning of the app or of particular features of the app. In [12], authors introduce an approach for integrating user reviews in the overall mobile app development processes, by feeding a) the feature requests reported through reviews in the design phase of the development, and b) the bugs in the testing phase of the app development. Such an approach is defined to bypass traditional software engineering methods requiring direct access to users for both requirements elicitation and usability evaluation and to make use of the overwhelmingly large amount of feedback coming from direct users of apps and made available on app stores.

However, currently, two main challenges make it difficult for this approach to be used on a large scale. First, the user interaction available on app stores provides little in the way of means for making sense of reviews. Currently, the reviews of an app can only be visualized in blocks of four or five (Figure 1). The only tool available on app stores for making sense of these reviews is sorting them using one of three criteria: 1) the date they were posted on (allow the recently posted reviews to be displayed first), 2) the rating they are associated with (allow the reviews associated with a higher rate to appear first), and 3) the level of helpfulness they are associated with (allow the reviews rated more helpful to appear first). This, however, does not address the problem as the visualization of reviews in limited blocks is maintained. An app such as SwiftKey [7], for example, has 1,976,396 reviews, so visualizing these reviews in blocks of four is not feasible.

SwiftKey is not an exceptional case as apps usually have millions of online reviews; even with better visualizations of the reviews, manual inspection of all of them is impractical. This second challenge has been partially addressed in [11], where MARA, a tool able to automatically extract feature requests from online reviews, was introduced. However, for an app like SwiftKey for example, the tool identified almost half a million genuine feature requests. The effort of making sense of such data is still considerable, if not impossible to achieve.

Through this work, we aim to address both of these challenges by introducing MARAM (Mobile App Review Analysis and Management), a tool designed to support mobile app developers interacting with online reviews in making sense of the overwhelmingly large number of reviews available on app stores by querying and integrating them with other software management tools available, namely GitHub [6], JIRA [1] or Bugzilla [2]. MARAM preserves the functionality of MARA by automatically extracting feature requests and bugs, and it extends it by a) allowing the developer to run queries on the set of reviews, feature requests, or bugs, and b) allowing the developer to select items returned by a query and export them to GitHub or JIRA as issues, or to Bugzilla as bugs.

This paper introduces the tool and it is structured as follows: Section 2 presents the requirements, design, architecture and details of the tool's implementation; Section 3 evaluates the design of MARAM by illustrating three possible scenarios of use of the tool's use in mobile app development; Section 4 is an overview of previous work done in the area. The paper concludes with ideas of future work and the broader implications of MARAM in mobile app development.

2. MARAM

This section introduces MARAM, a tool designed to support app developers in managing the online reviews of the mobile applications of interest to them. We describe the requirements of the tool, its overall design and architecture, and details of its implementation.

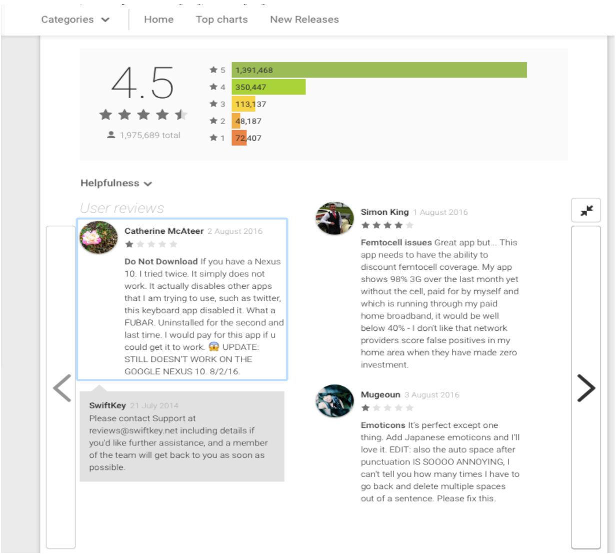
2.1 Requirements

MARAM is an extension of MARA, a tool introduced in [11] and designed to automatically extract feature requests from online reviews of mobile applications. In its current form, MARAM, is designed as a broader scope IDE able to support app developers in managing the user generated content (i.e. user reviews) associated with mobile apps and available on app stores. The requirements of the tool include:

a. Automatically extract app development relevant information from online reviews. In this respect, MARAM preserves the functionality of MARA [11] and automatically extracts feature requests and bugs from online reviews.

b. Support developers' queries on (subsets of) the user generated content available on app stores, namely online reviews, feature requests, and bugs. This feature allows the developer to use the tool as a search engine on the corpus of reviews, returned feature requests, or bugs. For example, if interested in all the comments users provided with respect to security, a developer can formulate a query containing the word 'security' and get a list of all feedback relevant to the query, ranked by the level of relevance.

c. Support the management of online reviews and their integration with other software management tools available, namely GitHub, JIRA or Bugzilla. It is rare that a piece of software, be it a mobile app, is developed by one individual in isolation. More often than not, it is a team of developers working together that contribute to the development of an app. In this context, software management platforms that support collaboration become crucial to the success of the app. MARAM provides a link between the app store (as a proxy of the users) and such software management platforms.



Catherine McAteer 2 August 2016

Do Not Download If you have a Nexus 10. I tried twice. It simply does not work. It actually disables other apps that I am trying to use, such as twitter, this keyboard app disabled it. What a FUBAR. Uninstalled for the second and last time. I would pay for this app if u could get it to work. 😱 UPDATE: STILL DOESN'T WORK ON THE GOOGLE NEXUS 10. 8/2/16.

SwiftKey 21 July 2014

Please contact Support at reviews@swiftkey.net including details if you'd like further assistance, and a member of the team will get back to you as soon as possible.

Simon King 1 August 2016

Femtocell issues Great app but... This app needs to have the ability to discount femtocell coverage. My app shows 98% 3G over the last month yet without the cell, paid for by myself and which is running through my paid home broadband, it would be well below 40% - I don't like that network providers score false positives in my home area when they have made zero investment.

Mugeoun 3 August 2016

Emoticons It's perfect except one thing. Add Japanese emoticons and I'll love it. EDIT: also the auto space after punctuation IS SOOOO ANNOYING, I can't tell you how many times I have to go back and delete multiple spaces out of a sentence. Please fix this.

Figure 1: Online reviews user interface on Google App Store

In the case of GitHub, for example, the community of contributors to the development of an app uses issues to communicate contributions, report bugs, ask for clarification, or simply coordinate efforts. Reporting a bug or asking for a feature to get implemented are examples of issues often seen on GitHub. However, the exact same information is reported on app stores through reviews, so automatically translating reviews (or subparts of reviews) into issues brings all development data together in one place, making it available to the larger community.

2.2 Design and Architecture

MARAM is designed around four main parts: 1) Review collection engine (RCE), 2) Review analysis engine (RAE), 3) Review query engine (RQE), and 4) Review export engine (REE) (Figure 2). The RCE acts as the interface with an app store. The RAE manages the analysis of the retrieved reviews by a) automatically extracting the feature requests users ask for through reviews, and b) automatically extracting the bugs that users report on through reviews. The RQE manages the interaction between the app developer and the corpus of reviews, allowing the developer to formulate and run queries on the reviews. The REE supports relating the online reviews available for an app with similar software management tools available, including GitHub, JIRA and Bugzilla.

As a whole, MARAM works as follows:

1. The RCE communicates with the app store, retrieving the online reviews available for an app at a given time. All such reviews and their metadata are extracted from the app store.

2. The RCE stores all the data scrapped from the app store in a database (Raw Reviews Database), without running any pre-processing on the reviews at this point.

3. The RAE takes as input the reviews in the format they were retrieved from the app store.

4. The RAE uses the content of the Raw Reviews Database to: a) format the content of reviews to facilitate further processing (i.e. identify the content of each review and the meta-data associated to it), and b) extract feature requests and bugs from reviews' content.

5. The formatted reviews and the extracted data are inputted into the RQE.

6. The RQE generates corresponding maps for the reviews, feature requests, and bugs. The maps index the content of the reviews and the extracted data in order to facilitate querying.

7. The RQE takes as input a developer's query. This query is a set of words. Although no restrictions are imposed on the query the developers can submit, the quality of the results of the querying process will differ depending on the number of words used in the query.

8. The RQE generates the results to the developer's query.

9. The developer can select a subset of the results or the results as a whole and his/her selection becomes the input of the REE.

10. The REE uses the items selected by the developer and automatically exports them to the software management platform selected by the developer.

2.2.1 Review Collection Engine (RCE)

The role of the RCE is to retrieve all the online reviews available for a given app in an app store. A developer is able to login to the RCE and retrieve all the reviews available on the app store for the app(s) s/he manages. The reviews are stored in a local database and, at this stage, no preprocessing is required. Every review is associated with meta-data including the date it was posted, the device it is associated with, the version of the app the reviews is for, the rating, a title, and the content of the actual review.

2.2.2 Review Analysis Engine (RAE)

The role of the RAE is to automatically extract from reviews information relevant to the app development process, namely feature requests and bugs. The RAE takes as input the reviews database storing all the reviews and the meta-data associated to them in their raw version and outputs the corpus of feature requests and the corpus of bugs automatically extracted from the reviews. Although in its current version the RAE only extracts feature requests and bugs, for future versions of this tool we will explore the possibility of automatically extracting other types of information, such as security related comments, usability related comments, comparisons to other similar apps, or recommendations of similar apps.

2.2.3 Review Query Engine (RQE)

The role of the RQE is to support app developers in running queries on the corpus of reviews, the feature requests, or/and the bugs. In the context of MARAM, we use the word 'artifact types' when referring to either reviews, feature requests, or bugs. Also, when referring to a review, a feature request, or a bug, we use the word 'artifact'. For allowing queries on the data, the RQE builds a map for each artifact type by following the steps detailed below:

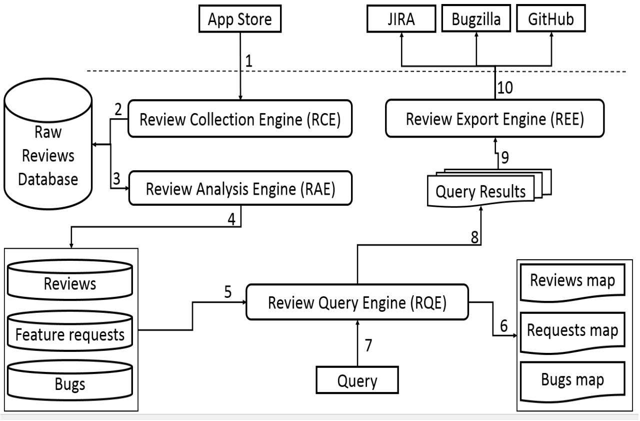


Figure 2: MARAM Architecture

a. Each artifact, R, is split into words (spaces and punctuation marks are used as separators): w_1, w_2, ... , w_n.

b. Each artifact is associated with the stemmed root [14] of each of the words it is formed of: R → { w'_1, w'_2, ... , w'_n}

c. All stop words are excluded from all associations.

The RQE is designed to run parameterized queries, defined as:
Q = {*app*, *version*, *device*, *artifact_type*, *max*, *summary*, *text*} ,
where:

a. *app* is the name of the app whose reviews are queried.

b. *version* is the version number of the app considered for the query (for example, a developer can query only the reviews of the latest version of his/her app)

c. *device* is the name of the device considered for the query (for example, a developer can query only those reviews written for a particular type of phone)

d. *artifact_type* is one of the three: reviews, feature requests, or bugs (for example, a developer can choose to run a query only on the corpus of bugs)

e. *max represents* the maximum number of results displayed at once

f. *summary* indicates whether the query results should be summarized (i.e. similar results displayed once with a higher weight)

g. *text* is the text of the query.

Feature requests	*add, allow, complaint, could, hope, if only, improvement, instead of, lacks, look forward to, maybe, missing, must, needs, please, prefer, request, should, suggest, waiting for, want, will, wish, would*
Bugs	*closes, can't/cannot/couldn't, don't/doesn't/does not/didn't, not, fail, crashes, lag, error, stopped, freeze, won't/will not, not able to, locks, unable to, erased, eliminated, impossible to, impossible to, glitches, reboot, annoying, problems with, bugs, causes, lost, restart, horrible, slows down, reloads, switches off, timeout, wouldn't, malfunction*

Table 1: Keywords for expressing feature requests and bugs

2.2.4 Review Export Engine (REE)

The role of the REE is to allow the export of selected query results to formats supported by other software management tools, such as GitHub, JIRA or Bugzilla. The developer is able to select one or more results returned for his/her query and represent the selected results as issues recognized by GitHub and JIRA, or bugs recognized by Bugzilla. For example, MARAM uses the GitHub API to post a new issue to a repository that has been predefined within MARAM. The issue contains information about the app's name, version, and device it has been used on together with information about the actual issue, namely a title (generated based on the text of the query) and a short description (generated based on the results returned by the query). The current version of the tool supports GitHub integration only.

2.3 Implementation

MARAM is a prototype developed as a desktop based application; this is currently implemented in Java. This section provides implementation details for the components of MARAM.

2.3.1 Review Analysis Engine

The implementation of the RAE is based on the identification of a set of linguistic rules that encapsulate, syntactically and semantically, the users' expressions of feature requests and bugs. For example, feature requests are more prone to be expressed in sentences such as *'Adding an exit button would be great'*. Similarly, major bugs are often expressed as *"forced shutdown many times"*, *"crashes at the end of the race"*, *"fails and loses entire workout"*, *"app is not working"*, or *"it won't open"*. Medium bugs are less intense and more specific, focusing on a particular aspect of an app - *"does not show any tracks of previous workouts"*, *"the Slovenian dictionary is missing even the basic words"*, *"miles do not add right"*, *"I did not create a password. Yet it asks me for my password"*. Minor bugs are merely observations related to a feature of an app - *"slight issue with GPS data"*, *"a little bug on the Persian keyboard"*, *"text overlaps for lower percentages"*.

We defined the two sets of linguistic rules (i.e. one for feature requests and one for major and medium bugs) based on the analysis of the manually identified feature requests and bugs in the random sample of reviews described in [11].

Feature Requests	<request>*would make it* <COMPARATIVE-ADJ>
	(<*SB*>) (<*ADV*>) *wish there was* <request>
	please include <request>
	could use (more) <request>
	add the ability to <request>
	(the only thing) missing <request>
	needs the ability to <request>
Bugs	*stopped* {downloading, running, syncing}
	{don't, doesn't, none, didn't} *work (since the update)*
	impossible to <action>
	will not (even) {open, start, execute, activate, install, show up}

Table 2: Examples of linguistic rules for defining feature requests and bug

The analysis comprised of the following steps:

a. Associate each sentence labeled as a feature request or a bug with a keyword, which denotes the sentence as a feature request or a bug. In the case of feature requests, we identified 24 keywords. In the case of bugs, we identified 33 keywords (Table 1).

b. Filter all those sentences containing at least one keyword and define the contexts (i.e. fragments of sentences) in which these keywords point to actual feature requests or bugs. Examples of such contexts are: "an exit button *would* be fantastic", "adding more icons *would* be great", "tips and math support *would* also be nice".

c. Abstract contexts into linguistic rules such as: "(*adding*) *<request> would* (*<ADV>*) be *<POSITIVE- ADJECTIVE>*". The word "adding" is optional, while ADV and POSITIVE-ADJECTIVE can be replaced by any adverb or positive adjective, respectively. We defined 237 linguistic rules for capturing feature requests, and 74 linguistic rules to capture bugs. Examples of such linguistic rules are presented in Table 2.

Identifying a feature request or a bug translates into identifying contexts, which match at least one linguistic rule corresponding to either feature requests or bugs.

In the context of evaluating feature request and bug mining, we consider the following elements for the definition of the performance metrics: a) true positives (TP) as the correctly returned feature requests/bugs, b) false positives (FP) as the returned results which are not actual feature requests/bugs, c) true negatives (TN) as the non-feature requests/bugs not returned as results, d) false negatives (FN) as actual feature requests/bugs not returned as results. Precision is the ratio between the returned results, which are actual feature requests/bugs and the total number of returned results. Recall is the ratio between the returned results, which are actual feature requests/bugs and the total number of feature requests/bugs in the input. Lastly, Matthews Correlation Coefficient (MCC) is a value between -1 and 1, with MCC = 1 for a perfect predictor, MCC = 0 for a random predictor, and MCC = -1 for a perfect inverted predictor.

	Definition	Feature Requests	Bugs
P	$\frac{TPs}{(TPs+FPs)}$	0.85	0.91
R	$\frac{TPs}{(TPs+FNs)}$	0.87	0.89
MCC	$\frac{TPXTN-FPXFN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$	0.90	0.91

Table 3: Precision, recall, and MCC in RAE's evaluation

We evaluated the performance of the RAE on the sample of 136,998 reviews defined in [11], using precision (P), recall (R), and MCC as performance metrics (Table 3). In the case of feature requests, we randomly selected 3000 feature requests returned by the algorithm, and a human coder analysed the sample to check which were FPs. The task of the coder was to identify, for each result analysed, the actual request the user expressed. Overall, the P calculated for the sample was 0.85. In terms of recall and MCC, we used the reviews of a random chosen app as a sample for measuring the two metrics. For the 480 reviews sampled, the value of R was 0.87, and the value of MCC was 0.90. We used a similar approach in the case of bugs, with a P value of 0.91, R value of 0.89, and MCC of 0.91.

2.3.2 Review Query Engine

The implementation of the RQE is based on:

a. Generating an artifact map for a set of artifacts (i.e. reviews, feature requests, and bugs),

b. Computing the similarity score between a given query and each artifact in a given set, and

c. Ranking the artifacts in a given set based on their similarity score with the query.

```
Algorithm 1: GenerateArtifactMap
   input  : Artifacts[] $items$: reviews, feature requests, or bugs
   output: ArtifactMap $map$: $AM = \{(Artifact, Keywords[])\}$
1  for $ItemA \in items$ do
2      String[] $wordsA$ = tokenize($ItemA$);
3      RemoveStopWords($wordsA$);
4      StemEachWord($wordsA$);
5      $map$.add(($ItemA$,$wordsA$));
6  return $map$
```

Algorithm 1 describes the process of generating an artifact map for a given set of artifacts. The aim of the map is to associate each artifact with a set of representative keywords for the item. The process includes: tokenizing the text of an artifact (line 2), removing the stop words (line 3), stemming each word to its root (line 4), and associating the artifact with the obtained set of words (line 5).

When faced with a query Q = {*app*, *version*, *device*, *artifact_type*, *max*, *summary*, *text*}, the RQE performs the following steps (Algorithm 2):

1. Identify the list of artifacts, *artis*, the query refers to (i.e. all the artifacts of type *artifact_type* for the application *app*, in its version *version*, and for the device *device*). The corpus the query will be run on becomes *artis* (line 1).

```
Algorithm 2: RunQuery
    input  : $Q=\{app,version,device,artifact\_type,max,$
             $summary,text\}$
    output: Artifact[] $topResults$: top results returned by the
             query
1   $artis$ = returnArtifactFor($app$, $version$, $device$, $artifact$);
2   $AM$ = GenerateArtifactMap($artis$);
3   $AQ$ = new Artifact($text$);
4   $QM$ = GenerateArtifactMap($AQ$);
5   SimilarityScores[] $scores$ = new SimilarityScores[];
6   for $A \in artis$ do
7       $sim$ = computeSimilarity($A$,$AQ$);
8       $scores$.add(new SimilarityScore($A$,$AQ$,$sim$));
9   Sort $scores$ based on $similarity$;
10  if $summary == true$ then
11      return summary of top $max$ artifacts in
        $scores.ArtifactA$;
12  else
13      return top $max$ artifacts in $scores.ArtifactA$;
```

2. Generates an artifact map, *AM*, for *artis* (line 2).

3. Represents the query Q as an artifact (AQ), using the text of the query as a parameter (line 3).

4. Generates the artifact map, QM, for the artifact representing the query Q (line 4).

5. Calculates the similarity score between the artifact represented by the query (AQ) and each artifact in AM (lines 5-8). A similarity score is represented as *SimilarityScore* = {*ArtifactA*, *ArtifactB*, *sim*}, and is calculated based on the Jaccard Similarity coefficient:

   $$\text{JS(A, B)} = |\text{A} \cap \text{B}|/|\text{A} \cup \text{B}| \ (1)$$

   , where |A ∩ B| represents the cardinal of the intersection of the set of words associated to A and the set of words associated to B, and |A U B| represents the cardinal of the union of the set of words associated to A and the set of words associated to B.

6. Sorts the similarity scores and returns the top max results as either a summary or as a full list of results (line 9-13).

In the current version of the tool, the implementation of the summary generation for a query result is unavailable. However, the summarization process includes the following steps:

1. For each artifact, A, in the results returned by the query, identify the set of all the artifacts similar in meaning with A, i.e. *synA*. Two artifacts are similar if their similarity score is higher than a set threshold.

2. Associate each set *synA* with a weight calculated based on the number of the artifacts included in *synA*, and their respective similarity scores.

3. Generate a description for the set, *synA*, based on the keywords common to all the artifacts in *synA* and their order of appearance.

4. Add the description generated for *synA* together with its weight to the summary.

5. Sort the summary based on the weights of the sets *synA* with the sets weighting higher appearing the first in the summary.

6. Return the sorted summary as the summary of the query results.

2.3.3 Review Export Engine

The design and implementation of REE is based on the Abstract Factory pattern [5], where *ExportEngineFactory* objects are created based on the export target. For example, we implemented the class *GitHubIssueEngineExportFactory* to generate GitHub Issues from artifacts. This factory class is constructed with the URL of the export target, and the authentication credentials necessary to both access the target and import artifacts.

When faced with an Artifact A = *app*, *version*, *device*, *artifact_type*, *text*, the RQE performs the following steps (Algorithm 3):

1. Authenticates with the export target corresponding the URL using the credentials provided during the factory class' construction.

2. Constructs a name for the artifact *EAName*, based on the *app*, *artifact_type*, and *device*. For example, where *app*="App A", *artifact_type*="crashes", and *device*="Galaxy 1.1", the name would be "App A crashes on Galaxy 1.1".

3. Constructs the body of the artifact *EABody* based on the provided *text*. The body will be the form recognizable to the export target, e.g. JSON in the case of GitHub

4. Pushes the artifact to the export target using facilities provided by the target. For example, *GitHubIssueEngineExportFactory* uses the GitHub web services API to post new issues.

5. Returns the identifier *ArtifactId* for artifact created on the export target; this identifier is meaningful to the export target. For example, a GitHub Issue Id is returned by the algorithm implemented by *GitHubIssueEngineExportFactory*.

Algorithm 3: ExportArtifact

```
input  : $A=\{app,version,device,artifact\_type,text\}$
output: $ArtifactId$: identifier of the export artifact
$Authenticate()$;
$EAName =$
ConstructArtifactName($app,artifact\_type,device$);
$EABody =$ ConstructArtifactBody($text$);
$ArtifactId =$ PushArtifact($EAName,EABody$);
return $ArtifactId$;
```

3. USAGE SCENARIOS

The target audience for MARAM is the community of app developers maintaining apps, or developing them from scratch. The tool is as a mediating layer between an app store and software management tools, also allowing the developers to query the corpus of user generated content (namely online reviews) associated for the app they maintain or develop. An app developer logs into MARAM and selects one of the apps she is maintaining from a dropdown list, say SwiftKey Keyboard (Figure 3). For the app selected, she can also select a version number and a device in case she wants to restrict the query only to the artifacts associated with a specific version of the app and/or a specific type of device. Any query can be run on either the entire corpus of reviews, the corpus of feature requests, or the corpus of bugs. Some possible examples of queries include: *"all reviews for SwiftKey Keyboard app, version 3.1 on Galaxy phone that speak about 'performance' "*, *"all bugs of SwiftKey Keyboard where the user reports a crash"*, *"all feature requests for SwiftKey Keyboard users asked for in the previous version of the app"* . Such queries are needed in various contexts and at various phases of the app development cycle; we will focus below on three possible scenarios.

3.1 Scenario 1 - Design

During the design phase of an app, an app developer consults the reviews for feedback on the feature requests users ask for or comment on. In the context of MARAM, an app developer is able to:

a. Generate a report of all the feature requests reported for an app or for a specific version of the app.

b. Generate a report of all the feature requests related to a query reported for an app or for a specific version of the app. For example, the app developer can identify all the requests coming from users for a particular feature, settings preference, or aesthetic parameter (the shape or location of a control, for example).

c. Generate a report with the N most relevant feature requests to a query. For example, the developer can identify what are the top 10 requests for a particular control, or feature (i.e. registration).

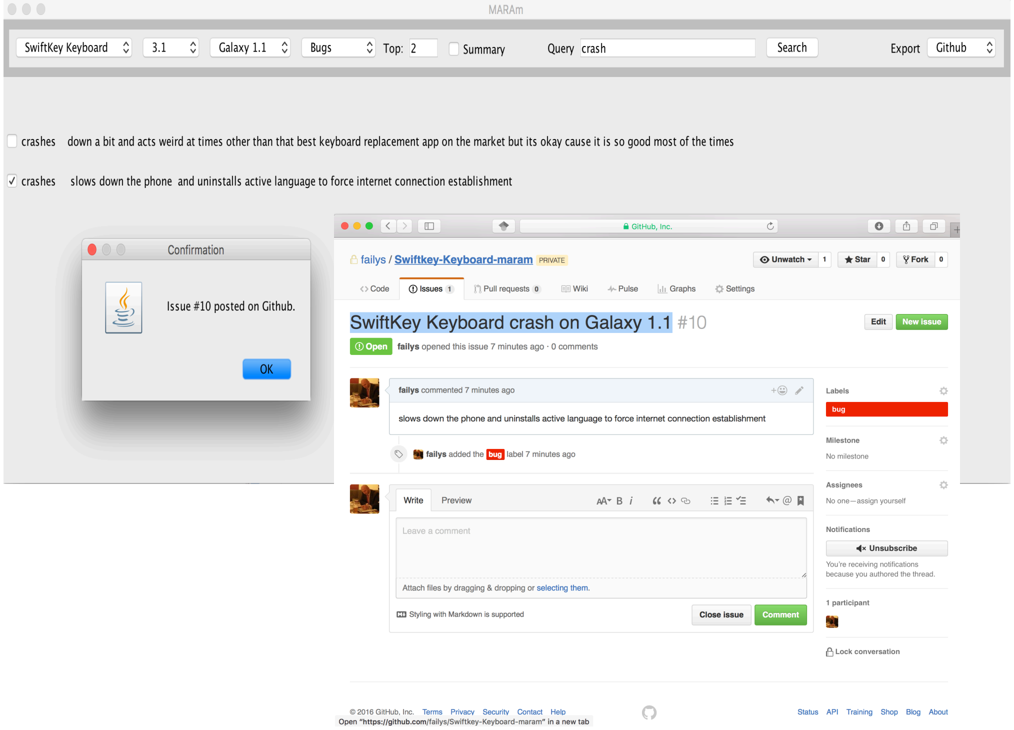
Based on these basic actions supported by MARAM more complex scenarios can be defined:

Jack, the lead developer for AppT, checks the features requests left unimplemented from the previous version of the app (i.e. version 3.2). For that, he will run a query for AppT, choosing version 3.2 and returning all the feature requests available for that version (i.e. the text of the query is blank). Similarly, he will run a query for the same app only choosing the latest version of the app, i.e. version 4.1. For identifying those feature requests left unimplemented, Jack will identify those requests returned for both queries. We assume that once a feature has been implemented in a given version of an app, users will no longer ask for it through online reviews.

3.2 Scenario 2 - Testing

During the testing phase of an app, an app developer consults the reviews for feedback on the malfunctioning the users report on with respect to the app or to specific features of the app. In the context of MARAM, an app developer is able to:

a. Generate a report of all the bugs reported for an app or for a specific version of the app.



MARAm

SwiftKey Keyboard | 3.1 | Galaxy 1.1 | Bugs | Top: 2 | □ Summary | Query crash | Search | Export Github

□ crashes   down a bit and acts weird at times other than that best keyboard replacement app on the market but its okay cause it is so good most of the times

☑ crashes   slows down the phone and uninstalls active language to force internet connection establishment

Confirmation

Issue #10 posted on Github.

OK

failys / Swiftkey-Keyboard-maram PRIVATE

SwiftKey Keyboard crash on Galaxy 1.1 #10

Open failys opened this issue 7 minutes ago · 0 comments

failys commented 7 minutes ago

slows down the phone and uninstalls active language to force internet connection establishment

failys added the bug label 7 minutes ago

Labels: bug

Milestone: No milestone

Assignees: No one—assign yourself

Notifications: Unsubscribe

You're receiving notifications because you authored the thread.

1 participant

Open "https://github.com/failys/Swiftkey-Keyboard-maram" in a new tab

Figure 3: MARAM Screenshot: GitHub integration

b. Generate a report of all the bugs related to a query reported for an app or for a specific version of the app. For example, the app developer can identify all the comments coming from users that are related to the crash of a specific feature (eg. storing data, remembering password, etc).

c. Generate a report with the N most relevant bug reports to a query. For example, the developer can identify what are the top 10 bug reports related to a specific task (eg. registration). This is designed to help prioritizing the developers' work.

Based on these basic actions supported by MARAM more complex scenarios can be defined in the context of testing an app:

> Jane, the lead developer for AppM, is interested in identifying all bugs reported for a beta version of AppM. For that, she will run a query for an app, choosing the beta version of the app as a parameter, and returning all the bugs available for that version. Jane is presented with either a list of bugs, or a summary of those bugs (similar bug reports are grouped together and presented as one with a higher weight). She can opt to go through the bugs screen by screen, showing a maximum number of bugs at once, or simply read the bugs related to a set of words or topics. Once presented with a list of bugs, the developer can choose one or more of them and generate BugZilla reports for them. These can further be imported into BugZilla and made publicly available.

3.3 Scenario 2 - Maintenance

Maintenance runs throughout the life of an app. In the current app ecosystem, maintenance goes hand in hand with development and both highly depend on user feedback. It is usual for an app to be deployed on an app store in its beta version and than extended and maintained at the same time based on feedback reported by users through reviews. Such feedback is not limited to feature requests and bugs and can include critical comments on specific features, comparisons with other similar tools, comments on the level of usability of the app, or on its value-for-money. MARAM supports queries on all types of comments and identifies those fragments of reviews most relevant to a given query. Some examples of such queries include:

- *'set up'*, which will identify all comments reported by users on the set up process of the app. Such comments may include positive or negative remarks, or they may describe glitches users faced during the set up process.

- *'the name of a different tool'*, which will identify all users' comments related to the tool, such as 'tool X does <feature> better', or 'why doesn't this tool do what tool X does when....'. This would give developers ideas for improvement and it would help them make better sense of what the user want in the context of an app's market.

Based on the basic feature developed for MARAM more complex scenarios of use can be defined in the context of app maintenance:

John, the lead developer for AppA, is interested in getting all user comments available for AppA that are related to a specific topic or theme, say ‘performance standards’ or ‘usability’. For that, he will run a query on an app using topic-sensitive words as the text of the query and the corpus of reviews as the selected artifact. The returned results might include positive or negative comments on the topic, comparative comments with other available apps, details of specific usage experiences users had with that app, and recommendations for other users. After reviewing the results of the query, John can decide to select one or more results and export them as issue recognisable in GitHub, thereby making them available to the larger community of developers.

4. RELATED WORK

App stores today are becoming increasingly powerful tools for app developers. A survey of app stores analysis for software engineering [15] identifies several uses of app stores in mobile app development, including API analysis on the Android platform [19, 17] app feature analysis [10], online review analysis [11, 8, 9], and the role online reviews can play in the broader mobile app development cycle [12]. Although all the studies on app reviews point out the usefulness and potential [16, 4] these artifacts can bring to mobile app development processes, little work has been done in providing developers with support in interacting with reviews, making sense of them, and incorporating them in their day-to-day job.

App stores provide very little in the way of navigating and making sense of reviews, forcing developers to browse millions of reviews manually, and a few at a time. The filters available for making sense of reviews are very limited. For example, Google app store allows its users to filter reviews by helpfulness, rating, or date, while the iTunes app store only shows three random reviews for any app. However, such interaction means are far too simplistic and they do not match the ways app developers design, test, or maintain their apps. App-store external tools to support developers in using online reviews include AR-Miner [3] and CLAP [18]. AR-Miner is designed to i) extract informative information from user reviews, ii) groups such information using topic modeling, iii) prioritize informative reviews based on a predefined scheme, and iv) visualize the groups of most informative reviews. CLAP is a similar tool designed to: i) categorize user reviews as bugs, requests or others, ii) cluster related reviews, and iii) prioritize the clusters of reviews to be implemented in a next release of the app. None of these tools supports querying the corpus of reviews and the only interaction available for the app developer includes deleting a review or changing its cluster.

5. CONCLUSIONS AND FUTURE WORK

This paper introduced MARAM, a tool dedicated to app developers and designed to support querying online reviews and their derived artifacts; the tool facilitates the integration of these artifacts with other software management tools. The tool is designed around four main parts, all supporting further extensions.

The RCE can act as a browser add-on, allowing a smoother interaction with a particular app store.

The RAE can be extended to automatically extract other types of information from reviews. Such information can include comments around usability, security, or comparative feedback between various apps.

The RQE can be extended to support alternative metrics for computing similarity scores, such as cosine similarity. It can also be extended to allow additional query parameters, including:

- Rating associated with artifact - e.g. running queries on all feature request extracted from reviews associated with 3 stars or more.
- Date - e.g. running queries on all reviews posted after a certain date or within a time interval.
- Length - e.g. running query on all reviews longer than 2 sentences.

The REE can be extended to support integration with additional software management tools and environments.

6. REFERENCES

[1] Atlassian. JIRA website. https://www.atlassian.com/software/jira, October 2016.

[2] Bugzilla project. Bugzilla website. https://www.bugzilla.org, October 2016.

[3] 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, New York, NY, USA, 2014. ACM.

[4] 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 *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’13, pages 1276–1284, New York, NY, USA, 2013. ACM.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[6] GitHub, Inc. GitHub web site. http://github.com, October 2016.

[7] Google Play. Swiftkey reviews. https://play.google.com/store/apps/details?id=com.touchtype.swiftkey, August 2016.

[8] X. Gu and S. Kim. What parts of your apps are loved by users? In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 760–770, Nov 2015.

[9] E. Guzman, M. El-Haliby, and B. Bruegge. Ensemble methods for app review classification: An approach for software evolution. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 771–776, Nov 2015.

[10] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: Msr for app stores. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR ’12, pages 108–111, Piscataway, NJ, USA, 2012. IEEE Press.

[11] 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, MSR '13, pages 41–44. IEEE Press, 2013.

[12] C. Iacob, R. Harrison, and S. Faily. Online Reviews as First Class Artifacts in Mobile App Development. In *Proceedings of the 5th International Conference on Mobile Computing, Applications, and Services (MobiCASE)*, pages 47–53. Springer, 2013.

[13] C. Iacob, V. Veerappa, and R. Harrison. What are you complaining about?: A study of online reviews of mobile applications. In *Proceedings of the 27th International BCS Human Computer Interaction Conference*, BCS-HCI '13, pages 29:1–29:6, Swindon, UK, UK, 2013. British Computer Society.

[14] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[15] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A Survey of App Store Analysis for Software Engineering. Technical Report RN/16/02, University College London, January 2016.

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

[17] S. Seneviratne, H. Kolamunna, and A. Seneviratne. A measurement study of tracking in paid mobile applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[18] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta. Release planning of mobile apps based on user reviews. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 14–24, New York, NY, USA, 2016. ACM.

[19] H. Wang, J. Hong, and Y. Guo. Using text mining to infer the purpose of permission use in mobile apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 1107–1118, New York, NY, USA, 2015. ACM.