Establishing Continuous App Improvement by Considering Heterogenous Data Sources

https://doi.org/10.3991/ijim.v15i10.20613

Simon André Scherr (✉), Steffen Hupp, Frank Elberzhager
Fraunhofer IESE, Kaiserslautern, Germany
simon.scherr@iese.fraunhofer.de

Abstract—Mobile apps have penetrated the market and are being used everywhere. Companies developing apps face increasing challenges such as short time to market or demand for high quality. Furthermore, the success of an app also depends on how users perceive its quality. Feedback provided by users influences other potential users and provides new opportunities for identifying features. Consequently, it is a valuable source of input for app developers with respect to product improvements. One form is textual feedback. This kind of feedback is usually distributed across various data sources. Therefore, it must be captured from these sources and put into one single pool of data before it can be analyzed. The analysis must take into account the peculiarities of the short release cycles and high change rate of features for mobile apps. In this article, we present User Echo Service (UES), which was built to address heterogeneous data sources. The aim of UES is to allow product managers to be able to be always up to date with the latest feedback data. Therefore, we have created an extensible architecture aimed at supporting different data sources and present the feedback collection scheduling system. This forms the prerequisite for subsequent analyses of the collected data. We discuss our solution and provide ideas for future development.

Keywords—Quality Assurance, Apps, User Feedback, Architecture, Product Improvement, Quality Monitoring

1 Introduction

Mobile apps have become an essential part of our daily life. For many people, their smartphone has become their primary electronic device. For app-developing companies, key challenges are high market pressure, the diversity of devices, and the often limited resources during development. However, users demand high quality (including great user experience) as well as features that support their tasks in a meaningful way.

Various approaches and techniques support the development with regard to some of these challenges, such as the use of MVPs for quick market entry, agile and lean development practices [1], increased automation of quality assurance activities, or different web services. In our research, we focus on analyzing and using user feedback to assure and improve app quality. Such feedback is often not considered systematically by developers, but can offer great value in terms of enabling a better understanding of customers, their real needs, and bugs discovered by users. In recent years, the role of user feedback for development has increased [2]. This also means that developers of apps need to consider feedback, as users and potential users are influenced by it. Bad feedback can have an impact on other potential users who are not using this app yet and may prevent them from using it. To make it easier for developers to consider feedback, we have developed an approach that captures usage feedback and textual feedback – the latter being the focus of this article.

For such an approach to be used, textual feedback must be gathered systematically and quickly. Artur Strzelecki [3] notes that ratings and reviews play an important role as success factor for mobile apps in the stores. Users frequently give feedback and also change their minds based on recent changes of the product [4]. Therefore, data should be continuously gathered and evaluated in order to enable reaction to fast trends and implementation of changes within short release cycles [5]. As many different sources exist nowadays where users can provide feedback (e.g., app stores, social media), another central requirement is that different sources must be taken into account. Research has shown that considering just one type of source lacks a significant portion of feedback [6]. As far as textual feedback for apps is concerned, these could be the Apple App Store, Google Play, or social media sources like Twitter. The data from these sources must be unified to create a holistic view of the users' perspective on the product. In the best-case feedback would be captured and analyzed continuous by considering all relevant data sources.

In this article, we therefore focus on the following research question:

How to continuously gather feedback from different sources in order to provide a foundation for user feedback analyses representing a holistic view on this feedback?

In this article, we introduce User Echo Service (UES), which implements these requirements. This means that we focus on data acquisition and not on subsequent feedback analysis steps. UES offers continuous user feedback monitoring and includes new sources such as different app stores and social media. Our tool was further developed to cope with the new requirements. We embed our work into the current state of the art (Section 2) and discuss our solution in Section 3. Section 4 provides a mapping of our data model to different sources, while Section 5 discusses our solution. Section 6 closes with a summary and an outlook on follow-up work.

2 Foundations and Related Work

2.1 User feedback and the crowd

The rise of central app stores and smartphone platforms has enabled users to submit reviews that can be viewed by other (potential) users and developers and to view the entries of other users. As a result, app store reviews and ratings have become an important factor for a product's success. At the same time, the use of social media has become a society-wide phenomenon. These two factors have accelerated the spread of the concept of crowd requirements engineering (CrowdRE). With the help of CrowdRE, requirements engineers are no longer limited to performing only traditional requirements engineering to create the requirements for a product – they are now also supported in paying attention to the information provided by its potential user base. In contrast to giving tasks to the crowd, CrowdRE gives the crowd its own voice [7].

In this context, requirements researchers have started to investigate the potential benefits of performing app store and social media mining to gather new requirements. Broadening the idea beyond requirements, we are not only able to validate, verify, or identify requirements for a product. Applying such mining in a continuous fashion also allows identifying potential bugs in a product. Projects like SUPERSEDE [8] or Opti4Apps [9] focus on the potential of product improvement by systematically considering the crowd and user feedback.

In order to systematically gather, analyze, and provide feedback, different classifications or taxonomies exist. For example, Groen et al. [10] introduced five dimensions: Awareness, mode, data type, intention, and homogeneity, each with two to three sub-categories. This taxonomy was developed with respect to CrowdRE and is based on a literature analysis. Elberzhager and Holl [11] provide another classification with a strong focus on mobile app feedback. This classification was mainly developed to identify different feedback channels and to understand the different kinds of feedback. Considering these classifications, textual feedback, which is the focus of this article, can be understood as explicit feedback. It provides concrete content, for example as written text or through emojis in the text. Textual feedback can be directed at the developers (or the app-developing company in general) but can also be used to share one's experience with other users. Finally, feedback can be very general (e.g., "nice app"), which gives an idea about the general perception, but also very concrete (e.g., "I'd like to have feature X"). These characteristics already show that textual feedback can be of great value for app developers.

2.2 Existing approaches for textual user feedback collection and analysis

In recent years, a lot of publications have focused on textual feedback analysis, its potential, as well as opportunities for analyzing such texts [12]. Within this field, many technical concepts relate more to natural language processing (NLP) than to feedback analysis in parallel with software development. In NLP, sentiment analysis, where (parts of) texts are classified between positive and negative, is a central concept. Another technique is topic modeling, for example to extract features of an app.

Groundbreaking work in sentiment analysis for user feedback was done by Guzman and Maalej [13]. They created an automated approach that analyzes user feedback data from the Apple App Store and Google Play. They applied a mixture of sentiment analysis and topic modeling to extract the features users are talking about. Their goal was to create an analysis solution that shows how people perceive certain features. However, their approach suffers from the problem that infrequently mentioned features are not properly detected. In addition, according to the authors, the third-party lexical sentiment analyzer they used suffers from limitations [13].

Martínez-Cámara et al. [14] created a sentiment analysis for Twitter that combines different sentiment analysis approaches. The combinations used and the way they combine the results of each sentiment analyzer show that multiple classifiers can increase the quality of the result compared to the use of a single one. As their approach also includes machine learning, the proposed ensemble classifier needs to be properly trained. Depending on the app that needs to be analyzed, this training must be repeated.

Vu et al. [15] present PUMA, an approach for extracting user opinions in a phrase-based way. For PUMA, a phrase is "a sequence of consecutive words corresponding to a grammatically correct phrase, clause, or sentence in English" [15]. The authors mention the problem that their approach suffers from the low language quality found in online texts. To detect negative and positive phrases, they assign star ratings to the identified phrases.

In addition to these approaches, many other approaches for analyzing feedback from app stores exist. These approaches can mostly be differentiated by the form of sentiment analysis they apply or by the approach they use for topic modeling; examples are [16], [17], [18], and [19].

Morales-Ramirez et al. [20] proposed a concept for making use of user feedback for the requirements prioritization process. Their aim is to evaluate user feedback in multiple prioritization dimensions and not only by one prioritization type. To realize this concept someday, the authors provide a characterization of the properties of user feedback, including ideas on how the different properties could be extracted from feedback.

In 2017, Guzman et al. [21] proposed ALERTme, an approach that analyzes textual user feedback on software from Twitter. This was one of the first studies focusing on gathering requirements from tweets sent to support accounts of software systems. ALERTme is only capable of classifying, grouping, and ranking tweets. Other social media sources are not supported, nor does the system cover app stores.

An investigation done by Nayebi et al. [6] compares app store feedback and Twitter feedback. As reported by the authors, when evaluated for the same app, these two sources have substantially unique feedback provided in just one of the sources. This means that neither using social media feedback only nor app store feedback exclusively can show the broad picture of user feedback. Considering Twitter in addition to Apple App Store feedback led to 22.4% additional feature requests and 12.89% additional bug reports. This shows that the picture of user feedback becomes more complete when multiple sources are considered.

Palomba et al. [22] present an approach called CRISTAL. The goal is to link user reviews to git commits. In contrast to the approaches mentioned above, CRISTAL does not focus on or provide an analysis of the feedback. The proposed solution is rather intended as a starting point for perceiving feedback as a continuous process to be considered during development.

The analysis of the work being performed in this field shows that many approaches focus more on aspects of automation support for analyzing textual feedback than on software development decision support. This leads to a focus on natural language processing (NLP). This clearly pushes the boundaries of NLP further, but leads to the problem that most studies merely consider a single data source to explore their capabilities in NLP. Unfortunately, research has shown that even though a lot of research has been performed, e.g., in the area of sentiment analysis, the quality of these results still varies to a large extent [23] [24]. Nayebi et al. [6] clearly showed that multiple sources must be considered if feedback analysis is to be embraced in software development practices. In an ideal world such a System should support any source of feedback information.

In addition, it is necessary to consider user feedback not only as a static data set to be analyzed. Due to the nature of mobile development, updates are issued frequently, features are added and changed; the same is the case for user feedback, which is provided continuously by the user base. Scherr et al. argue in [5] that feedback should be captured and analyzed side by side with continuous software evolution. Furthermore Scherr et. al. propose to include emotions of the end user for user feedback analysis [25]. Therefore, feedback analysis is a continuous process. This requires permanent acquisition of feedback to to match iterative changes of the product.

We argue that in order to apply textual user feedback analysis in practice, approaches have to be flexible in terms of data sources. In addition, they have to be able to detect trends and events in data that are related to subsequent changes in a product (other researchers such as Palomba et al. [22] and Hassan et al. [4] also support the goal of continuous feedback gathering).

3 Technical Solution for Feedback Gathering

We propose User Echo Service built with the needs of developers and product managers in mind. It provides the ability to use heterogeneous data sources and periodic data updates to form a data-source-agnostic basis for further analysis of the gathered feedback data. The overall vision is that it should be capable of collecting any kind of textual user feedback independent of where it came from. In addition, this data collection has to be as easy as possible. Therefore, we introduced the concept of crawlers for popular data sources, which can automatically fetch data, and an API for dynamically adding feedback from more individual data sources. These individual data sources could be, for example, a set of private data that is not available publicly, like customer support data. As such data can provide very valuable feedback, it has to be considered in order to get a holistic view on the opinions of the user base.

From a high-level perspective, the UES consists of the data collection infrastructure, the analysis backend, and the web-based front end for real-time analyses and visualization of the analysis results and the collected feedback, as depicted in Fig 1. In the following, we will explain our data collection infrastructure, which is the foundation for the subsequent user feedback analysis not discussed in this article. We will explain how the system copes with the heterogeneous nature of user feedback and the different data sources as well as with permanent monitoring of user feedback.

The data collection infrastructure can support different data sources, from which it can automatically capture textual user feedback. To achieve this, we designed a plugin system where different crawlers can simply be added or exchanged without having to rebuild and deploy the whole system. Therefore, we created an API for crawlers and a data model representing feedback in an abstract manner. An overview of the different components involved in UES is shown in Fig 2. In the next sections, we will present the three major aspects of our data collection infrastructure: the feedback data model, the API for the crawlers, and our scheduling mechanism.

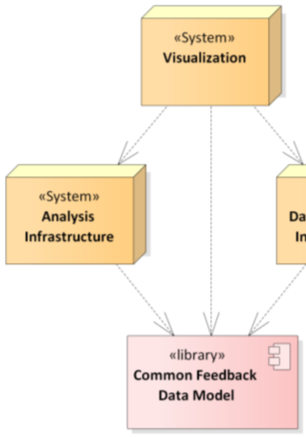

«System» Visualization
«System» Analysis Infrastructure
«System» Data Collection Infrastructure
«library» Common Feedback Data Model

Fig. 1. System overview

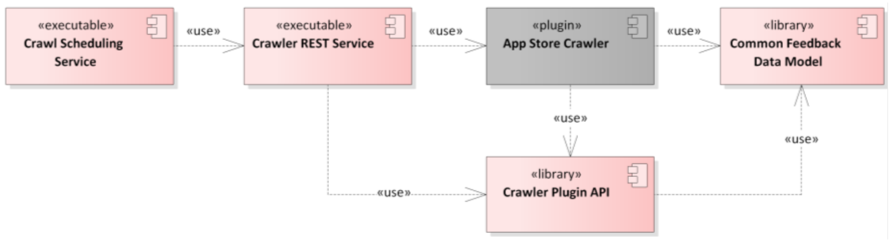

«executable» Crawl Scheduling Service
«use»
«executable» Crawler REST Service
«use»
«plugin» App Store Crawler
«use»
«library» Common Feedback Data Model
«use»
«use»
«use»
«library» Crawler Plugin API

Fig. 2. Functional decomposition of the data collection infrastructure

3.1 Common feedback data model

First, we created a common feedback data model, which is used throughout all components and is therefore located in a shared library. Our main challenge regarding the data model was to support all kinds of possible data sources without losing any relevant data. To come up with this data model, we first examined what kinds of data could be extracted from an initial set of data sources. Data sources explicitly considered for the initial creation were Apple App Store, Google Play, Amazon Marketplace, and Twitter. In addition, one of our requirements was that it should be possible to include different kinds of feedback, meaning individual feedback items like customer support data should be supported. We also considered different analysis possibilities involving these data points. The result was an abstract model containing all the data we need for these analyses, like feedback text or rating. We created a mapping between the crawlable data points and this model in order to maintain traceability to the data source and the semantics. Then we included other data sources and verified that the data model was still compatible with them.

As we want to collect user feedback, the central data structure is the *Review*, which contains the textual feedback of the users on the product (depicted in Fig 3). Some other feedback metrics are also available in the data source, like the five-star rating in the app stores. Such metrics are modeled as the data structures *Rating* and *RatingSetting*. The *RatingSetting* stores information on what kind of *Rating* this is (enumeration *RatingType*). *RatingTypes* encompasses star rating (which is basically any rating where a value between a defined minimum and maximum can be chosen), positive/negative (including neutral), and reaction (which can be, for example, a tag, a sentiment, or an emoji). The actual value is stored in the *Rating*, which references the corresponding shared RatingSetting. *RatingSettings* are defined for each data source. Each *Review* can have several (or no) *Ratings* depending on the data available from the specific data source.

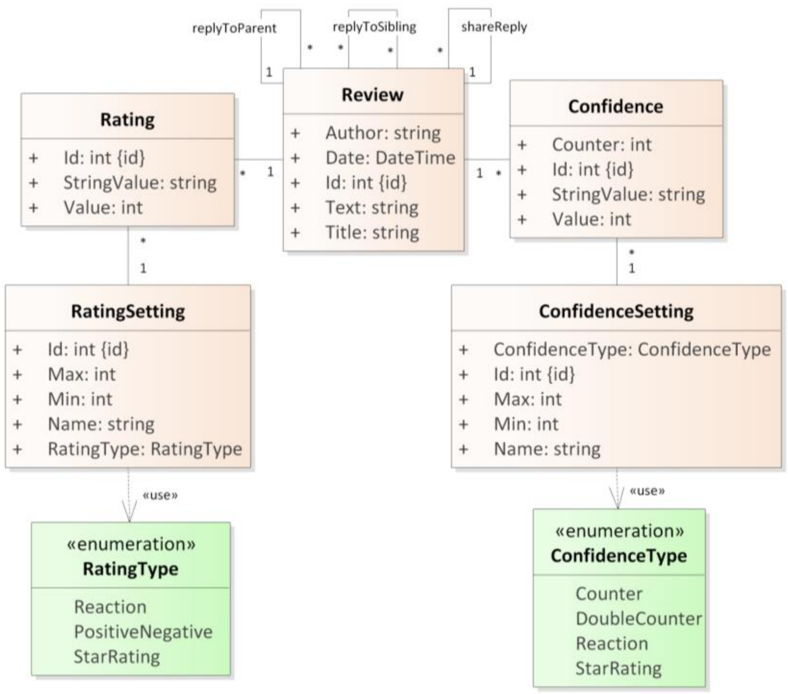


Fig. 3. Data-source-agnostic feedback data model – Review view

Another set of metrics resulted in the data types Confidence and ConfidenceSetting.
These are very similar to *Rating* and *RatingSetting*, but do not represent a user's feedback on a product; rather, they give an indication of how reliable or important this user's feedback is considered by others. An example is Amazon's "helpful review"

functionality, where other Amazon customers can mark a review as helpful and therefore it can be assumed that the review seems convincing, which improves confidence in the review. This kind of confidence measure is represented by the *ConfidenceType DoubleCounter*, with the value being the helpful count and the counter being the total number of votes (helpful + not helpful). Other possible confidence indicators are, for example, the number of retweets or comments on Twitter, which are examples of the *ConfidenceType Counter*. These metrics give an indication of the visibility of the feedback to other users. Facebook's new like-system with different emojis instead of a plain counter is modeled as the *Reaction ConfidenceType*, with the *StringValue* specifying the reaction and the value specifying the number of occurrences.

In many sources, different types of replies to *Reviews* are also possible. We modeled three different types of replies as relationships between *Reviews*. First, we have *replyToParent*, where the reply references the original *Review*. Second, there is *replyToSibling*, indicating a relationship between two *Reviews* on the same hierarchical level, e.g., citing a previous comment in a forum thread. Finally, we have the *shareReply* relationship, which models references to a *Review* without content of its own, but, e.g., with its own set of *Confidence* metrics.

With *Review*, *Rating*, *RatingSetting*, *Confidence,* and *ConfidenceSetting,* we have several means for modeling the user's feedback. However, as we want to have full traceability from the analysis results to the actual feedback on the data source, it is necessary to add references to the data sources and to further structure the feedback. This would also enable us to recrawl specific reviews in case we need to extend the feedback model in the future. Fig 4 depicts the data model used to address this issue.

Product represents an entity for which users give feedback in one or more data sources. An example is the Facebook app, which is available in different stores, such as the Apple App Store, Google Play, or the Microsoft Store. All *Reviews* for this entity are linked to *Product*, so reviews can be found easily by *Product*. *Product* also has a *ProductCategory*. We compared the different categories of multiple app stores and homogenized them into a joint tree of categories.

As we also need to distinguish the data sources from which the feedback is gathered, we introduced *ProductSource* and *SourceType*. *SourceType* models a data source, like the Apple App Store in the United States in English. The language is relevant for conducting some language-dependent analyses. *ProductSource* connects a *Product* with a *SourceType* and holds information on the URL and the identifier of the product in the specific data source, e.g., the package identifier in Google Play ("com.facebook.katana" for Facebook). We further introduced *ReviewSource*, which associates a *Review* with a *ProductSource* and has a URL and an identifier to directly reference the feedback in a data source.

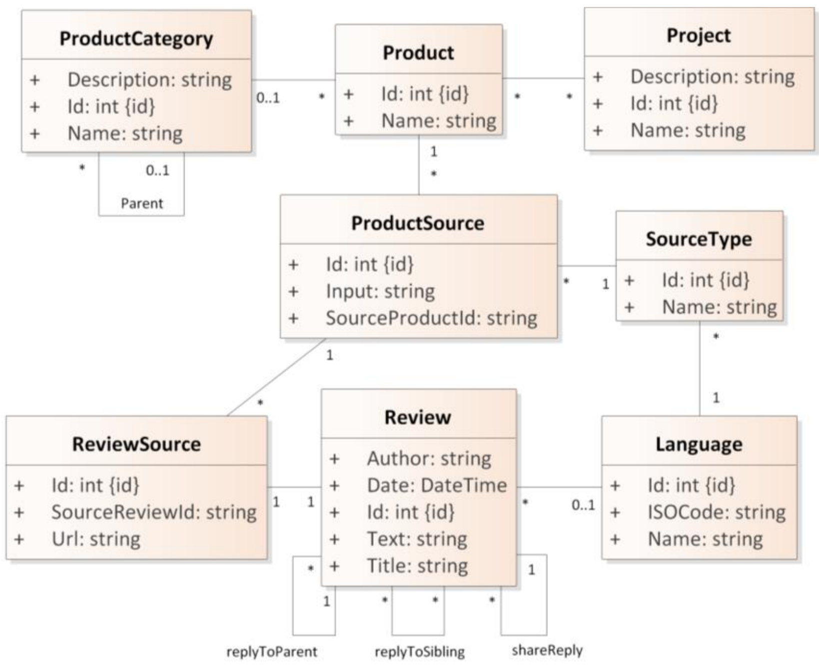


Fig. 4. Data-source-agnostic feedback data model – Product view

We chose Entity Framework and SQL Server as the technologies for creating the code-first data model described above. All changes to the model will be modeled as migrations to ensure data integrity and consistency when we want to extend the model without having to discard already collected feedback.

3.2 Crawlers and crawler plugin API

Once the data model is defined, we can build crawlers for different feedback sources. All crawlers are separate components and use different means to get the data of their data source (e.g., existing APIs from Twitter). Our crawlers also support different functionalities depending on the crawling method used for the source, like crawling a past timespan (start and end in the past), limiting how many reviews should be crawled at most, and approximating the progress of the current crawl. The different means of accessing the data sources, the necessary extensibility to support new data sources, and the need for a central component that controls all crawlers were our main challenges for coming up with a suitable system. Our solution is a plugin system with a specified API for crawlers and a plug-and-play crawler plugin API, which is depicted in Fig 5.

The crawlers can be used by other components via the plugin API, which is packaged as a library and used by the crawlers as well as by the consumers of the plugin. This enables us to work independently on creating different crawlers using different technologies without having a central component that needs to reference all of them.

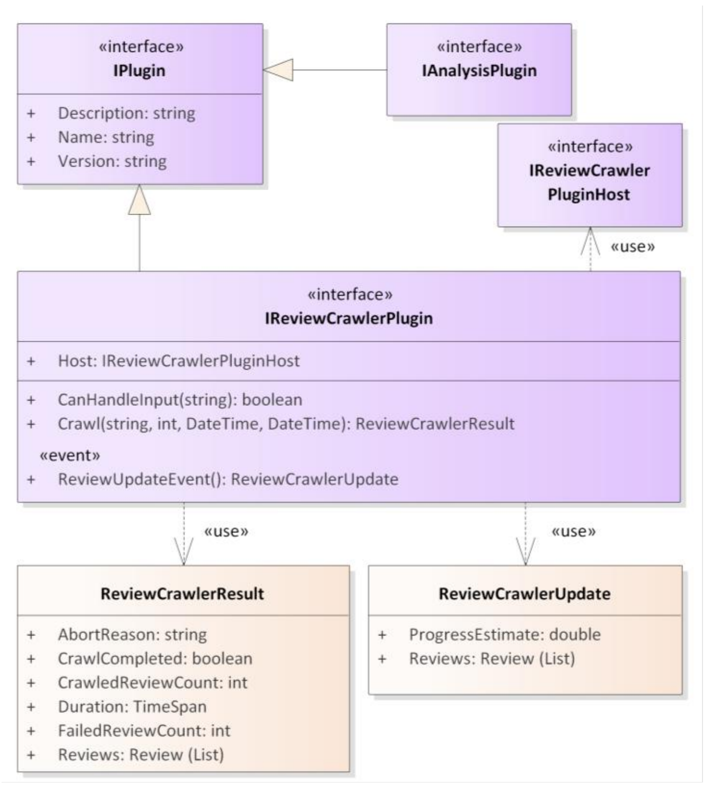


Fig. 5. Fig 1Crawler API and plugin

The API specifies that a crawler can be asked whether it supports the input, and it can be told to start crawling the source, using optional parameters to further limit the crawl, like the start and end timestamps or the maximum number of reviews to crawl. The input can differ a lot and depends on the crawler; it can be, for example, a product ID or a URL on Amazon, or search terms and hashtags on Twitter. As an additional feature, crawlers also send regular updates of the crawl's status and the reviews crawled up to that time using events. If the crawl is aborted, information on the reason and on the progress the crawler has made until it was aborted are returned to the caller. A crawler might abort a crawl, for example, if the data source repeatedly blocks attempts to access data or if usage limits are exhausted. If a crawler is used for a product that does not exist yet, it will also extract all necessary information and create *Product* and *ProductSource* before starting to crawl *Reviews* and related entities.

If no APIs exist, parsing HTML is an alternative. This comes with the risk that the crawler may stop working or may crawl incorrect data, e.g., if the layout of the crawled website changes. To handle this problem, the crawlers perform some checks and will abort if these fail. When using unofficial APIs, we designed our crawlers in a way that should not introduce exceptionally high loads on the target site, e.g., by slowing the crawls down and waiting between single requests to the data source.

As some sources have low usage limits for APIs or start blocking the crawls after some time, it has become apparent that manually starting crawls every couple of days on different products is not effective. The aforementioned measures taken to avoid high loads on the target site have alleviated the problem, but have not completely solved the issue of getting blocked. If we could spread out the crawls for different products on the same data source, our crawls would be much more likely to be successful. We also want to assure that we are gathering all feedback from a source without any gaps, meaning we have to regularly crawl new feedback.

3.3 Scheduling mechanism for periodic crawls

The last remaining major challenges for the data collection infrastructure are related to regularly crawling data for all products we are interested in. We have to consider the request limits of the different data sources and the varying rates between products of new collectable feedback per data source, and we must be able to handle aborted or unsuccessful crawls. We tackled these problems by designing a scheduling mechanism that enables periodic crawls. Our crawl scheduling service lets us define which product should be crawled on which data source how often, so we can fine-tune the schedule to the rate of new feedback. This extends the previous ability of manually starting a crawl for a delta update on the data. To have a history of the crawls, we introduced the *CrawlRun* data type, which stores information on when a crawler has started and finished, what it crawled, and what the result of the crawl was. The *CrawlSchedule* data type defines the *ProductSource* (and thus the *Product*) to work on, the maximum number of reviews to crawl, and the timespan between the start times of the crawls in minutes. It also holds the next scheduled crawl time, which can be modified to spread out crawls on the same data source. The CrawlerConfiguration type holds the information on the connection and the setting for the crawler itself. Location and IsRemote determine the local path or URL of the crawler. CooldownMinutes defines the minimum time after each crawl of the referenced crawler before it can crawl again, delaying crawls scheduled for it. MaximumCount defines the maximum number of reviews to be crawled at once. SupportedSourceTypeName refers to the SourceType.Name from the common feedback data model and is the connection between the CrawlSchedule and the CrawlerConfiguration via the ProductSource. This model is shown in Fig 6. The scheduler stores its data in its own database, which is not directly connected to the database holding the common feedback data model.

To make it easier to optimize the crawls, we introduced a functionality to distribute them for given timespans. Crawls to different data sources may be executed in parallel, as these crawls do not influence each other. The scheduling service runs continuously and utilizes a task scheduling mechanism set to the next execution time of any crawl to start the respective crawl.

To be able to deploy the crawlers on several different machines, we created a standardized web service executable that uses the aforementioned crawler plugin API to utilize any crawler's capabilities. Conceptually similar to the web service approach, we also developed a command line interface for the crawlers that is more convenient to use if crawling on a local machine. We use this, for example, when crawling a new product that does not have a *CrawlSchedule* yet or does not need one.

Our crawl-scheduling service is a web service that offers a REST API to manage schedules and provide information on connected crawlers. It furthermore provides a web-based set of forms for creating, listing, modifying, and deleting schedules and upcoming run times. The crawlers can be accessed using the aforementioned web services, whose URLs are configured in the scheduling service.

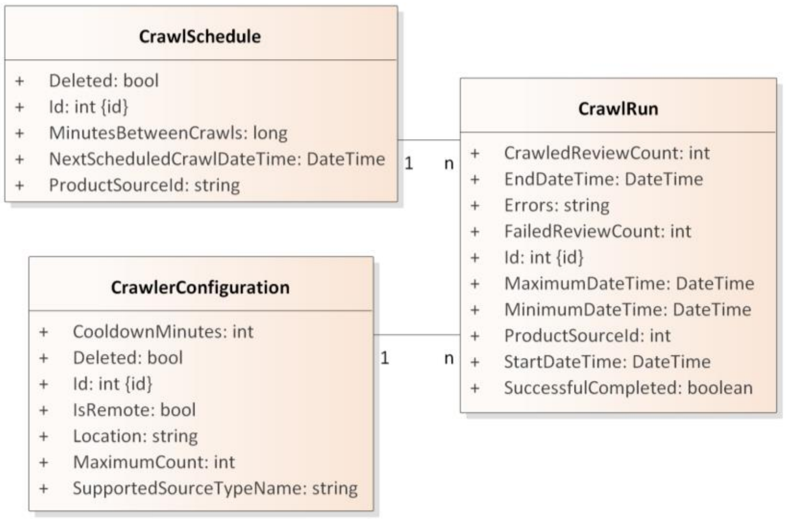


Fig. 6. Fig 2Crawl scheduler model

As we explored several data sources, we also came across sources that cannot be effectively and efficiently crawled by machines, but might still be interesting for inclusion in our data collection. Some of these sources are, for example, custom websites and forums, but also graphical content like pictures and videos. For such cases, another tool was developed to provide a simple UI for manually entering data into the database according to our model. Such entries are marked, so if we later develop a crawler for that source, we can distinguish them from automatically crawled entries.

All this has resulted in a growing collection of feedback on several products in our database. However, this collection has to be processed further to provide further insights. To enable continuous analysis of the crawled feedback, we extended the crawl-scheduling service in order to be able to notify other components that new feedback has been crawled so they can start analyzing it. We have already begun to develop an extensible analysis infrastructure for the tool, which will support continuous process feedback by applying different, partly interdependent, analysis steps while always preserving traceability to the source data.

4 Evaluation

In this section, we will present the evaluation of our data acquisition approach. We will first show how we map our data model to different data sources. Then we will provide an example of how UES can be used in practice to make product owners aware of the latest data.

4.1 Mapping data sources to our data model

In the following, we will show that our data model (see Fig 3) is capable of handling different data sources. Currently we have added support for the stores Amazon Marketplace, Google Play, and Apple App Store. In addition to that, we have an implementation for the social media data sources Twitter, Instagram, and Facebook, including a former crawler to the now closed Google+. We will conceptually cover how we have currently mapped the data sources implemented as crawlers and the crawlers under development, and describe the manual pasting functionality of the system. This will show that even though the model is general and abstracted from concrete data sources, we are able to host different data sources while keeping the detailed data they provide for user feedback. We will skip the attribute *Id* for each entity as it is only used by our database to maintain primary keys and foreign key relationships.

Project, *Product*, and *ProductCategory* are independent of any data source as they are purely related to the app or set of apps whose feedback is being monitoring. *ProductSource*, *SourceType*, and *ReviewSource* are entities that are set based on the data source and are only used to identify the data being retrieved; they do not contain the actual feedback data.

The mapping of the data provided by a data source to the data model itself starts with the *Review* entity, as this entity is our foundation for representing feedback. For the Apple App Store, Amazon Marketplace, and Google Play, the mapping of the attributes is similar and covers all attributes. We get a non-unique identifier for the *Author* from the Apple App Store and Google Play, which preserves the anonymity of the user, but from Amazon Marketplace, we get the actual user id. The data we get from the source for *Date* only contains the date but not a time component for the app stores. *Text* and *Title* for *Review* are set in a straightforward manner.

Our social media sources Twitter, Instagram, and Facebook behave differently. They provide a unique identifier for the *Author*, and the *Date* of the posting also contains the exact timestamp. As these sources only allow texts without titles, the title attribute is always empty and only the text is set. In the case of a multimedia posting, we are not able to capture the multimedia content of the source, only the textual description. Manually added feedback supports all entities, but the degree of specificity depends on the data. An email to customer support might contain data with all attributes fully set, but feedback derived from a phone call might just contain a date and a textual summary. It is mostly up to the developer or customer support team how they fill out the form.

For the *Language* entity, Google Play provides this information, as the store itself is divided by language codes. Therefore, the language is fixed with the crawl. As the Apple App Store as well as Amazon Marketplace are divided by country, we could assume that the language of the feedback is the primary language of that country. To get a more precise classification, however, it is more appropriate to use language detection frameworks based on the text and titles submitted in a review and set the optional *Language* in the *Review* accordingly. The latter is necessary for all social media sources as they do not offer any division by language or country at all.

The ratings available for a review are individual for each data source. First of all, social media sources currently in use have no means of providing a direct rating for a product. Ratings exist primarily in store scenarios. Apple App Store, Amazon Marketplace, and Google Play currently have a five-star rating system. Our system maps this rating as shown in the following example. A *Review* only has one *Rating*, with a *Value* between one and five. The corresponding *RatingSetting* shows that this is a *Rating* with a *Max* of five and a *Min* of one. The name is "five-star-rating" and the *RatingType* is set to *StarRating*. Manually added data in the form of our inserter also supports ratings, although it depends on the kind of feedback entered whether there is relevant data.

For the *Confidence* entity, the Apple App Store does not provide any information. Amazon Marketplace contains a feature showing whether a review was helpful or not. We capture this as *DoubleCounter Confidence*, with the counter combining the helpful and unhelpful votes and the value containing the helpful votes. In Google Play, only helpful votes are counted, so this is only a normal *Counter*.

Social media sources come with several potential confidence metrics. Twitter, for example, offers the ability to express likes with a heart or share the same thought in the form of a retweet. The number of likes and retweets shows how many people potentially agree with the posting. Therefore, we capture both in the form of *Confidence* expressed as a *Counter*. In addition, Twitter offers commenting on posts (replies), which also indicates the interest of people in a posting. In contrast to likes and retweets, however, comments do not directly signal agreement or disagreement. In the end, the number of comments is implemented in the form of *Counter Confidence*. The same is the case for the comment counts on Facebook and Instagram.

Like Twitter, Facebook shows the number of shares of a post. Instagram does not offer such a counter. Both platforms also support forms to express one's attitude towards a posting. In the case of Instagram, a like feature is available similar to the one on Twitter. For Facebook, this form of confidence is more complex, expressed in the form of *Reactions*, including Love, Haha, Wow, Sad, Angry, and the traditional Like.

The *replyToParent* relationship can be found in forums (thread start and replies), on Facebook, and on Instagram as comments/replies to a post and on Twitter as retweets with the user's own commentary ("RT <Twitter-url>"). The *replyToSibling* relationship is mostly present in forums, where in a reply to a thread one can usually cite multiple comments in a thread. The *shareReply* can be found in all social networks, e.g., on Twitter as a retweet (without the user's own commentary) or on Facebook as a share.

The mapping shows that we can currently express the two most popular app stores, Google Play and Apple App Store, with our model. In addition, Amazon Marketplace is a third store offering Android apps and apps for Kindle Fire. From the perspective of social media, we have mapped postings from Facebook, Twitter, and Instagram. Looking at the number of users on these platforms, we should cover a total of more than 3.5 billion active users [26]. This shows not only the flexibility of the approach but also the fact that we support a huge share of the social network market.

4.2 Using the data collection infrastructure in practice

One major goal of our solution is is that the end user does not interact directly with the data collection infrastructure. Instead, we added the management capabilities for it to our web-based visualization. This makes it easy to use, as the product manager does not need to use the REST API directly, but can use it in a browser. The design aims at providing a simple user interface for management, as the product manager should be able to focus on the analyzed data for their product and not on configuring the tool. A new product can be easily added directly on the landing page. The only elements that have to be specified at this stage are the name and the category of a product (Fig 7).

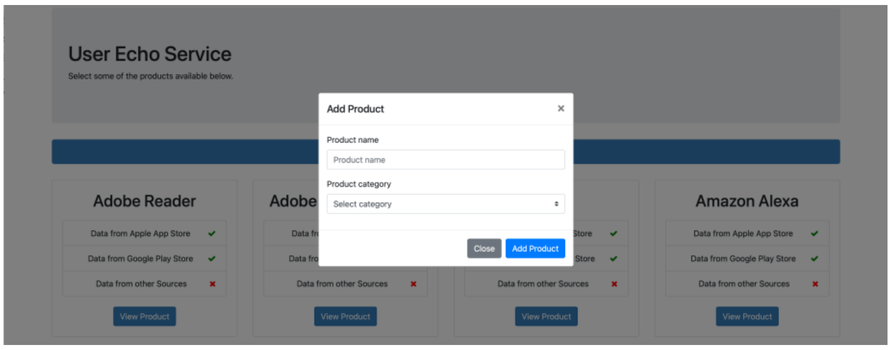
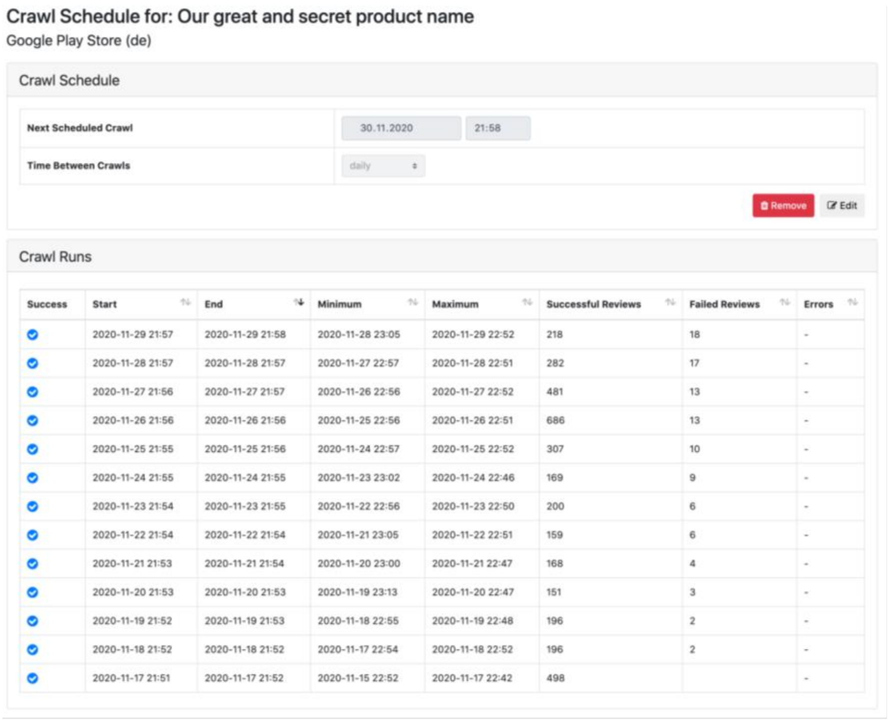


Fig. 7. Adding a product to our tool

The product sources and the corresponding schedules can be maintained in the settings tab of each product. Here, product sources can be added or altered. Once a product source contains raw data, it cannot be deleted anymore. On the same screen, the product manager can create, alter, and delete schedules for the product sources in order to get the latest data at the desired intervals.

The details section of a schedule shows information on when the next crawl for this source is scheduled and what the desired interval is (Fig 8). Both values can be altered here. However, no immediate manual update of the data right from the dashboard is possible. Changing the next interval shifts the schedule. An ‘update now’ feature is planned for inclusion in a future release.

Next to the details, the user sees the history of frequent crawls. Information is shown on when the crawl took place and from which timeframe the feedback was, as well as the total amount of feedback collected, divided into successfully crawled, crawling failed, and number of errors. This history gives the user insights into whether a higher frequency of updates would make sense.



Crawl Schedule for: Our great and secret product name

Google Play Store (de)

Crawl Schedule

Next Scheduled Crawl	30.11.2020 21:58
Time Between Crawls	daily

Crawl Runs

Success	Start	End	Minimum	Maximum	Successful Reviews	Failed Reviews	Errors
✓	2020-11-29 21:57	2020-11-29 21:58	2020-11-28 23:05	2020-11-29 22:52	218	18	-
✓	2020-11-28 21:57	2020-11-28 21:57	2020-11-27 22:57	2020-11-28 22:51	282	17	-
✓	2020-11-27 21:56	2020-11-27 21:57	2020-11-26 22:56	2020-11-27 22:52	481	13	-
✓	2020-11-26 21:56	2020-11-26 21:56	2020-11-25 22:56	2020-11-26 22:51	686	13	-
✓	2020-11-25 21:55	2020-11-25 21:56	2020-11-24 22:57	2020-11-25 22:52	307	10	-
✓	2020-11-24 21:55	2020-11-24 21:55	2020-11-23 23:02	2020-11-24 22:46	169	9	-
✓	2020-11-23 21:54	2020-11-23 21:55	2020-11-22 22:56	2020-11-23 22:50	200	6	-
✓	2020-11-22 21:54	2020-11-22 21:54	2020-11-21 23:05	2020-11-22 22:51	159	6	-
✓	2020-11-21 21:53	2020-11-21 21:54	2020-11-20 23:00	2020-11-21 22:47	168	4	-
✓	2020-11-20 21:53	2020-11-20 21:53	2020-11-19 23:13	2020-11-20 22:47	151	3	-
✓	2020-11-19 21:52	2020-11-19 21:53	2020-11-18 22:55	2020-11-19 22:48	196	2	-
✓	2020-11-18 21:52	2020-11-18 21:52	2020-11-17 22:54	2020-11-18 22:52	196	2	-
✓	2020-11-17 21:51	2020-11-17 21:52	2020-11-15 22:52	2020-11-17 22:42	498		-

Fig. 8. Crawl schedule status and editing view

5 Discussion

With our data collection infrastructure, we have created a flexible solution that is able to handle very different data sources. Our approach introduces and combines several new aspects for collecting data, like having the described flexibility, our data-agnostic understanding of the data, and the means to continuously process new data. The current implementation supports the Apple App Store, Google Play, and Amazon Marketplace as store sources. In addition, we have added the social media data sources Facebook, Twitter, and Instagram. We also had a crawler for the social network Google+, which is now shut down. Crawlers supporting YouTube and phpBB are currently under development and will also suit the API and data model. In addition to automated crawlers, we have also added a module where developers can manually insert feedback, e.g., from customer support. The range of sources currently supported and the ease of including new sources shows the flexibility of this approach.

If the approach presented here is embedded into the software engineering process, it can lead to continuous collection and analysis of user feedback. Due to its ability to handle data sources in the form of installable plugins as well as the ability to manually add raw data, acquiring a broad range of feedback is possible. By enabling a continuous and comprehensive stream of feedback, we offer a way to collect feedback from many sources continuously in order to improve the product under development.

Continuous analysis of user feedback is facilitated by UES right from the crawling step, as well as for each data source and query in a data source. Once the crawling intervals have been set up initially, the system handles the updating of the data. When new data is received, the product manager sees immediately the updated data. The same can be done for the analysis. By extending the system with an analysis framework that can continuously analyze newly collected data, we can provide faster and more complex insights into the user feedback. Such a framework should support different steps of interdependent analyses to provide flexibility while keeping the system complexity low. By continuously streaming new data into such a framework or analysis pipeline the degree of automation is increased substantially.

When collecting feedback continuously, the data set will grow so much that an experience database emerges. Teams will learn from past events and be able, for example, to identify failure patterns, which can then be checked in their test runs. These patterns can be derived from lessons learned about the analyzed detected events. In the case of bug and crash reports, developers have a growing database of checks they can introduce to prevent issues from being reintroduced in subsequent releases.

Switching the perspective from issues to events, showing ideas for feature requests or change requests as well for appreciating feedback, enables developers to extend, update, verify, and validate their requirements. Such an event detection can be implemented as one of the analyses embedded in the analysis pipeline. By doing this consistently in the development process, a product will subsequently match the users' expectations better. This should lead to a higher product acceptance rate.

As our approach allows collecting and detecting feedback in an automated way, it offers benefits for developers without creating additional effort for them. The ability of our web-based dashboard to fully customize when each data source of a product should be updated as well as the history of recent crawls give the product manager full control and transparency of the data acquisition process.

One limitation of our approach is that developers have to ensure that they are listening to the right feedback channels. Collecting feedback from app stores is straightforward, as it is given directly on the app. As soon as social media or online forums come into play, however, feedback is spread around the data source. It is not sufficient to just collect the feedback sent directly to the developer's social media account. In addition, a collection of hashtags – at least the product name – often also contain relevant feedback. Therefore, developers have to carefully set up the terms they are interested in on social media. The better they deal with the selection of the search terms, the more complete the picture of the feedback will be.

6 Conclusion

In this article, we presented parts of User Echo Service. We developed a data collection infrastructure that can handle various data sources by maintaining a single data structure. The data model abstracts from the actual data being retrieved and clusters it into different entities. New data sources can be added by creating a library following our plugin API for crawlers. In addition, the infrastructure supports changing the data sources as well as updating the data from a data source in an automated way by using a scheduling mechanism. As data sources are exchangeable for us, the crawling states have to be maintained for each data source rather than for the entire product. In addition to the six different crawlers available so far, we can manually add feedback not captured by our crawlers. This is used, for instance, to manually add customer support requests and allows us to capture feedback in the sources where the users of a certain app are giving it. This feature will be extended even more, as additional crawlers are currently under development to access additional social media sources. Even though the schedulers provide a comfortable way to get the latest data at the desired interval, it might be necessary to get a data update immediately. A future release of the dashboard will contain a trigger for the crawling system to manually update a data source. This will be done by scheduling a one-time crawl as soon as possible.

As apps are usually developed in fast and highly iterative development cycles, changes are introduced within short periods. Instead of trying to perform a full analysis, our approach focuses on the detection of trends. These trends can be detected with our event detection. To be able to catch short-term trends as well as longer ones, we apply different granularities for the time range to be considered. This keeps developers always up to data with the latest changes in the data. Analyzing feedback continuously along with performing event detection also allows comparing different releases and features. This makes it easier to detect how a series of changes to a product over multiple releases is perceived by the end users of the application. Future work will focus especially on increased automation of the trend detection.

7 Acknowledgements

The research described in this paper was performed in the project EnStadt: Pfaff (grant no. 03SBE112D and 03SBE112G) of the German Federal Ministry for Economic Affairs and Energy (BMWi) and the Federal Ministry of Education and Research (BMBF). We thank Sonnhild Namingha for proofreading.

8 References

[1] L. W. Khong, L. Y. Beng, T. W. Yip and T. S. Fun, "Software Development Life Cycle AGILE vs Traditional Approaches," in International Conference on Information and Network Technology, Chennai, India, 2012.

[2] W. Maalej, M. Nayebi, T. Johann and G. Ruhe, "Toward Data-Driven Requirements Engineering," IEEE Software, vol. 33, no. 1, pp. 48 - 54, 2015. https://doi.org/10.1109/ms.2015.153

[3] A. Strzelecki, "Application of Developers' and Users' Dependent Factors in App Store Optimizatio," International Journal of Interactive Mobile TechnologiesOpen Access , vol. 134, no. 3, pp. 91-106, 2020. https://doi.org/10.3991/ijim.v14i13.14143

[4] S. Hassan, C. Tantithamthavorn, C.-P. Bezemer and A. E. Hassan, "Studying the dialogue between users and developers of free apps in the Google Play Store," Empirical Software Engineering, vol. 23, no. 3, pp. 1275–1312. https://doi.org/10.1007/s10664-017-9538-9, 2018. https://doi.org/10.1145/3180155.3182523

[5] S. A. Scherr, F. Elberzhager and S. Meyer, "Listen to Your Users - Quality Improvement of Mobile Apps through Lightweight Feedback Analyses," in International Conference on Software Quality, Vienna, 2019. https://doi.org/10.1007/978-3-030-05767-1_4

[6] M. Nayebi, H. Cho and G. Ruhe, "App store mining is not enough for app improvement," Empirical Software Engineering, vol. 23, no. 5, p. 2764–2794, 2018. https://doi.org/10.1007/s10664-018-9601-1

[7] E. C. Groen, N. Seyff, R. Ali, F. Dalpiaz, J. Doerr, E. Guzman, M. Hosseini, J. Marco, M. Oriol, A. Perini and M. Stade, "The Crowd in Requirements Engineering," IEEE Software, vol. 34, no. 2, pp. 44-52, 2017. https://doi.org/10.1109/ms.2017.33

[8] S. N. Francesch, A. A. Gamazo, Ó. R. Moral and J. Varga, "Big data management challenges in SUPERSEDE," in Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference, Venice, Italy, 2017.

[9] S. A. Scherr, F. Elberzhager and K. Holl, "An automated feedback-based approach to support mobile app development," in Proceedings - 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017, Vienna, 2017. https://doi.org/10.1109/seaa.2017.45

[10] E. C. Groen, J. Doerr and S. Adam, "Towards crowd-based requirements engineering. A research preview," in International Working Conference on Requirements Engineering: Foundation for Software Quality, Essen, Germany, 2015. https://doi.org/10.1007/978-3-319-16101-3_16

[11] F. Elberzhager and K. Holl, "Towards Automated Capturing and Processing of User Feedback for Optimizing Mobile Apps," Procedia Computer Science, vol. 110, pp. 215-221, 2017. https://doi.org/10.1016/j.procs.2017.06.087

[12] N. Jha and A. Mahmoud, "Mining User Requirements from Application Store Reviews Using Frame Semantics," in Requirements Engineering: Foundation for Software Quality, 2017, Essen, Germany. https://doi.org/10.1007/978-3-319-54045-0_20

[13] E. Guzman and W. Maalej, "How Do Users Like This Feature? A Fine Grained Sentiment Analysis of App Reviews," in 2014 IEEE 22nd International Requirements Engineering Conference (RE), Karlskrona, Sweden, 2014. https://doi.org/10.1109/re.2014.6912257

[14] E. Martínez-Cámara, Y. Gutiérrez-Vázquez, J. Fernández, A. Montejo-Ráez and R. Muñoz-Guillena, "Ensemble classifier for Twitter Sentiment Analysis," in Proceedings of the Workshop on NLP Applications: Completing the Puzzle co-located with the 20th International Conference on Applications of Natural Language to Information Systems, Passau, Germany, 2015.

[15] P. Vu, H. V. Pham, T. T. Nguyen and T. T. Nguyen, "Phrase-based extraction of user opinions in mobile app reviews," in 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, Singapore, 2016. https://doi.org/10.1145/2970276.2970365

[16] W. Maalej, Z. Kurtanović, H. Nabil and C. Stanik , "On the automatic classification of app reviews," Journal Requirements Engineering, vol. 21, no. 3, pp. 311-331 , 2016. https://doi.org/10.1007/s00766-016-0251-9

[17] J. Huebner, R. M. Frey, C. Ammendola, E. Fleisch and A. Ilic, "What People Like in Mobile Finance Apps: An Analysis of User Reviews," in Proceedings of the 17th International Conference on Mobile and Ubiquitous Multimedia, Cairo, Egypt , 2018. https://doi.org/10.1145/3282894.3282895

[18] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora and H. Gall, "How can I improve my app? Classifying user reviews for software maintenance and evolution," in IEEE International Conference on Software Maintenance and Evolution, Bremen, Germany, 2015. https://doi.org/10.1109/icsm.2015.7332474

[19] 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, San Francisco, CA, USA, 2013. https://doi.org/10.1109/msr.2013.6624001

[20] I. Morales-Ramirez, D. Munante, F. Kifetew, A. Perini, A. Susi and A. Siena, "Exploiting User Feedback in Tool-Supported Multi-criteria Requirements Prioritization," in IEEE 25th International Requirements Engineering Conference (RE), Lisbon, Portugal, 2017. https://doi.org/10.1109/re.2017.41

[21] E. Guzman, M. Ibrahim and M. Glinz, "A Little Bird Told Me: Mining Tweets for Requirements and Software Evolution," in IEEE 25th International Requirements Engineering Conference (RE), Lisbon, Portugal, 2017. https://doi.org/10.1109/re.2017.88

[22] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk and A. De Lucia, "Crowdsourcing User Reviews to Support the Evolution of Mobile Apps," Journal of Systems and Software, no. March 2018, pp. 143-162. https://doi.org/10.1016/j.jss.2017.11.043

[23] A. Hogenboom, M. Bal, F. Frasincar and D. Bal, "Towards cross-language sentiment analysis through universal star ratings," Advances in Intelligent Systems and Computing, vol. 172, pp. 69-79, 2013. https://doi.org/10.1007/978-3-642-30867-3_7

[24] F. N. Ribeiro, M. Araújo, M. A. Gonçalves and F. Benevenuto, "SentiBench - a benchmark comparison of state-of-the-practice sentiment analysis methods," EPJ Data Sci, vol. 5, no. 1, 2016. https://doi.org/10.1140/epjds/s13688-016-0085-1

[25] S. A. Scherr, S. Polst, L. Müller, K. Holl and F. Elberzhager, "The Perception of Emojis for Analyzing App Feedback," International Journal of Interactive Mobile Technologies, vol. 13, no. 2, 02 2019. https://doi.org/10.3991/ijim.v13i02.8492

[26] "Most popular social networks worldwide as of October 2018, ranked by number of active users (in millions)," Statista, 11 2018. [Online]. Available: https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/. [Accessed 30 01 2019].

[27] J. A. Chevalier and D. Mayzlin, "The Effect of Word of Mouth on Sales: Online Book Reviews," Journal of Marketing Research, vol. 43, no. 3, pp. 345-354, 2006. https://doi.org/10.1509/jmkr.43.3.345

[28] N. Hu, P. A. Pavlou and J. Zhang, "Can online reviews reveal a product's true quality?: empirical findings and analytical modeling of Online word-of-mouth communication," in Proceedings of the 7th ACM conference on Electronic commerce, Ann Arbor, Michigan, USA, 2006. https://doi.org/10.1145/1134707.1134743

9 Authors

Simon André Scherr is a senior engineer at the Fraunhofer Institute for Experimental Software Engineering IESE in the field of UX. His current research interest is how to increase long-term product acceptance by analyzing emotions found in feedback. With emoji-poll.de/en, his research has collected one of the largest data sets about the perception of emojis.

Steffen Hupp is a senior engineer at the Fraunhofer Institute for Experimental Software Engineering IESE in the field of User Experience and Requirements Engineering. His current focus as full stack developer is on the development of mobile software solutions, DevOps and backend development.

Frank Elberzhager is an expert for software quality at the Fraunhofer Institute for Experimental Software Engineering IESE. He received his PhD in Computer Science from the University of Kaiserslautern, Germany, in 2012. His research interests include software quality assurance, inspection and testing, software engineering processes, and software architecture. He also transfers research results into practice.

Article submitted 2020-12-18. Resubmitted 2021-02-15. Final acceptance 2021-02-16. Final version published as submitted by the authors.